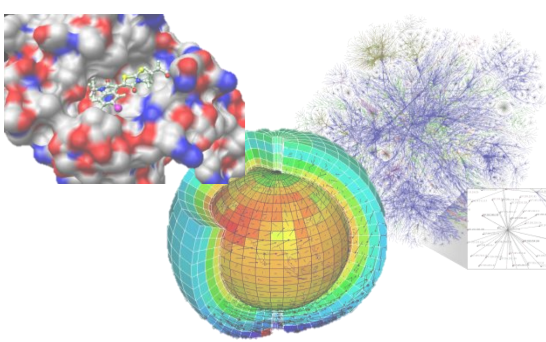
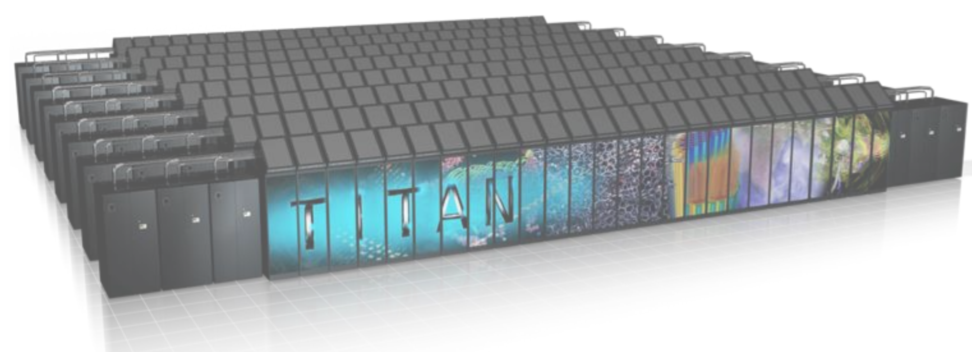
Computing on the Lunatic Fringe:
Exascale Computers and Why You Should Care

Mattan Erez
The University of Texas at Austin





Arch-focused whole-system approach

Efficiency requirements require crossing layers

Algorithms are key

- Compute less, move less, store less

Proportional systems

- Minimize waste

Utilize and improve emerging technologies

Explore (and act) up and down

- Programming model to circuits

Preferably implement at micro-arch/system

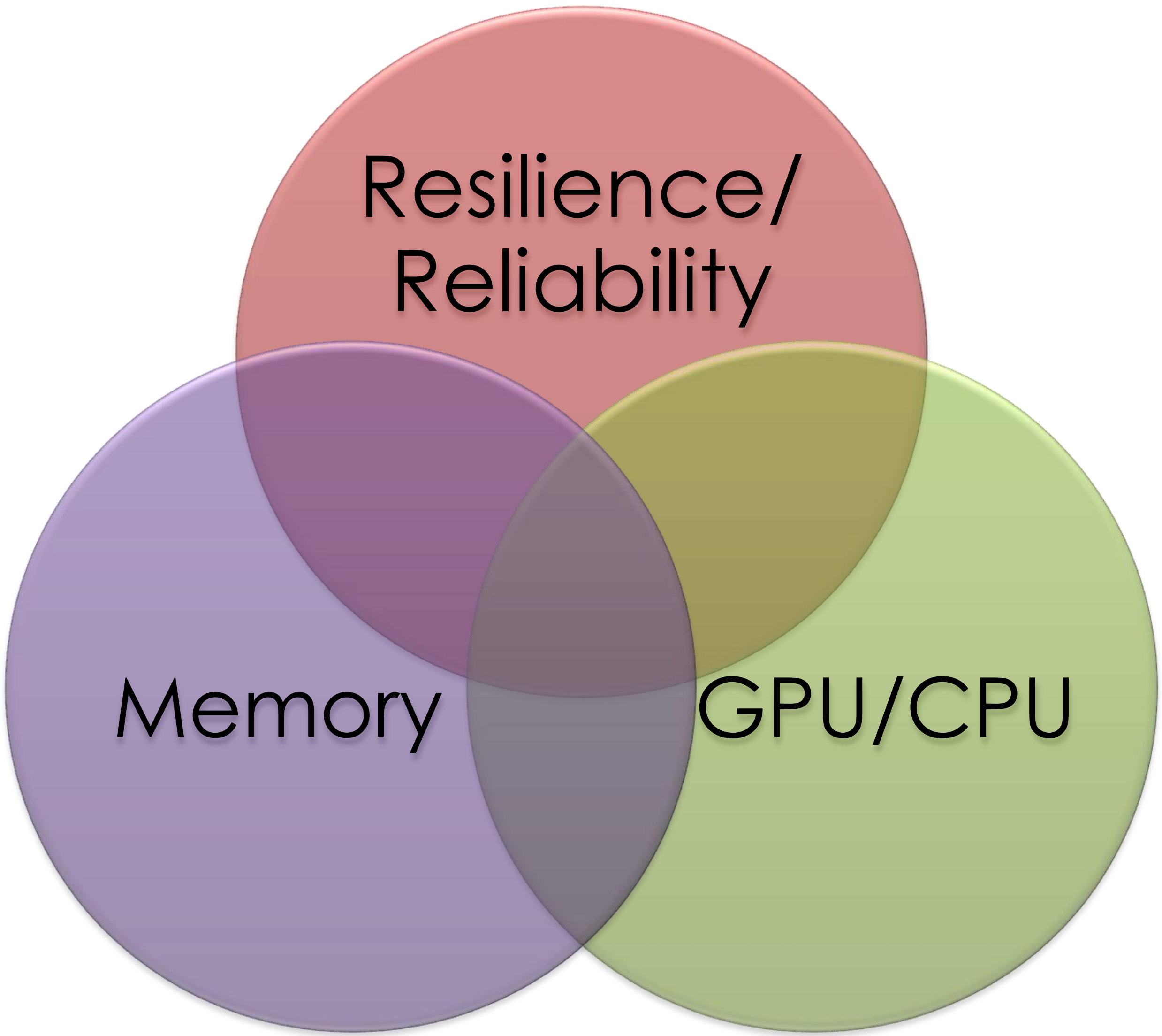
Big problems and emerging platforms

Memory systems
- Capacity, bandwidth, efficiency – impossible to balance
- Adaptive and dynamic management helps
- New technologies to the rescue?
- Opportunities for in-memory computing

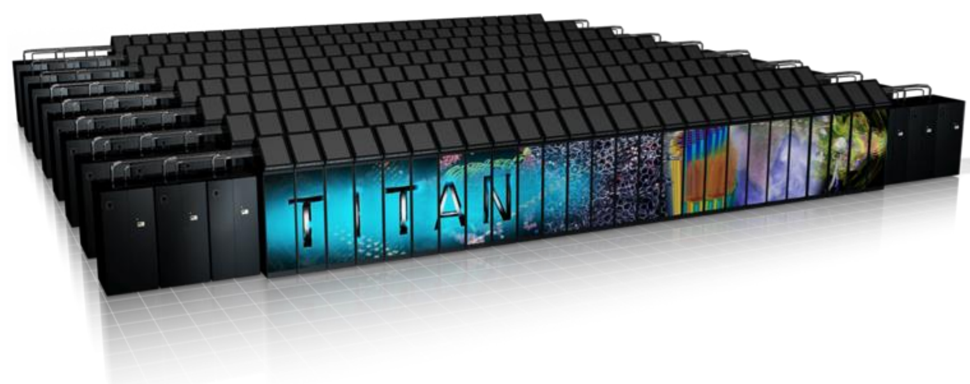
GPUs, supercomputers, clouds, and more
- Throughput oriented designs are a must
- Centralization trend is interesting

Reliability and resilience
- More, smaller devices – danger of poor reliability
- Trimmed margins – less room for error
- Hard constraints – efficiency is a must
- Scale exacerbates reliability and resilience concerns

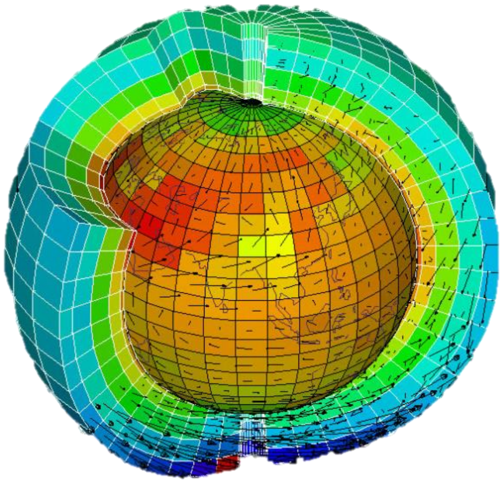
Lots of interesting multi-level projects

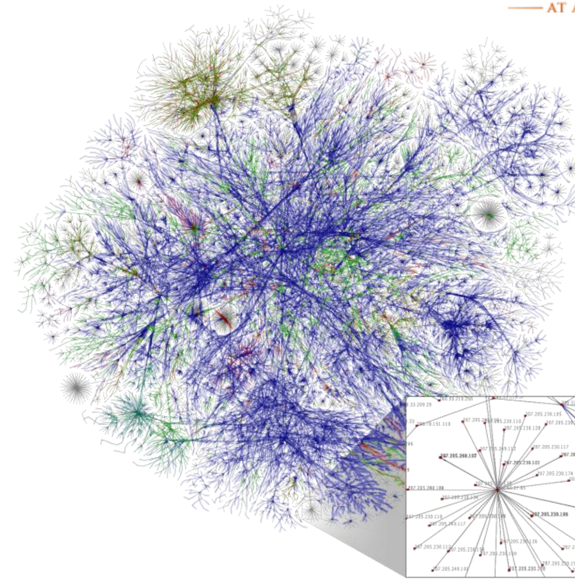


A **superscale computer** is
a high-performance system for
solving **big problems**

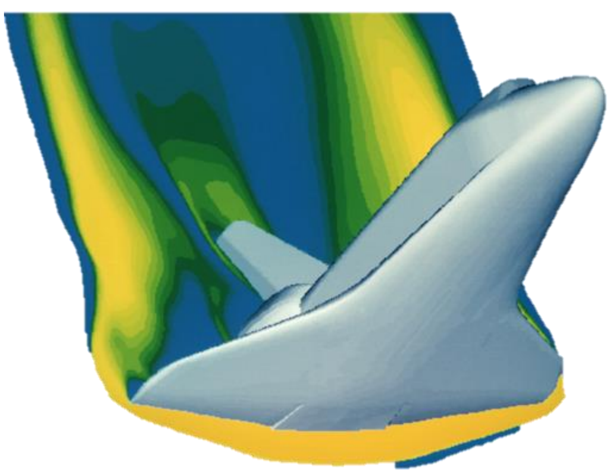


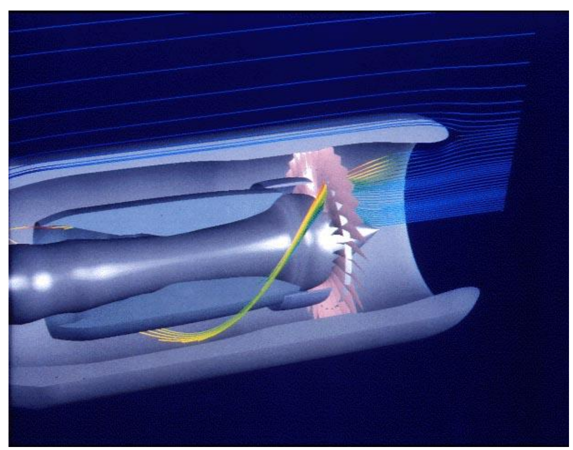
A **supercomputer** is a high-performance system for solving **big cohesive problems**

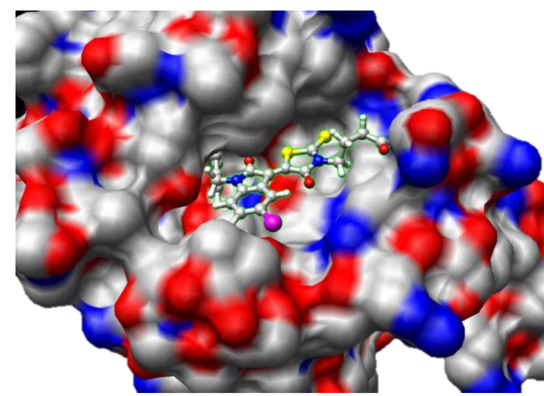




A **supercomputer** is a high-performance system for solving **big cohesive problems**







Simulate

Analyze

Predict

Simulate

Analyze

Predict

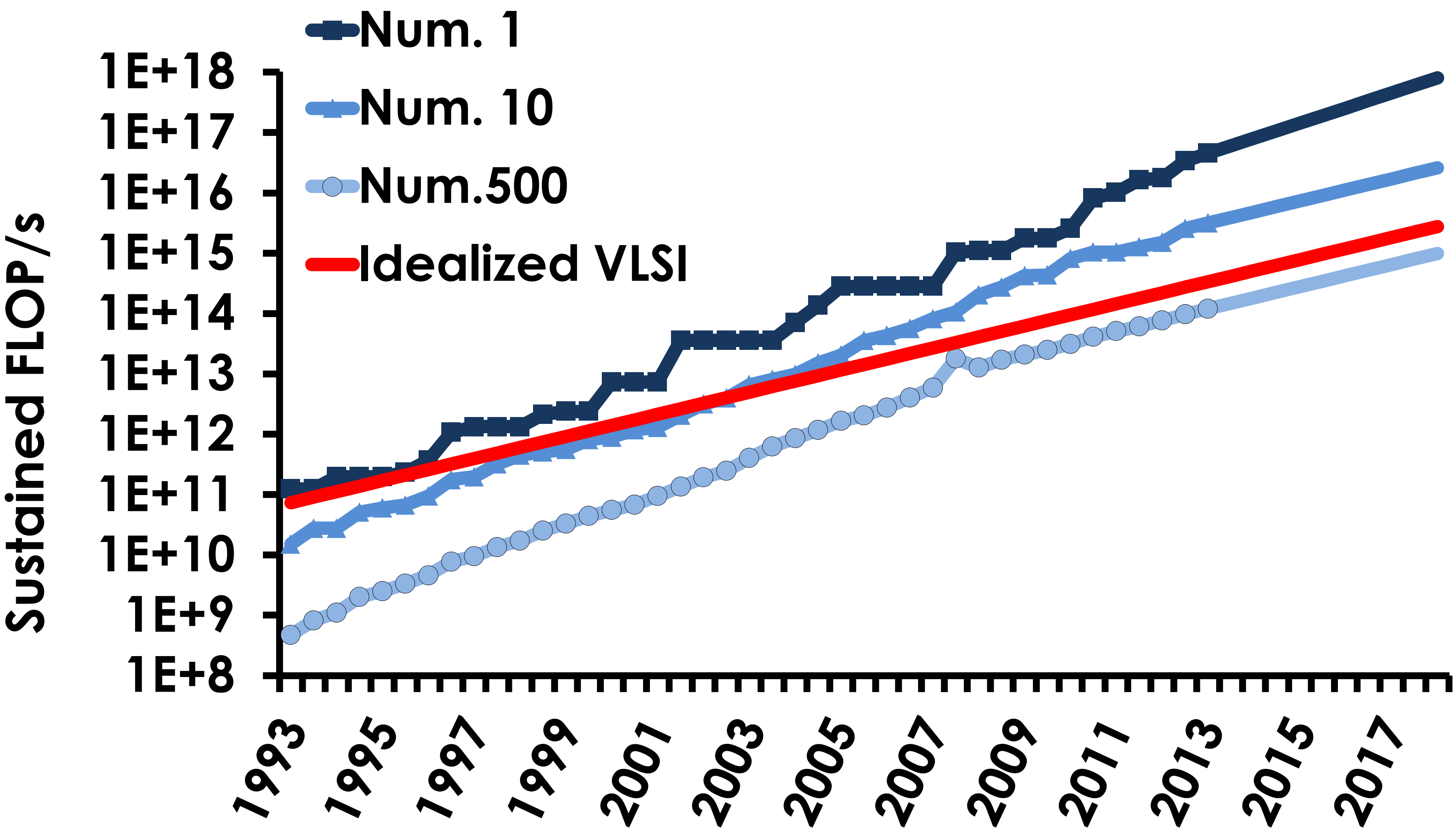
Supercomputers are general
- Not domain specific

Cloud computers are general
- Algorithms keep changing

Superscale computers must balance

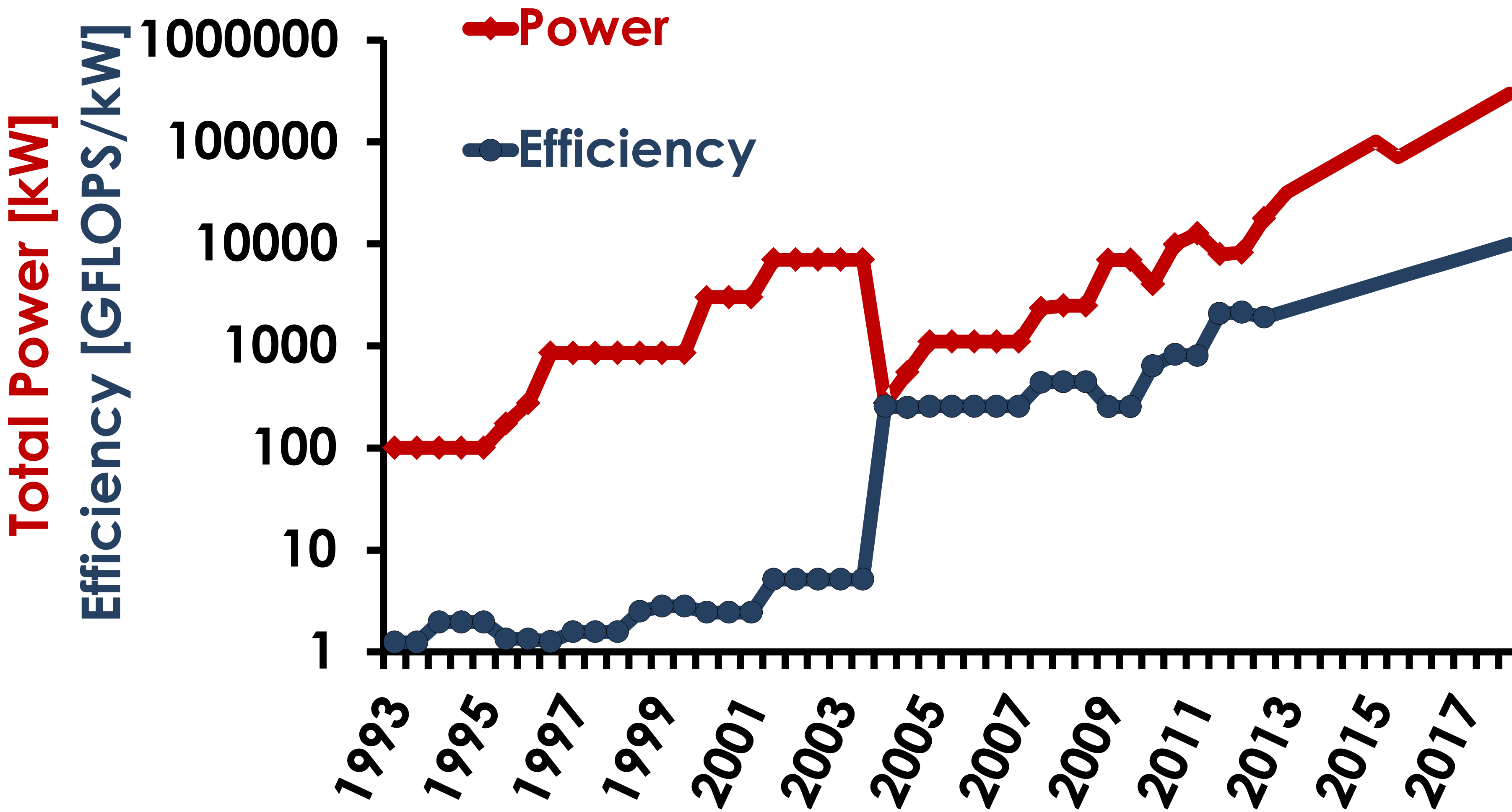
- Compute
- Storage
- Communication
- Constraints
  - OpEx and CapEx

Super is never super enough


Num. 1
Num. 10
Num.500
Idealized VLSI
Sustained FLOP/s
1E+18
1E+17
1E+16
1E+15
1E+14
1E+13
1E+12
1E+11
1E+10
1E+9
1E+8
1993
1995
1997
1999
2001
2003
2005
2007
2009
2011
2013
2015
2017

What's keeping us from **exascale?**

The **power** problem



Solution: build more **efficient systems**

Exascale computers are a lunacy:

- Crazy big (1M processors, >10MW)
- Need efficiency of embedded devices
- Effective for many domains
- Fully programmable

Why should you care?

Harbingers of things to come

Discovery awaits

Enable and promote **open innovation**

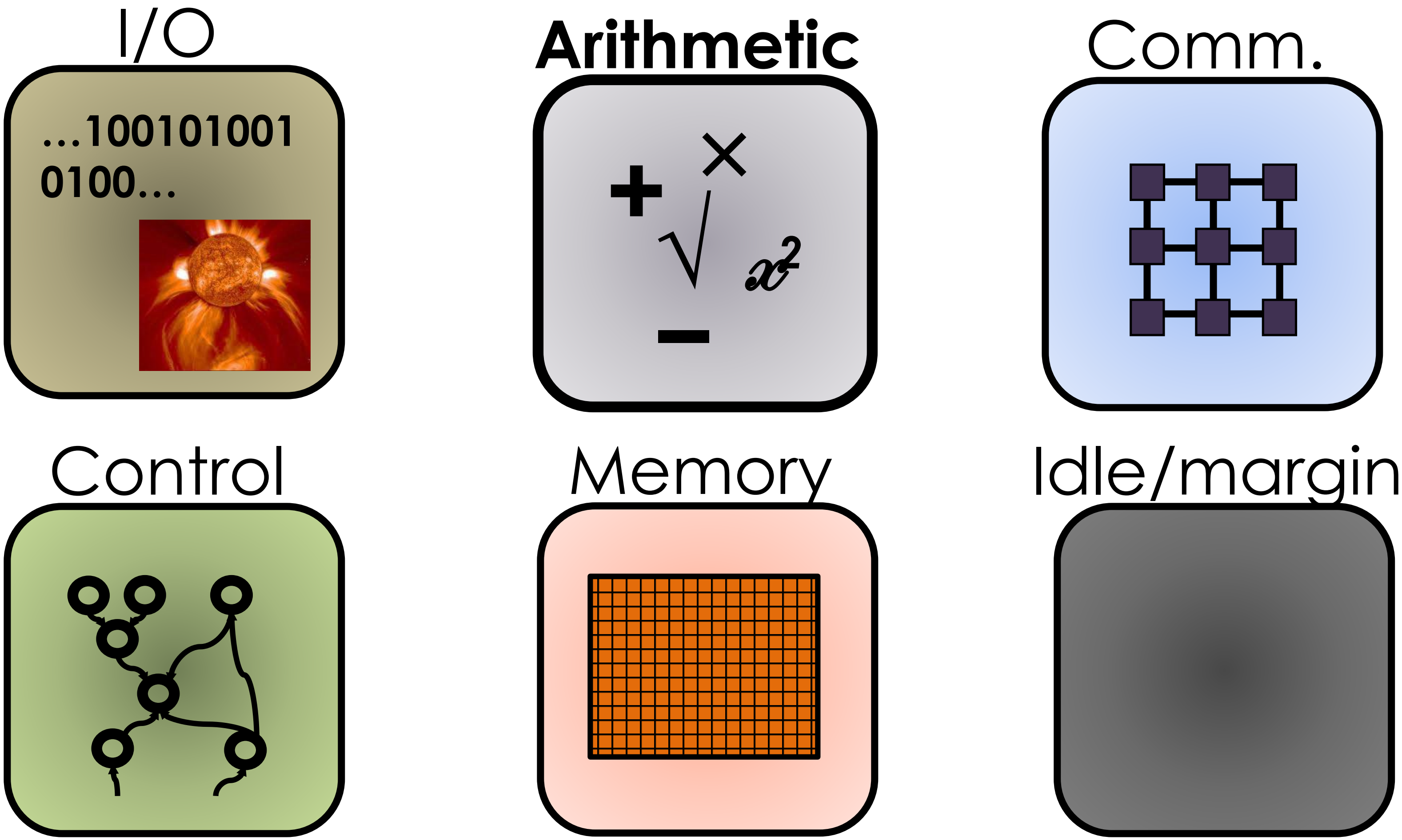
Where does the power go?

Cooling and infrastructure

- Amazing mechanical and electrical systems
- Getting close to optimal efficiency

17 PFLOP/s
18,688 nodes
>8 MW
~200 cabinets
~400 m^2 floorspace

Actual processing



How much of each component?

The budget: **Power ≤ 20MW**

20MW / 1 exa-FLOP/s

Energy ≤

20pJ/op

50 GFLOPs/W sustained

20MW / 1 exa-FLOP/s

Energy ≤

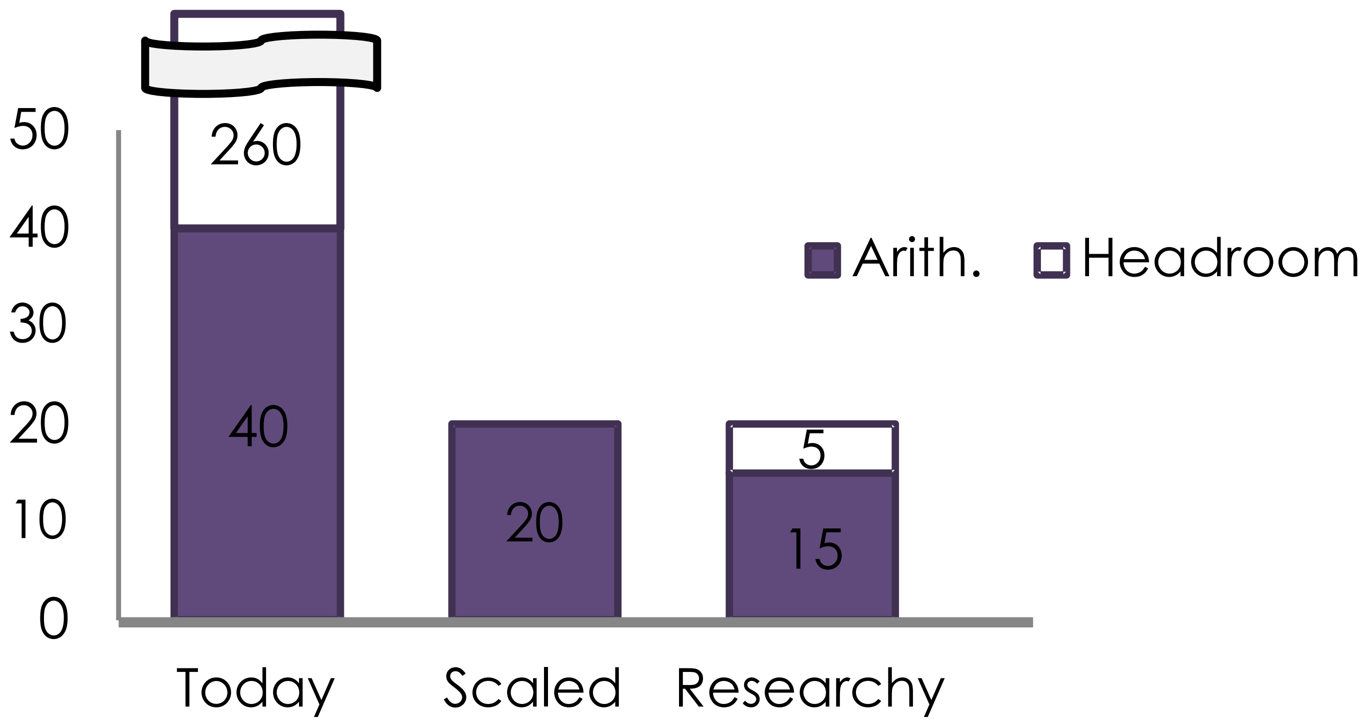
20pJ/op

50 GFLOPs/W sustained

Best supercomputer today: **~300pJ/op**

Arithmetic

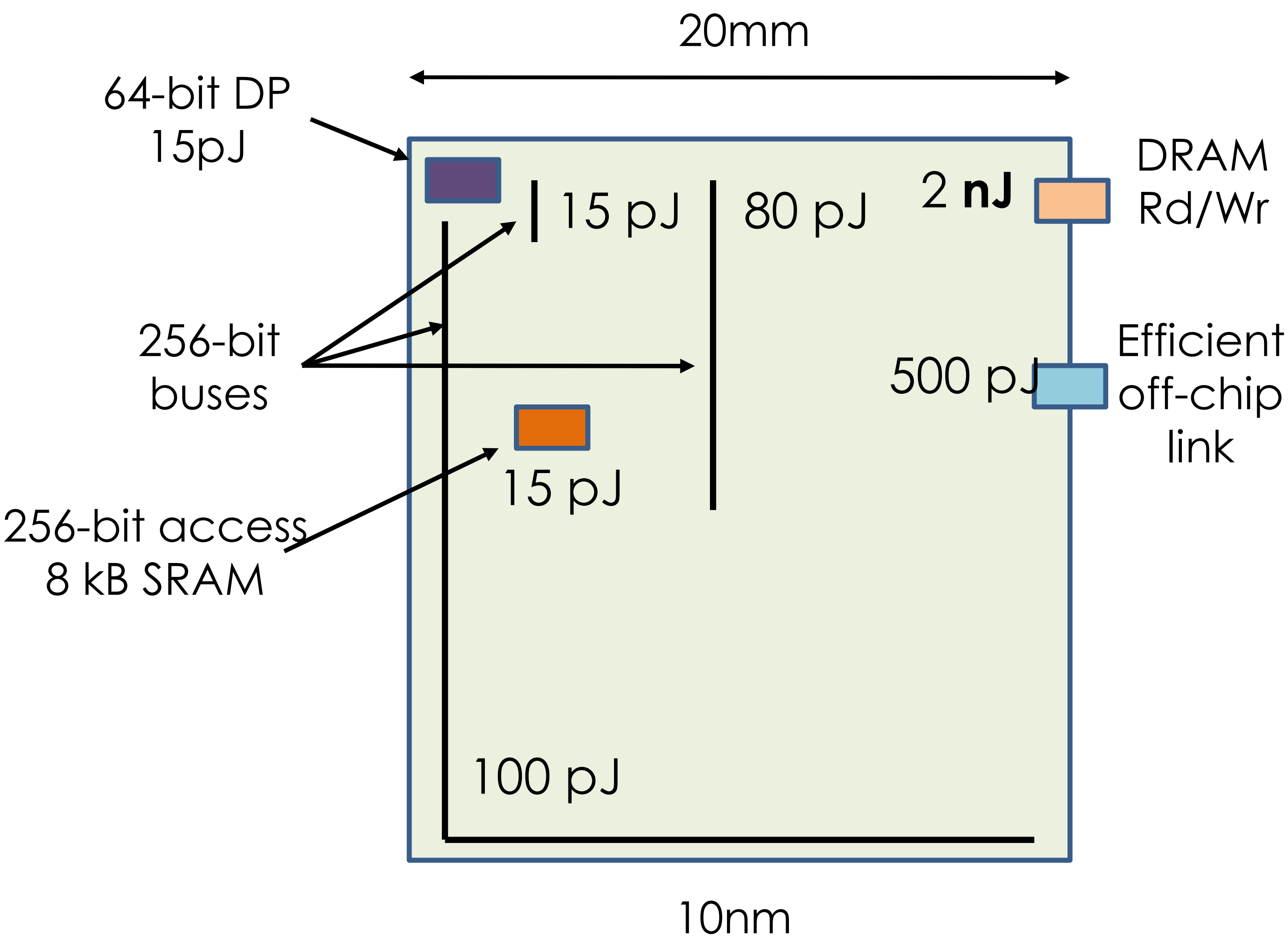
64-bit floating-point operation

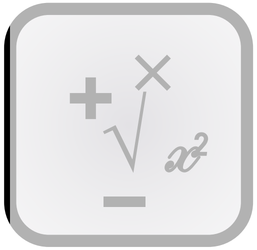


Rough estimated numbers

Enough headroom?

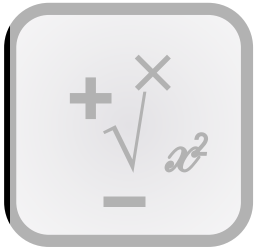
Unfortunately, hard tradeoffs





Need **more headroom**

– Minimize waste

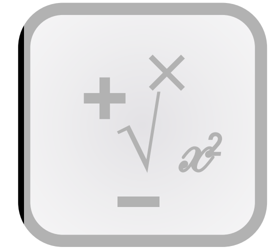


Do we care about single-unit performance?

Must all results be equally precise?

Must all results be correct?

Lunacy?

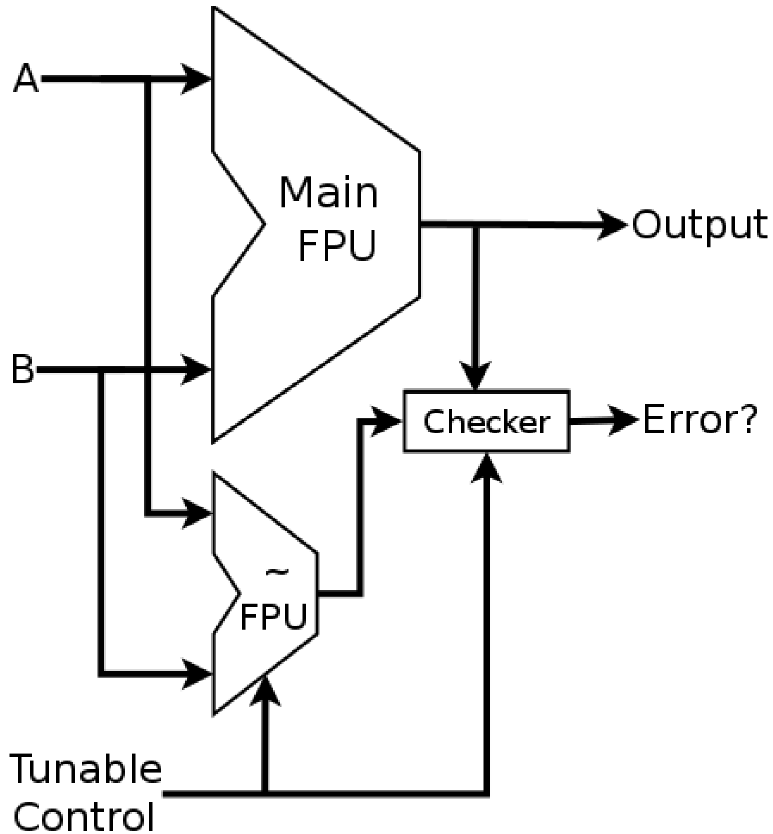


Relaxed reliability and precision

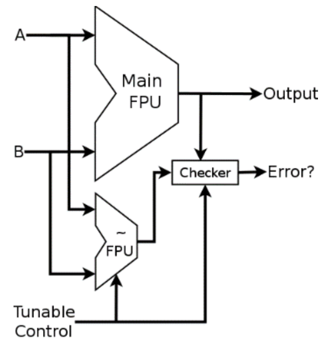
- Some lunacy
  (rare easy-to-detect errors + parallelism)
- **Lunatic fringe: bounded imprecision**
- Lunacy: live with real unpredictable errors

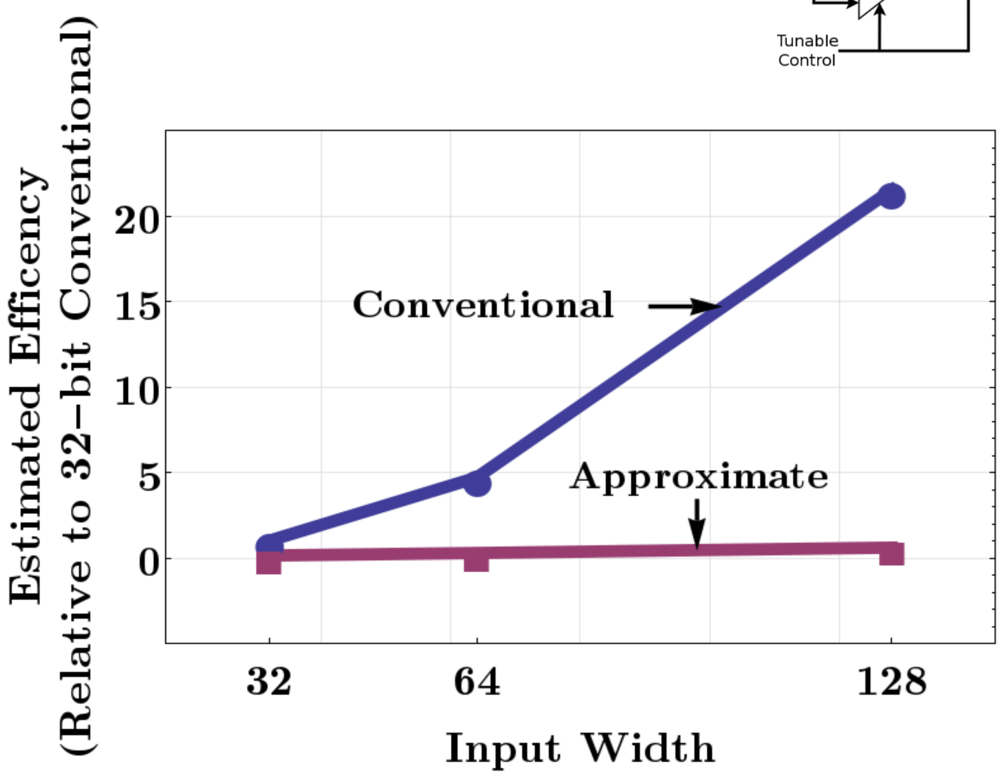
Rough estimated numbers for illustration purposes

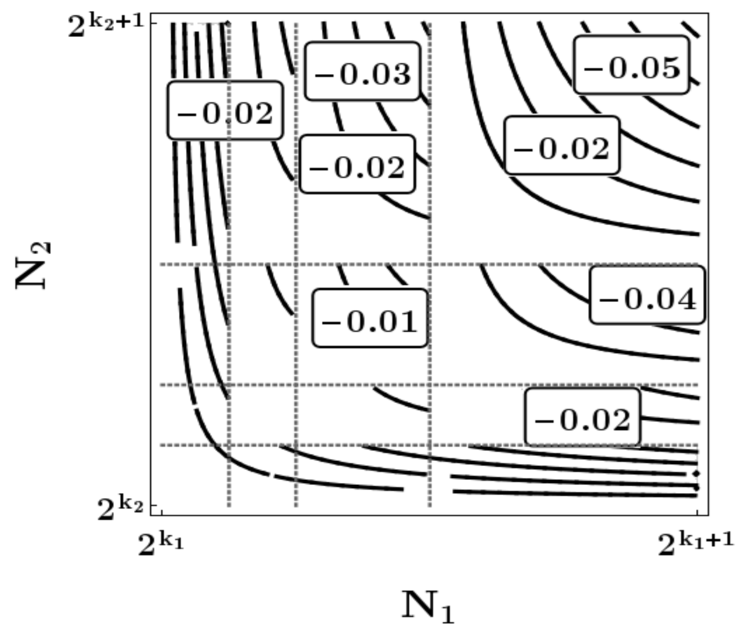
Bounded Approximate Duplication

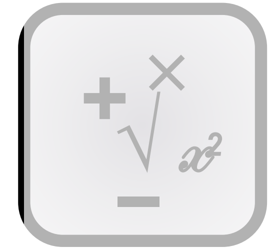


Bounded Approximate Duplication







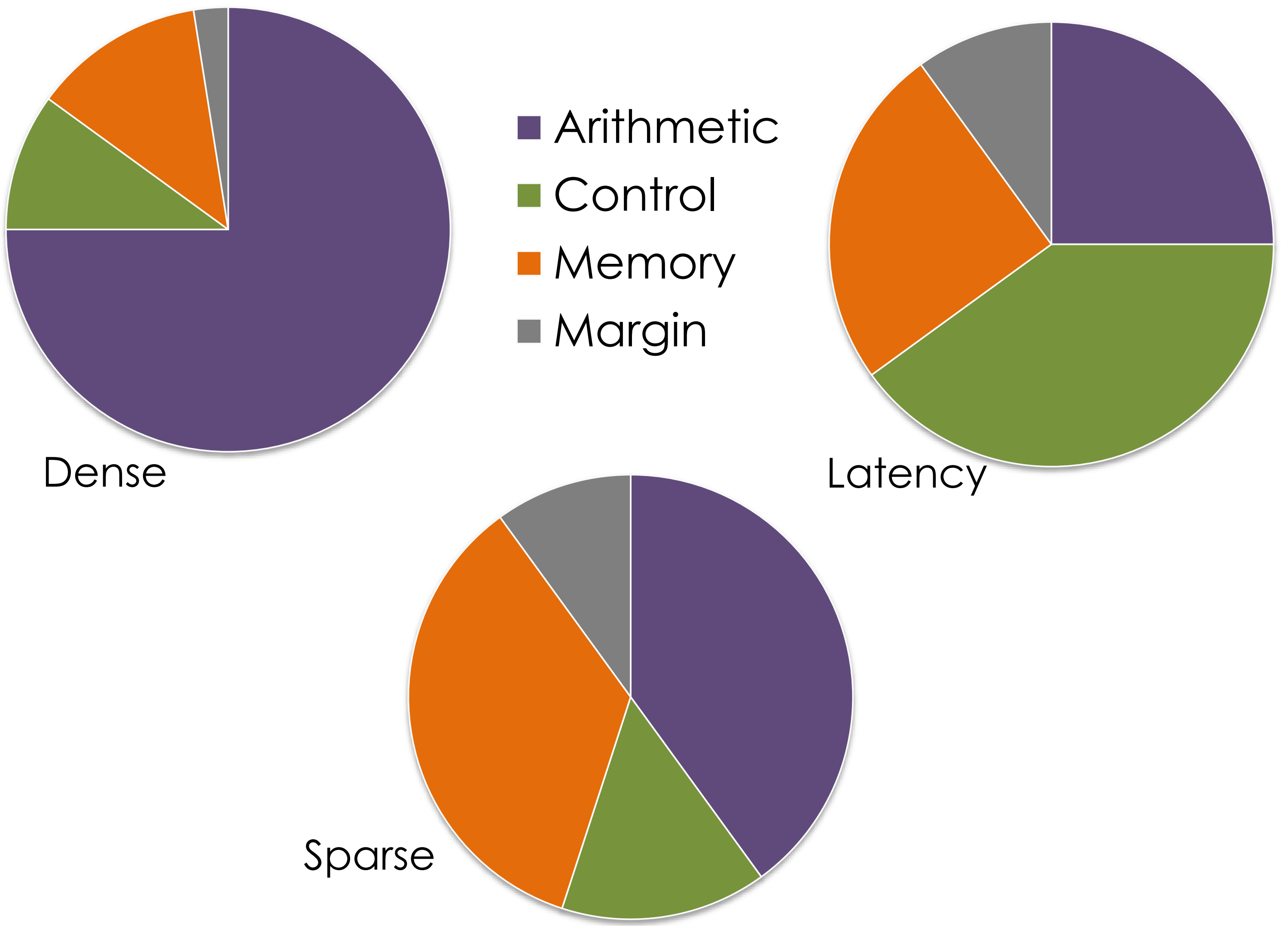


Supercomputers are general

→ Dynamic **adaptivity** and tuning

- Some programmers crazier than others
- Large effort in error-tolerant algorithms

Proportionality and overprovisioning



Architecture goals:

- Balance possible and practical
- Enable generality
- Don't hurt the common case

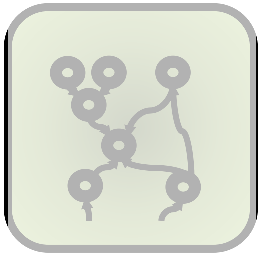
It's all about the algorithm

Architecture goals:

- Balance possible and practical
- Enable generality
- Don't hurt the common case

Architecture concepts so far:

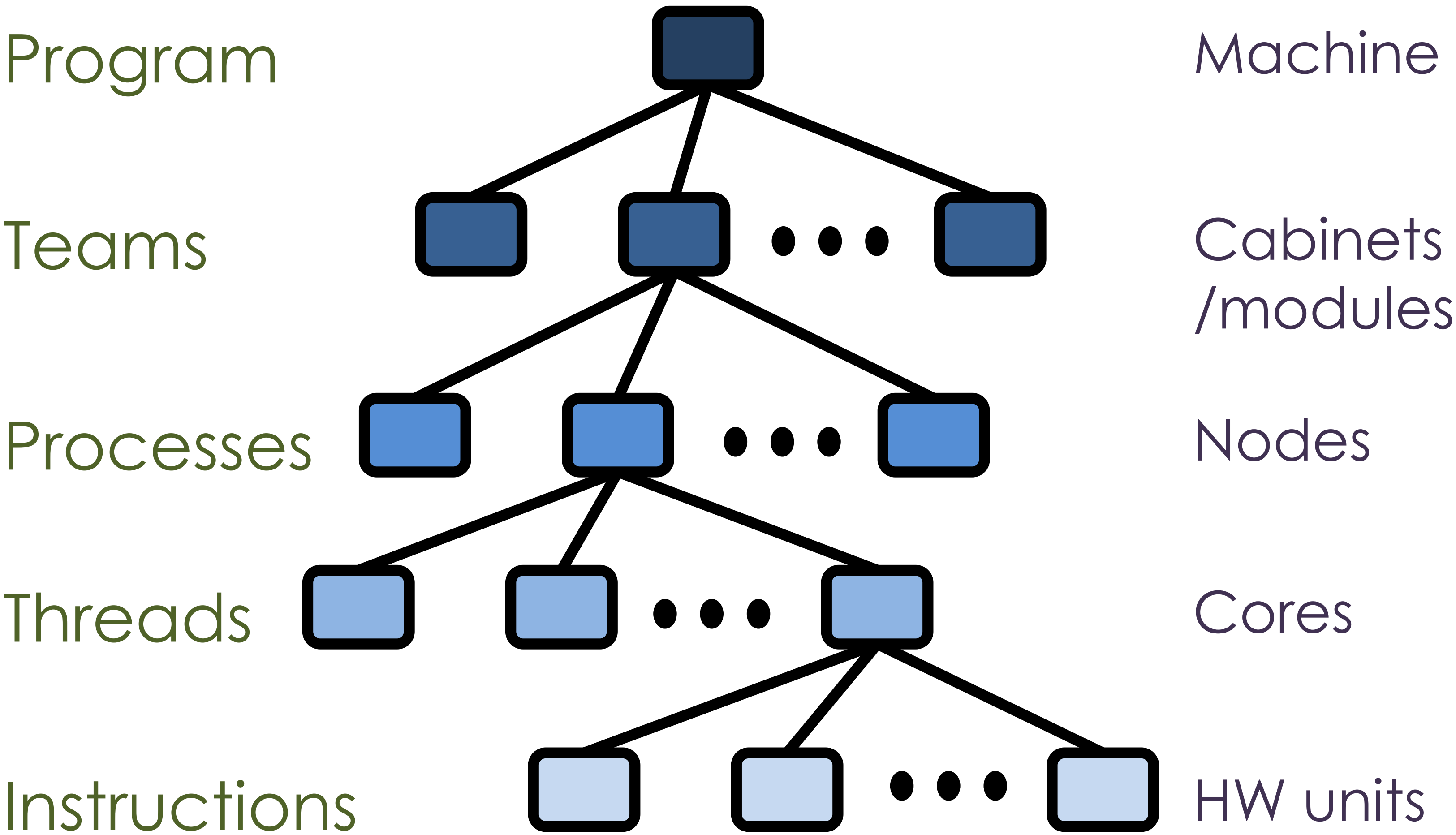
- Proportionality
  - Adaptivity
  - SW-HW co-tuning
- Locality
- Parallelism



How to control a **billion arithmetic units**?

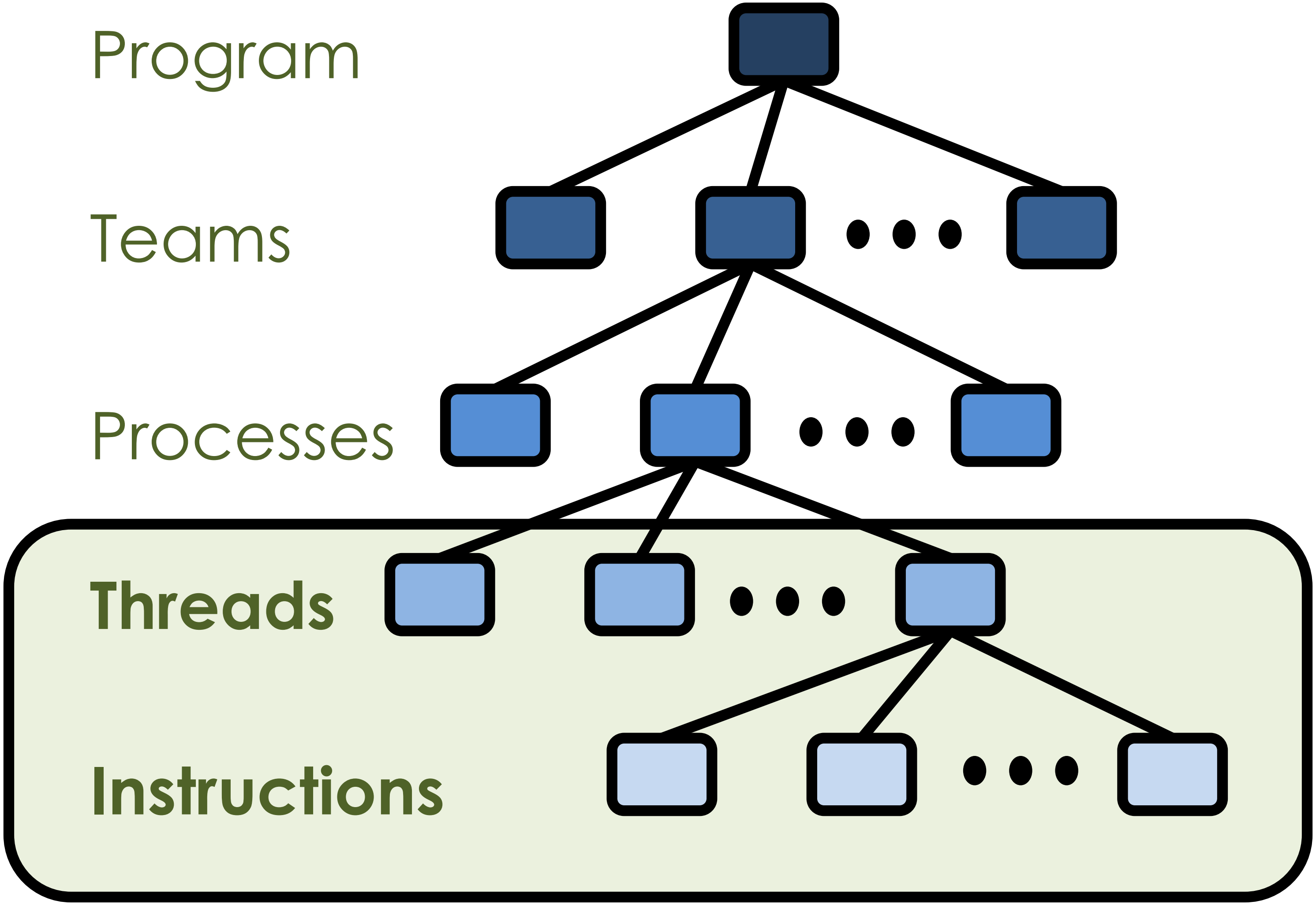
Hierarchy minimizes control cost

– Amortize control decisions

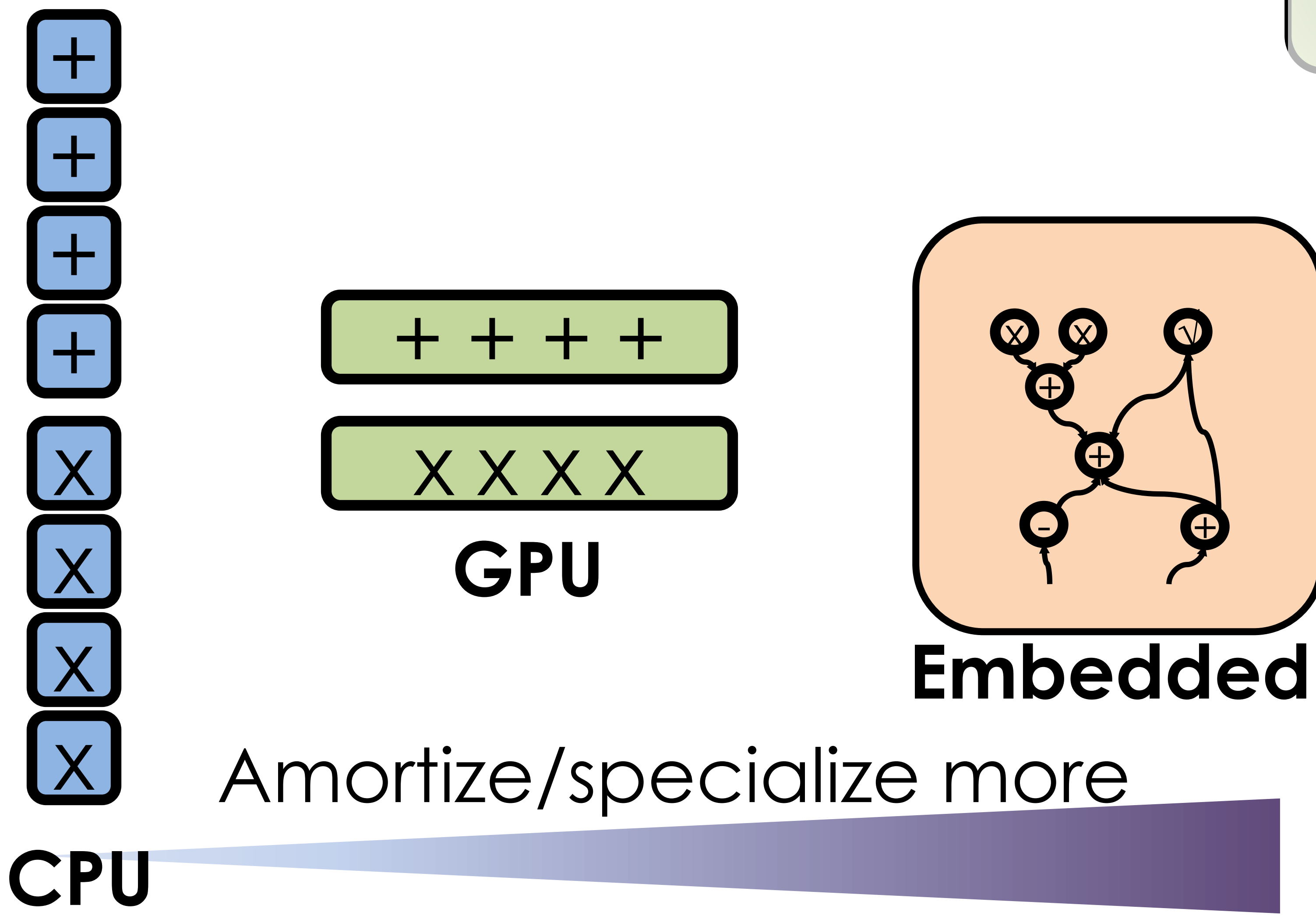


Hierarchy minimizes control cost

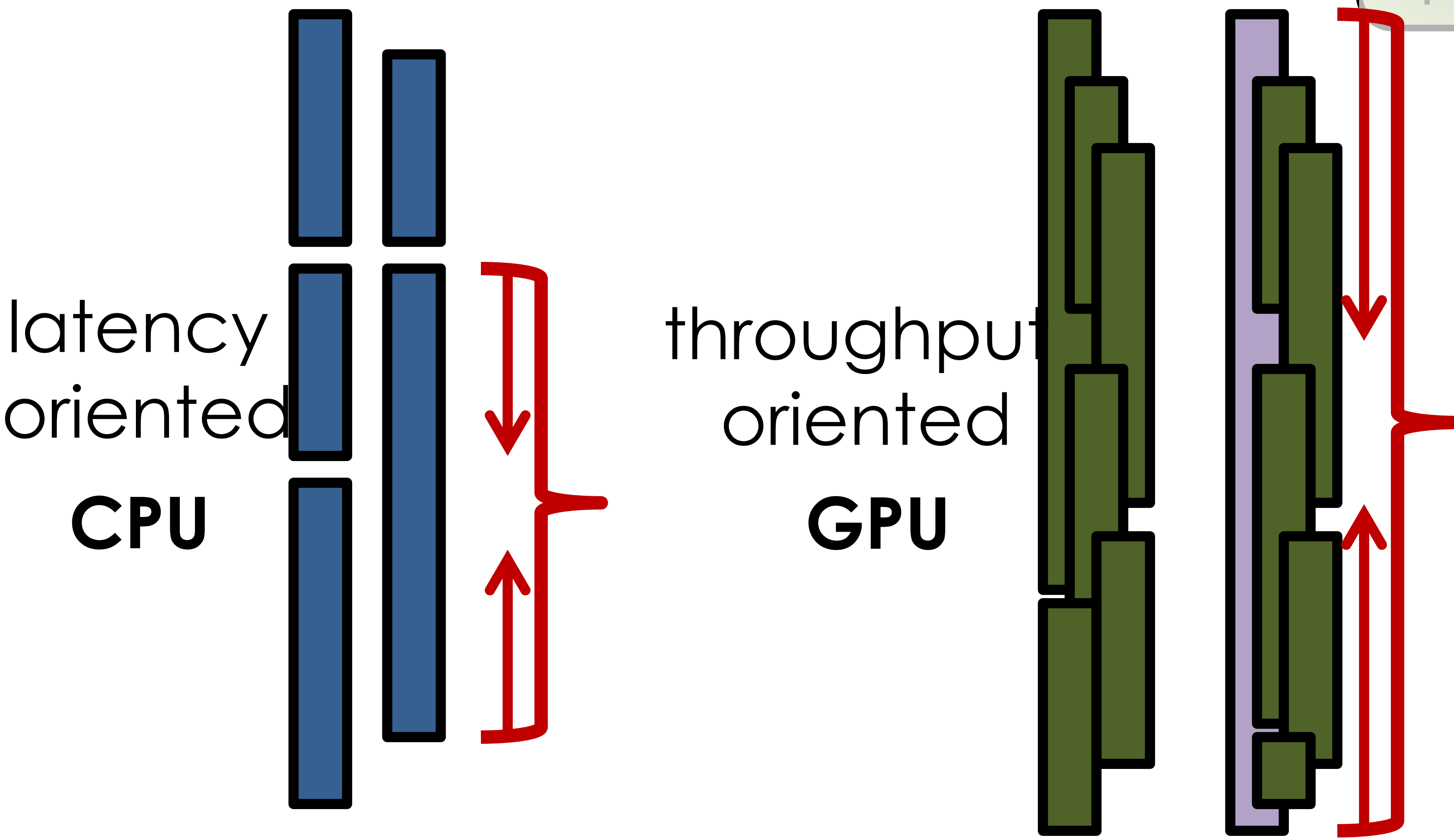
– Amortize control decisions

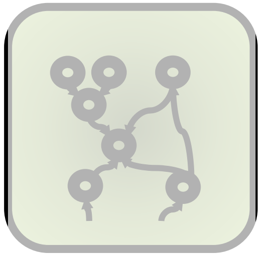


What does a **HW instruction** do?



Single-thread or bulk-thread



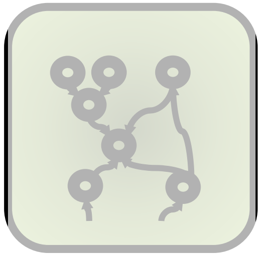


Heterogeneity is a necessity
→ **disciplined specialization**

Must balance generality and control cost

Over-specialization **stifles innovation**

– **Algorithms more important** than hardware



Heterogeneity is **lunacy**

- Programing and tuning extremely tricky
- Diverse and rapidly evolving algorithms

Disciplined heterogeneity

Scarcity of choice

- **Throughput** oriented
- **Latency** oriented
- **Disciplined reconfigurable** accelerators

Heterogeneous computing already here

- “GPU” for throughput
- CPU for latency
- Abstracted FPGA accelerators

Heterogeneous computing already here

Titan node (Cray XK7)

Copyrighted picture of an XK7 node

Copyrighted picture of Kepler die

Copyrighted picture of Opteron die

NVIDIA GPU + AMD CPU

Stampede node

Copyrighted picture of Xeon Phi card

Copyrighted picture of Xeon Phi die

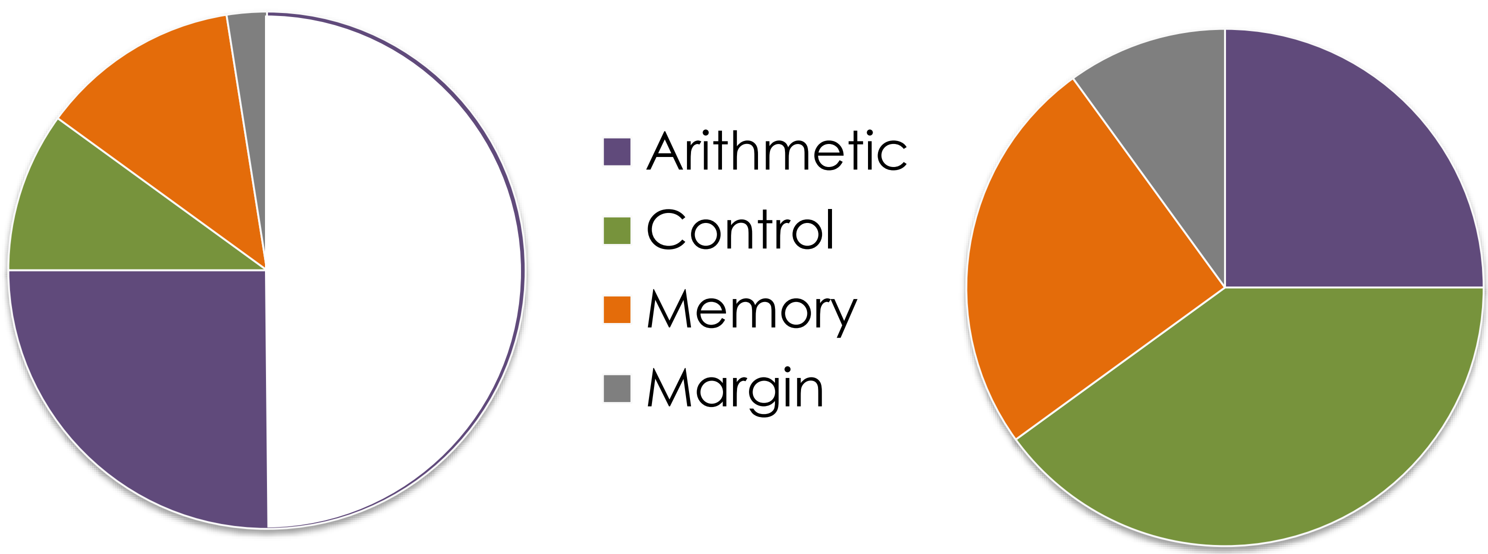
Copyrighted picture of Stampede node

Copyrighted picture of Sany Bridge die

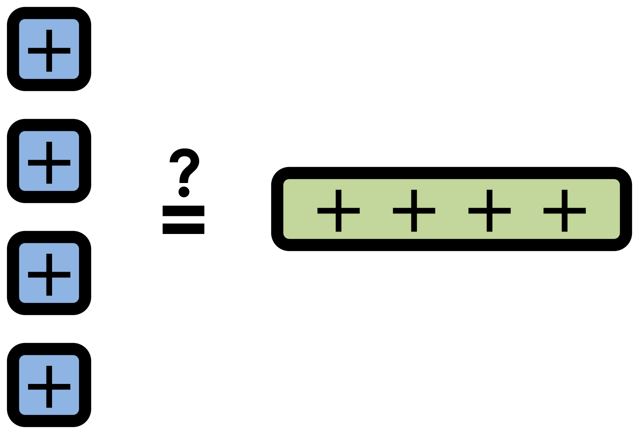
Intel Xeon CPU + Xeon Phi

Choose most **efficient** core

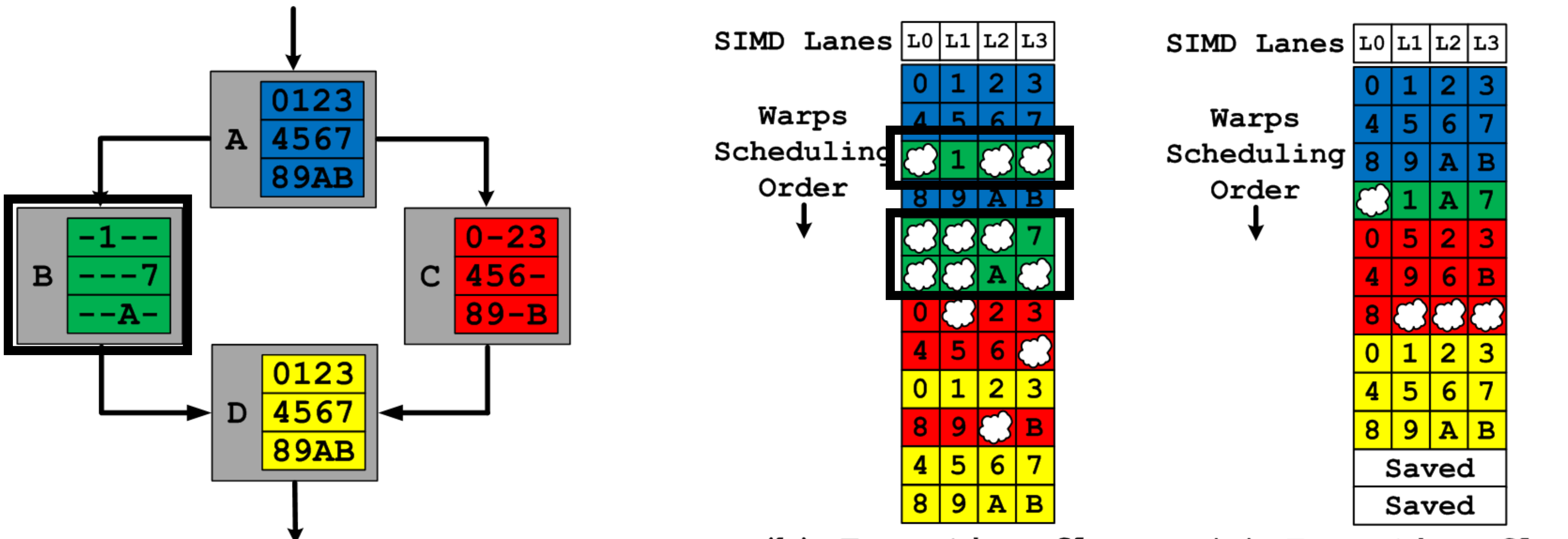
Choose most **efficient** core



Generalize the throughput core



Generalize the throughput core

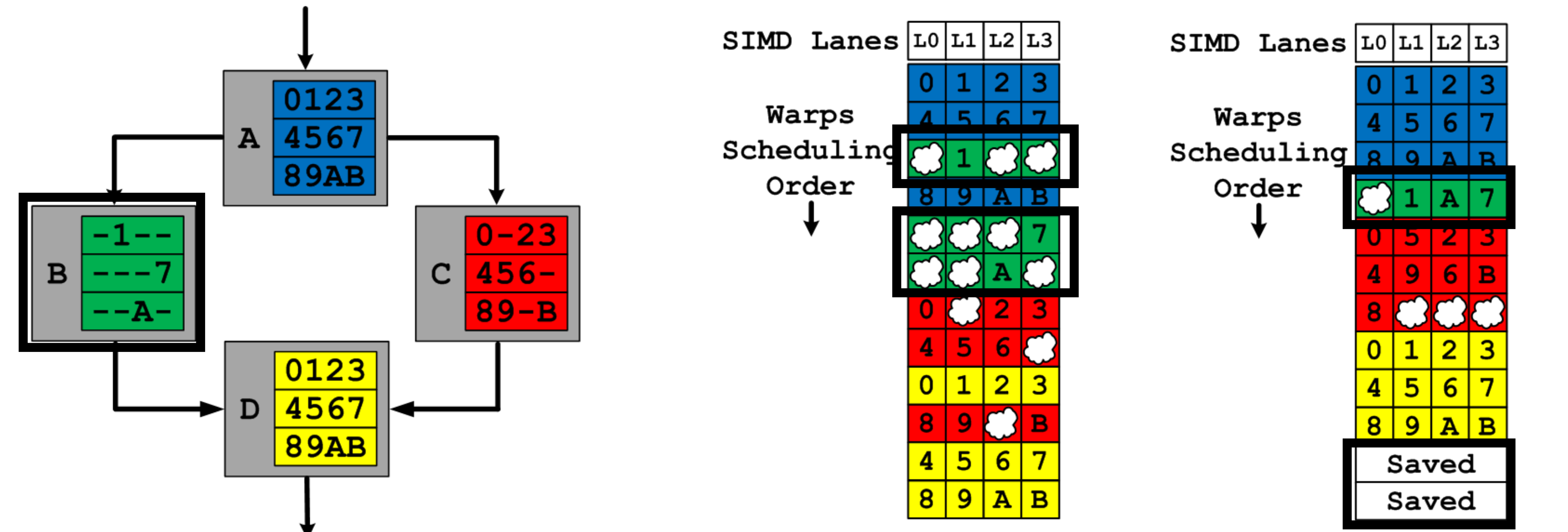


(a) Example control flow graph

(b) Execution flow without compaction

(c) Execution flow with compaction

Generalize the throughput core



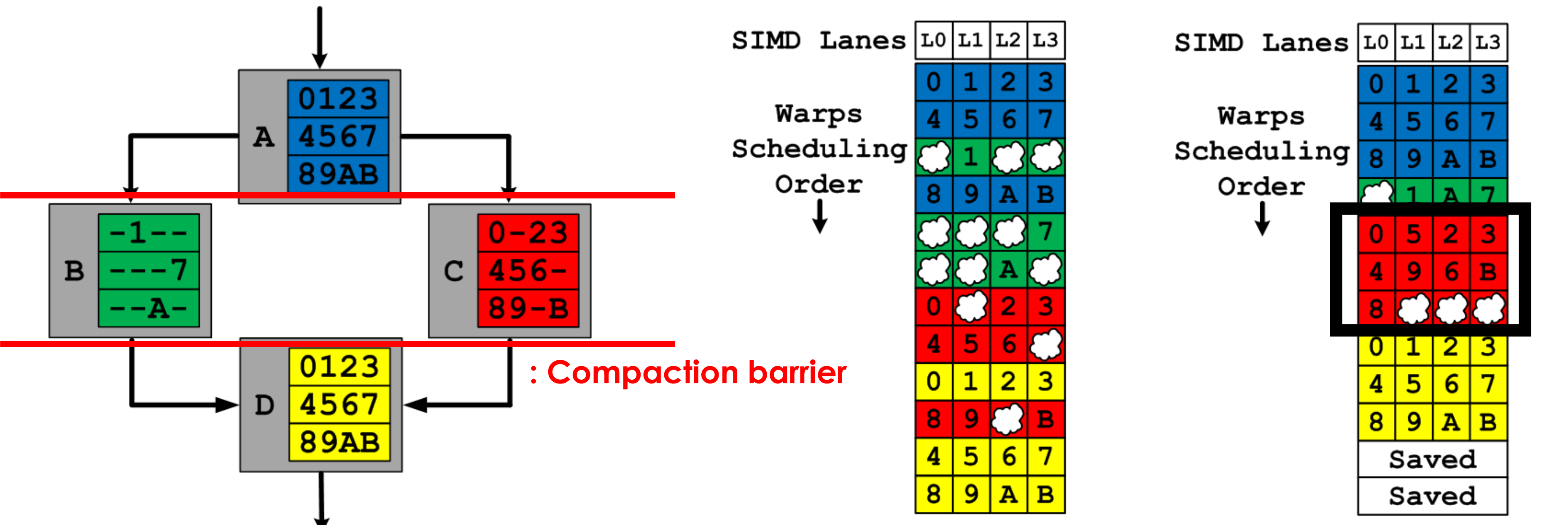
(a) Example control flow graph

(b) Execution flow without compaction

(c) Execution flow with compaction

Synchronization may impair execution

- Predict when necessary → robust optimization
- **Adaptive compaction**



(a) Example control flow graph

(b) Execution flow without compaction

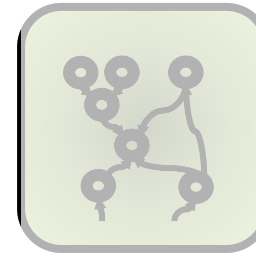
(c) Execution flow with compaction

Architecture goals:

- Balance possible and practical
- Enable generality
- Don't hurt the common case

Architecture so far:

- Proportionality
  - Adaptivity
  - Heterogeneity
  - SW-HW co-tuning
- Locality
- Parallelism
- Hierarchy



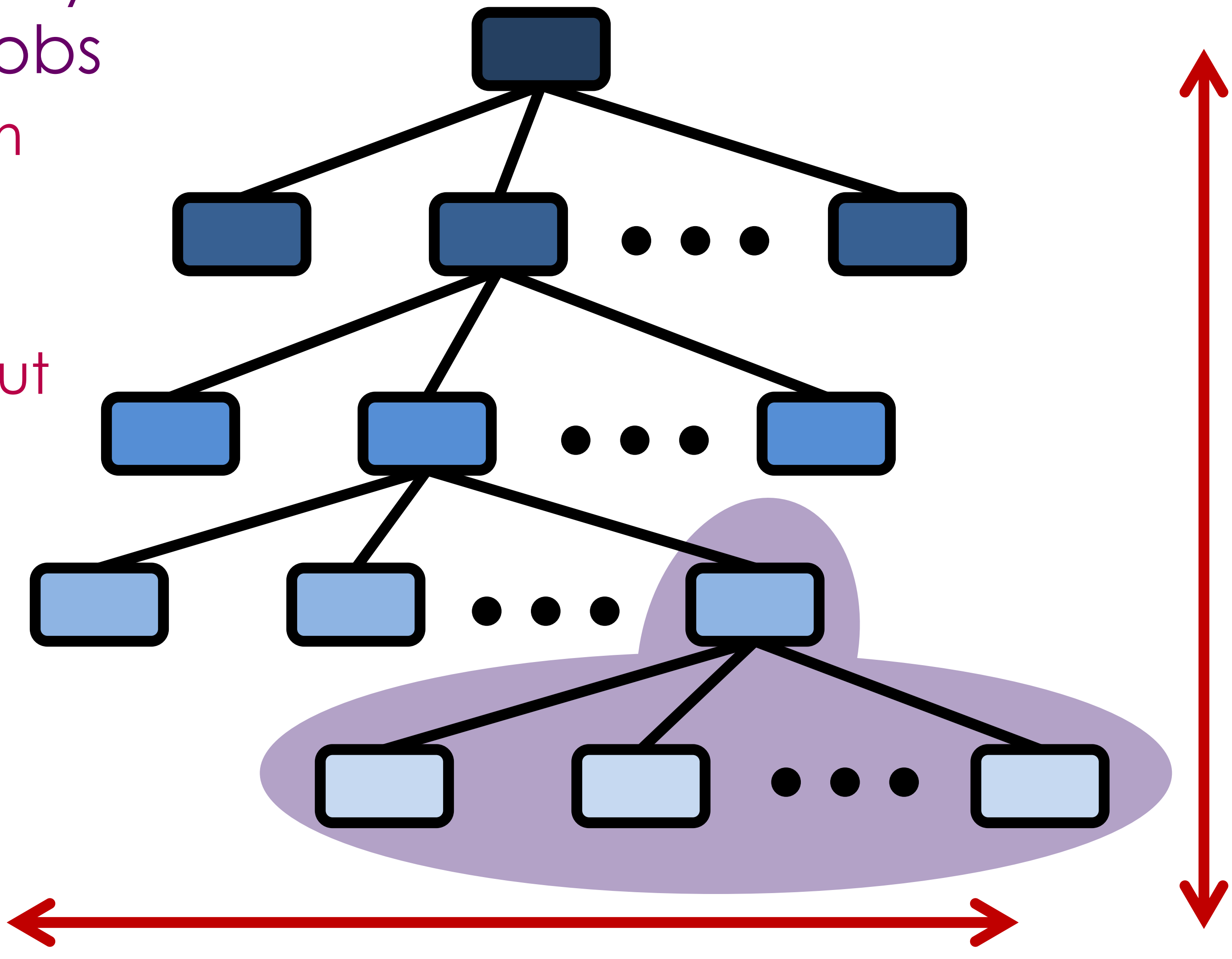
Heterogeneity is **lunacy**

- Programing and tuning extremely tricky
- Diverse and rapidly evolving algorithms

How can we **program** these machines

Hierarchical languages restore **some sanity**

- **Abstract** key tuning knobs
  - Parallelism
  - Locality
  - Hierarchy
  - Throughput



Hierarchical languages restore some sanity

- CUDA and OpenCL
- Sequoia (Stanford)
- Habanero (Rice)
- Phalanx (NVIDIA)
- Legion (Stanford)
- Working into X10, Chapel
- ...

Domain specific languages even better

A counter example:
DE Shaw Research **Anton**

Copyrighted
picture of
Anton package
(Google
image search)

Copyrighted
figure demonstrating
molecular dynamics
(Google
image search)

Copyrighted
figure of
Anton system
(Google
image search)

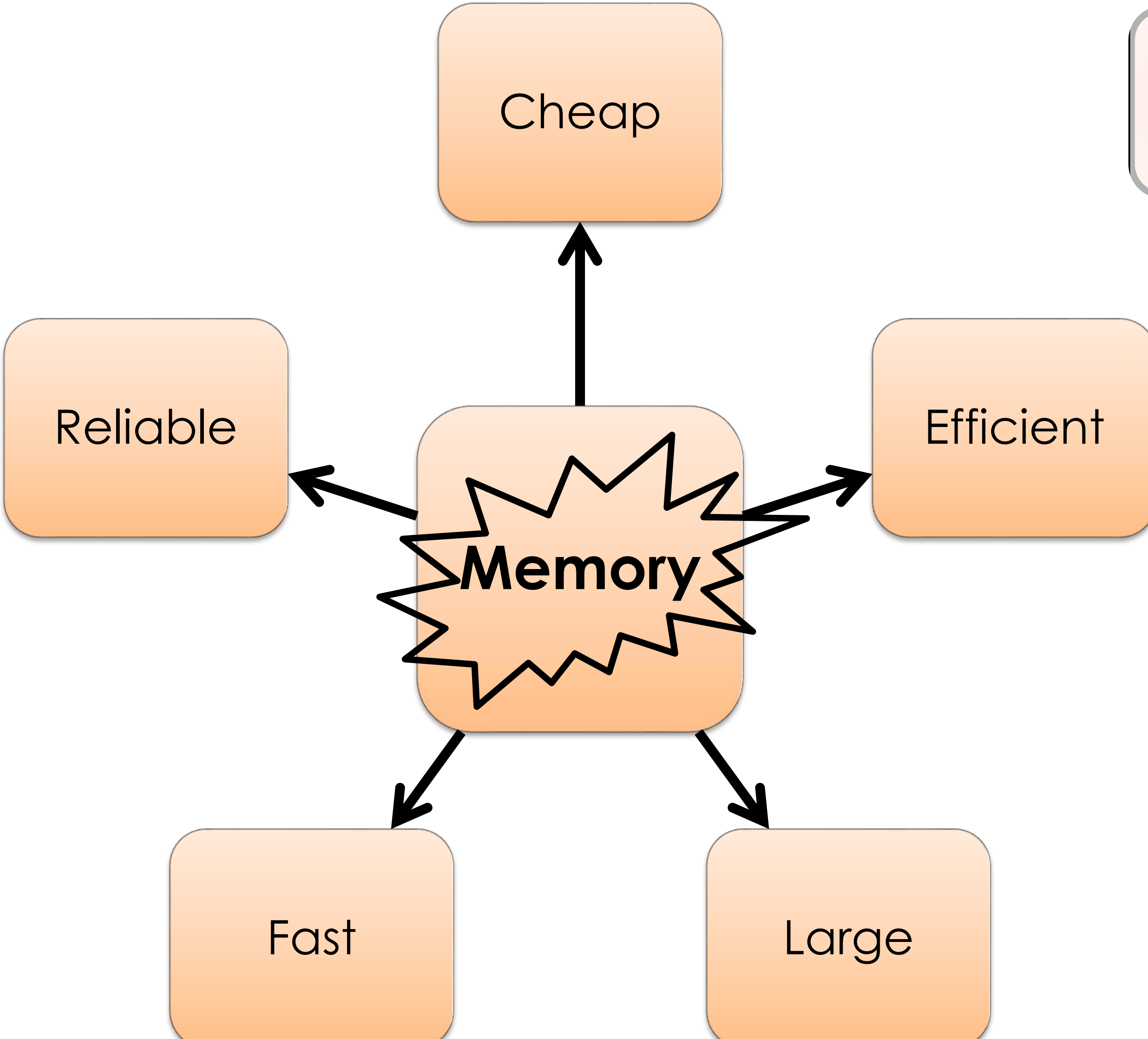
Another counter example?

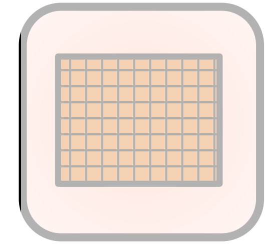
Microsoft Catapult

- Disciplined reconfigurability for the cloud
- Distributed FPGA accelerator
  - Within form-factor and cost constraints
- Architected interfaces and compiler
  - Compiler for software, not (just) hardeare

Copyrighted figure of datacenter (Google image search)

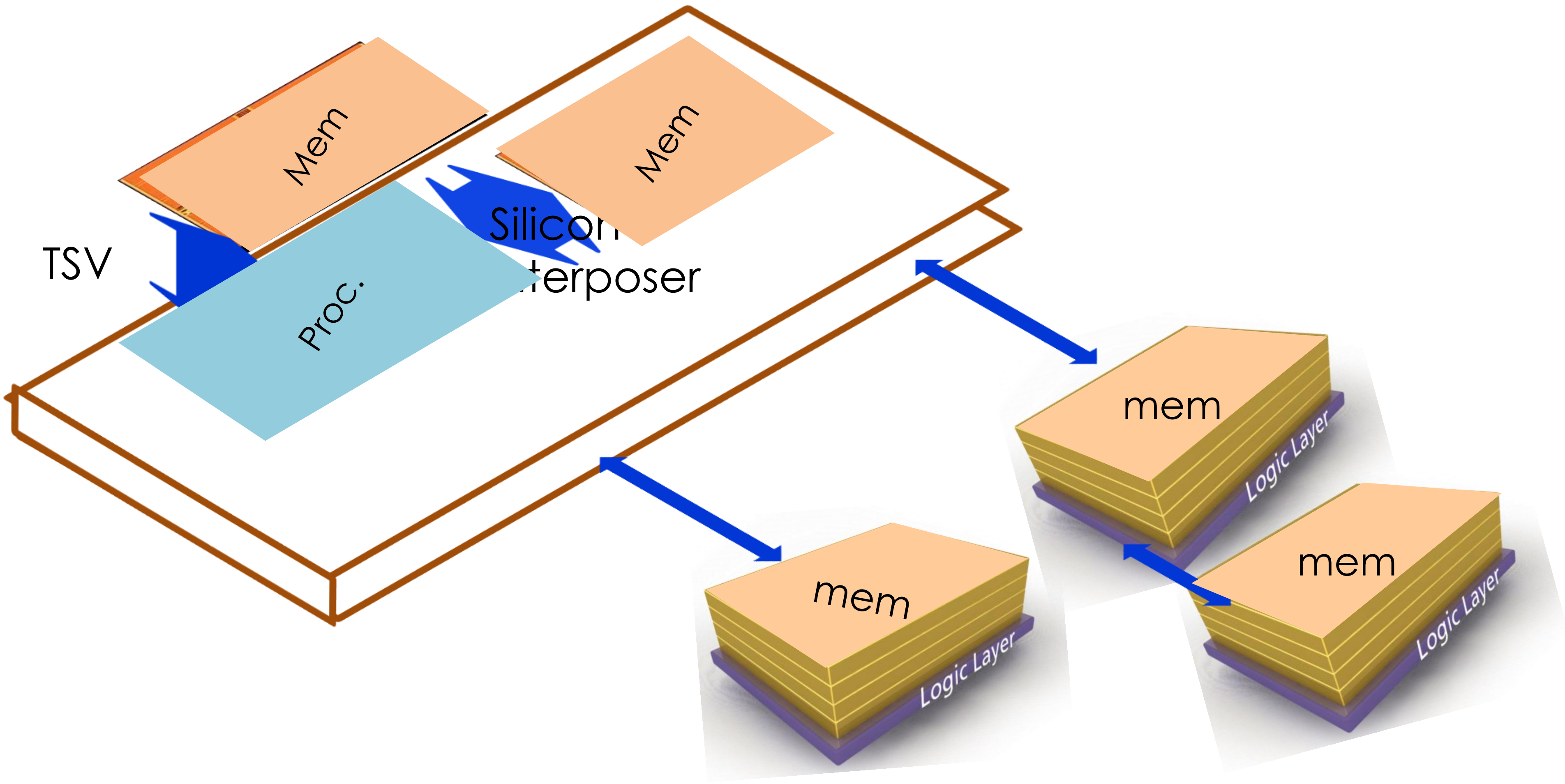
Copyrighted figure of Catapult diagram (from ISCA 2014 paper)

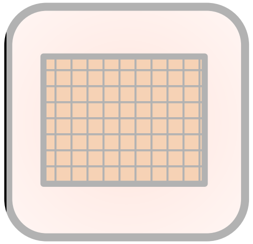

Cheap
Reliable
Efficient
Memory
Fast
Large



Fast + large → **hierarchy**

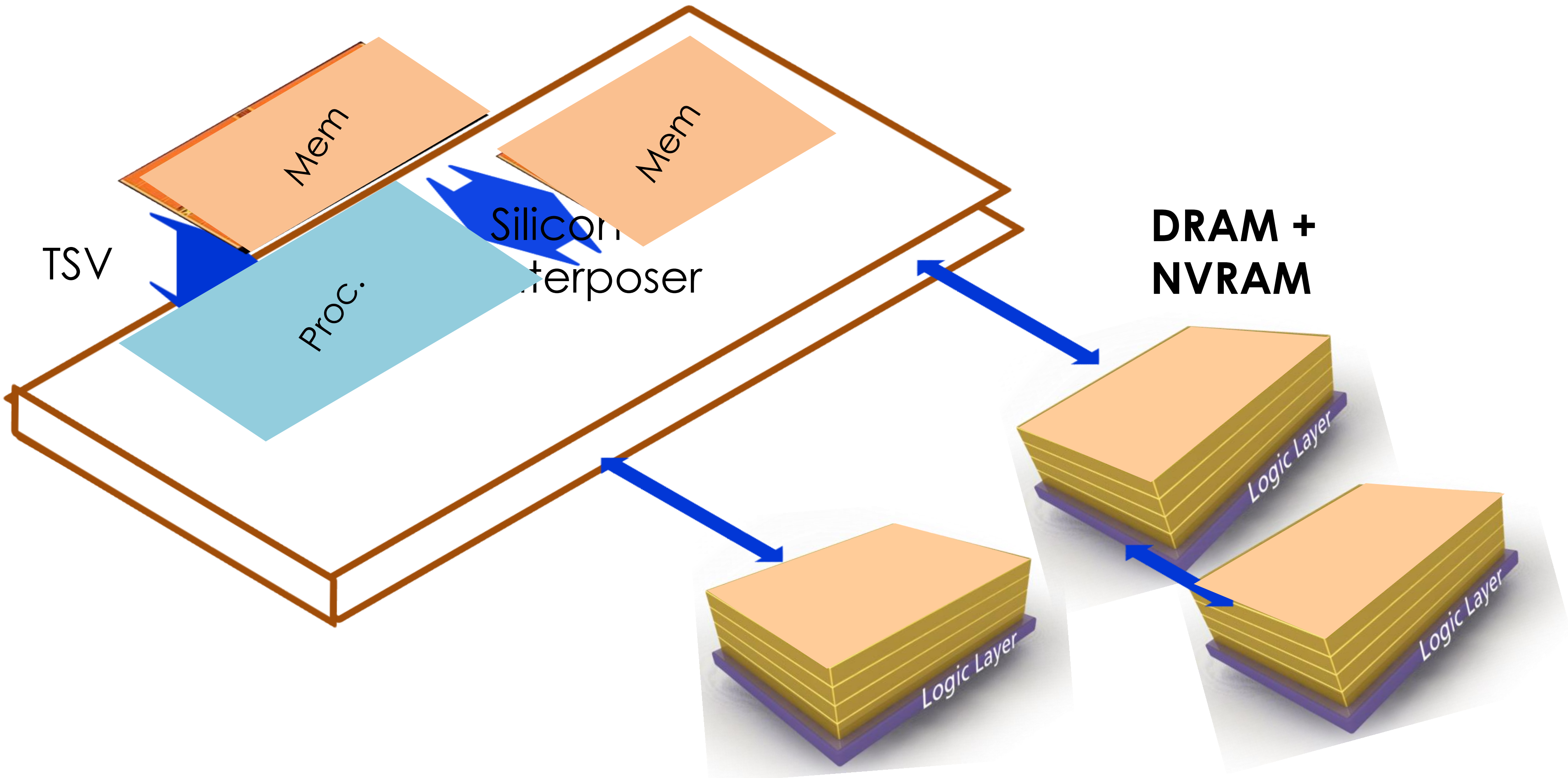
Fast + large → **hierarchical**

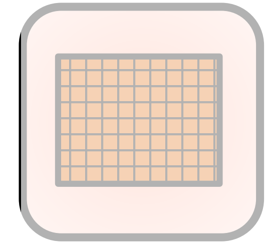




Large + efficient →
heterogeneity

Large + efficient → **heterogeneous**

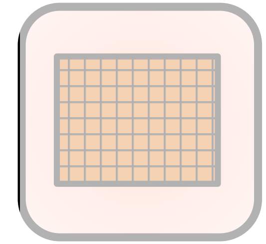




Heterogeneous + hierarchical
→ **big mess**

Research just starting

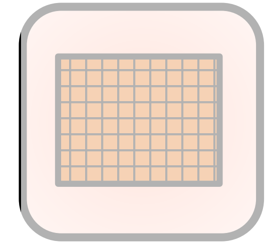
- New mechanisms needed
- Very interesting tradeoffs with reliability



Fast + efficient

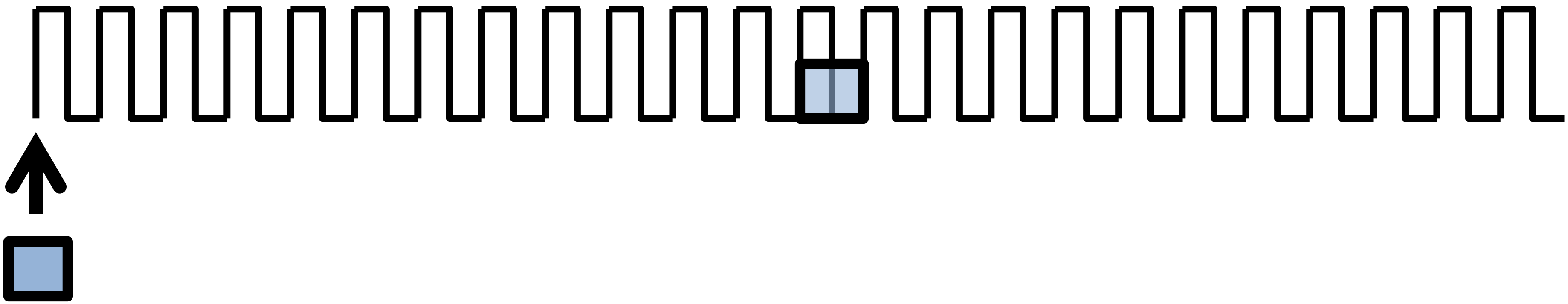
→ control hierarchy

+ parallelism

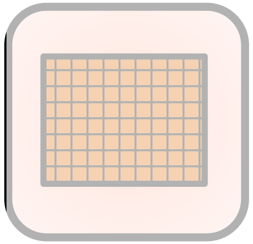


Fast + efficient

→ hierarchy + parallelism

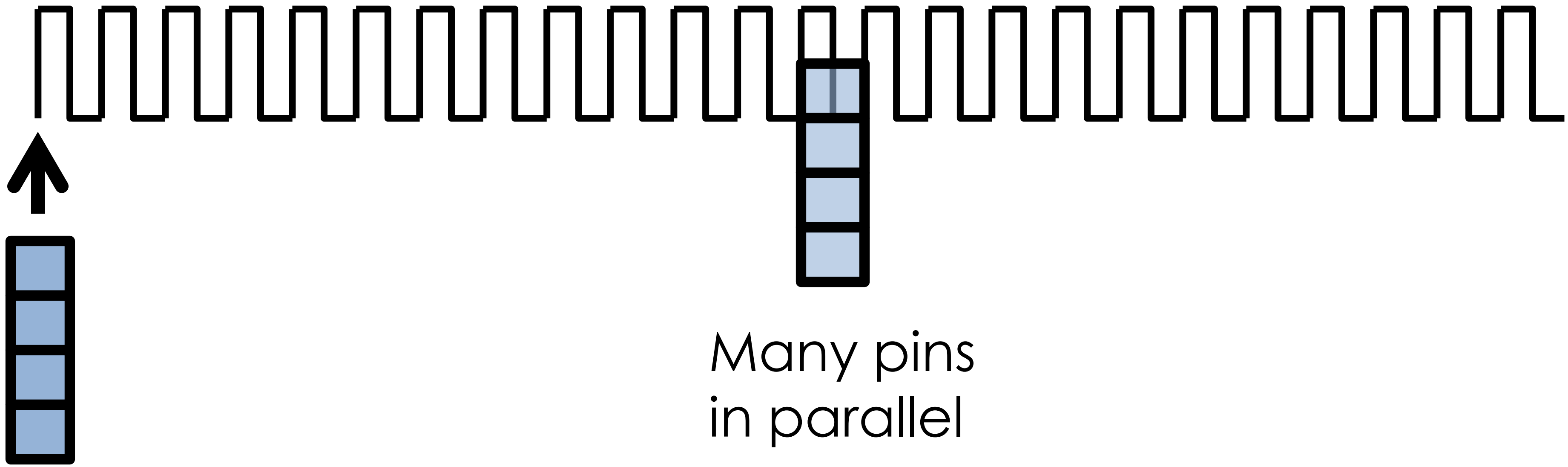


Access cell



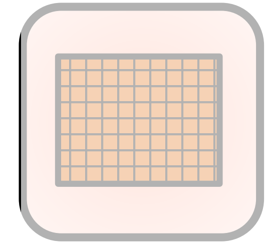
Fast + efficient

→ **hierarchy + parallelism**



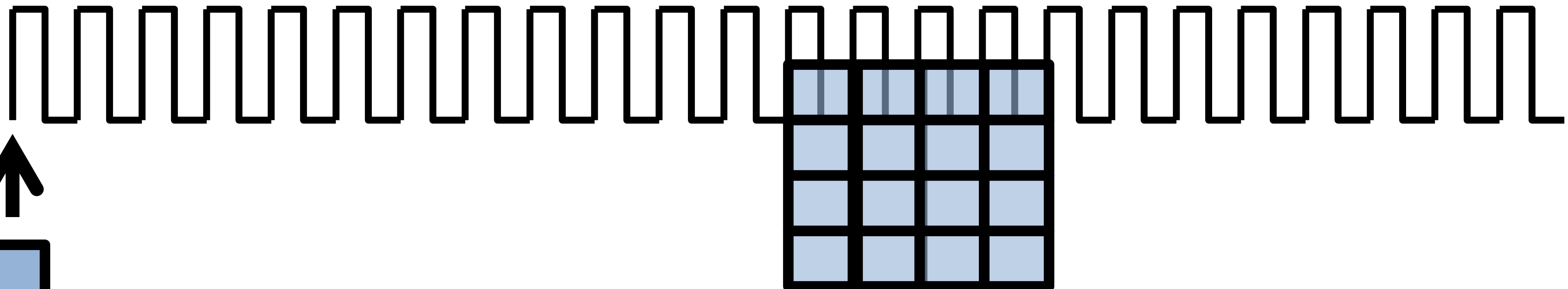
Many pins
in parallel

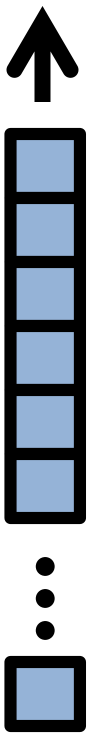
Access many
cells in parallel



Fast + efficient

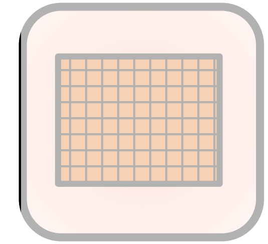
→ **hierarchy + parallelism**



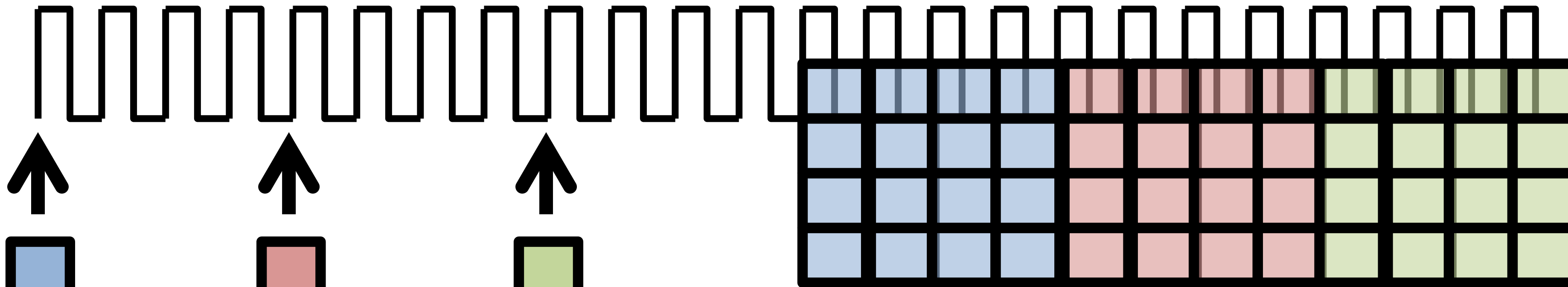


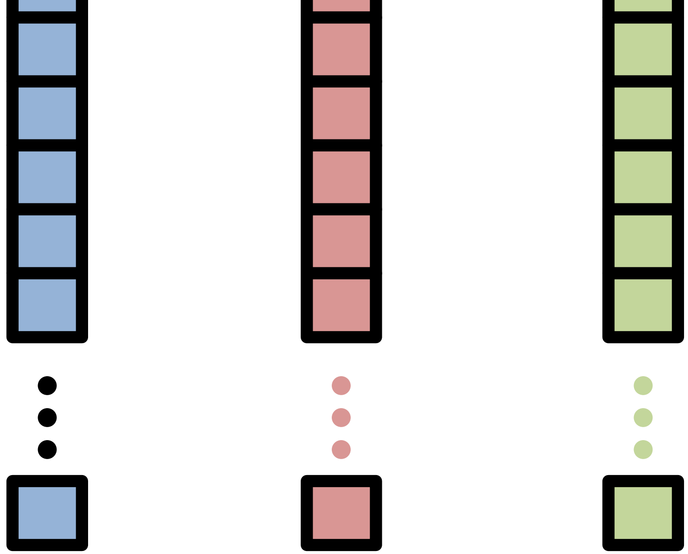
Many pins
in parallel over
multiple cycles

Access even more
cells in parallel



Fast + efficient
→ hierarchy + parallelism

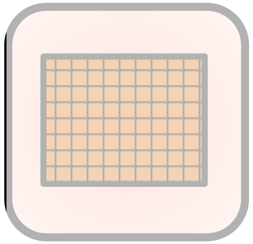


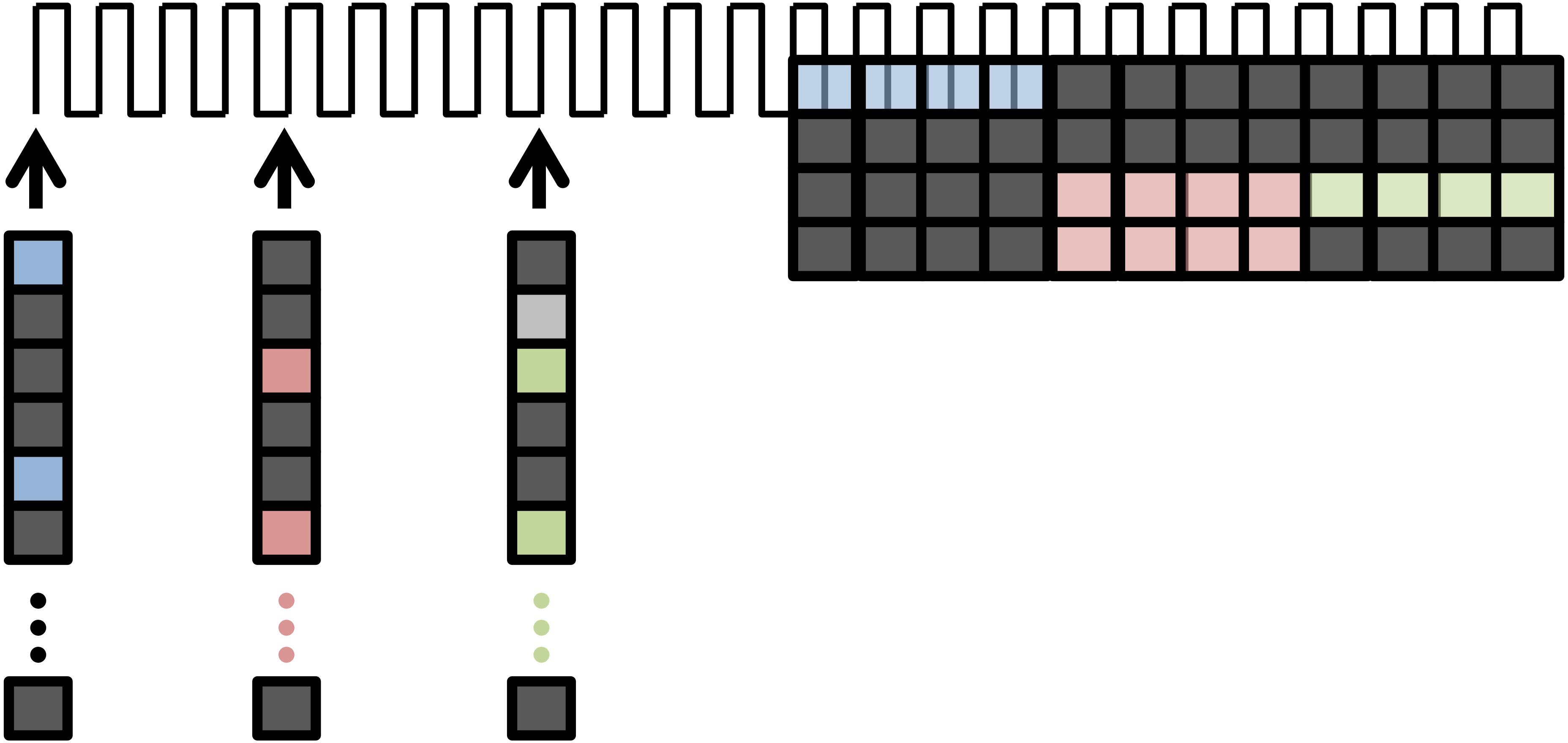


Many pins
in parallel over
multiple cycles

Access even more
cells in parallel

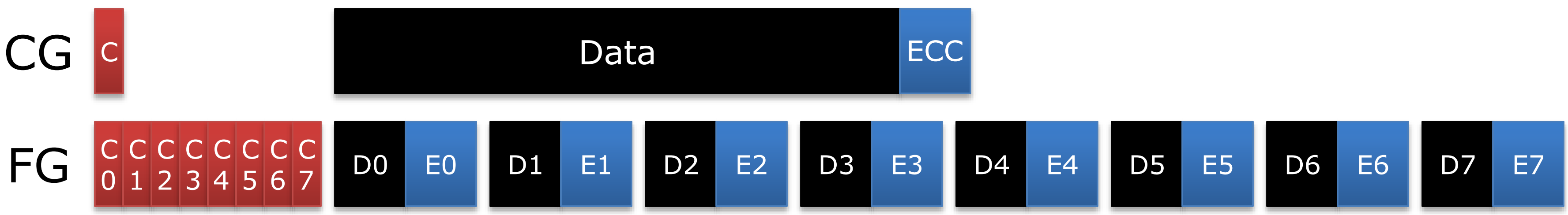
Hierarchy + parallelism → **potential waste**





Is fine-grained access better?

– Wasteful when all data is needed



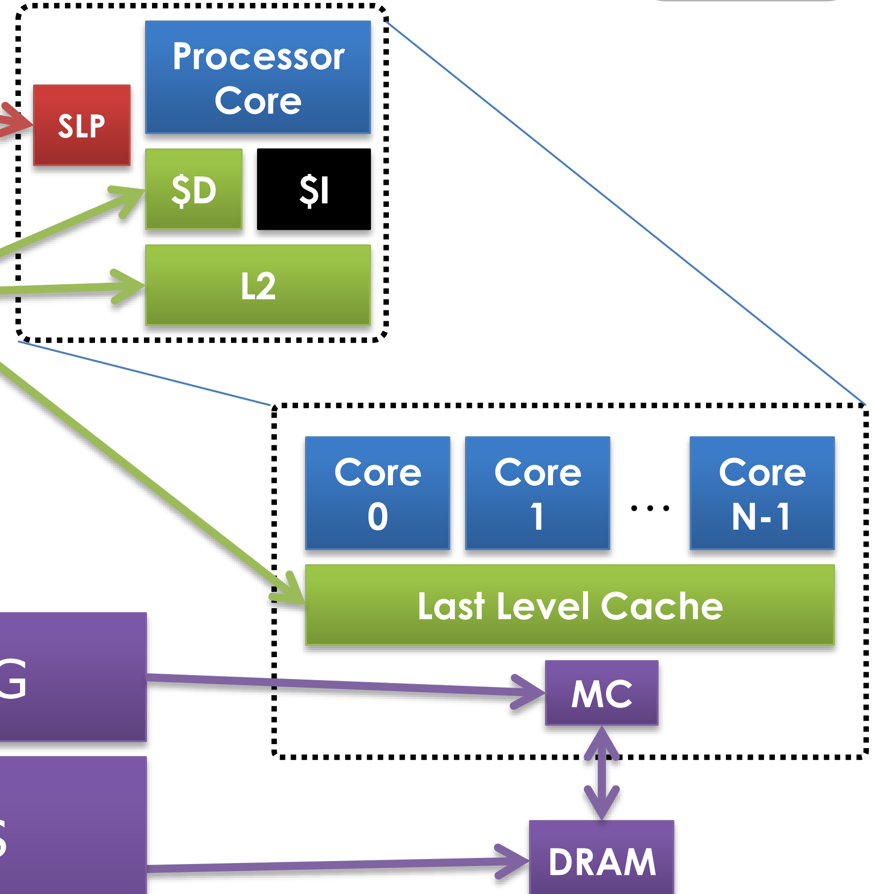
Dynamically **adapt** granularity → best of both worlds

Locality Predictor

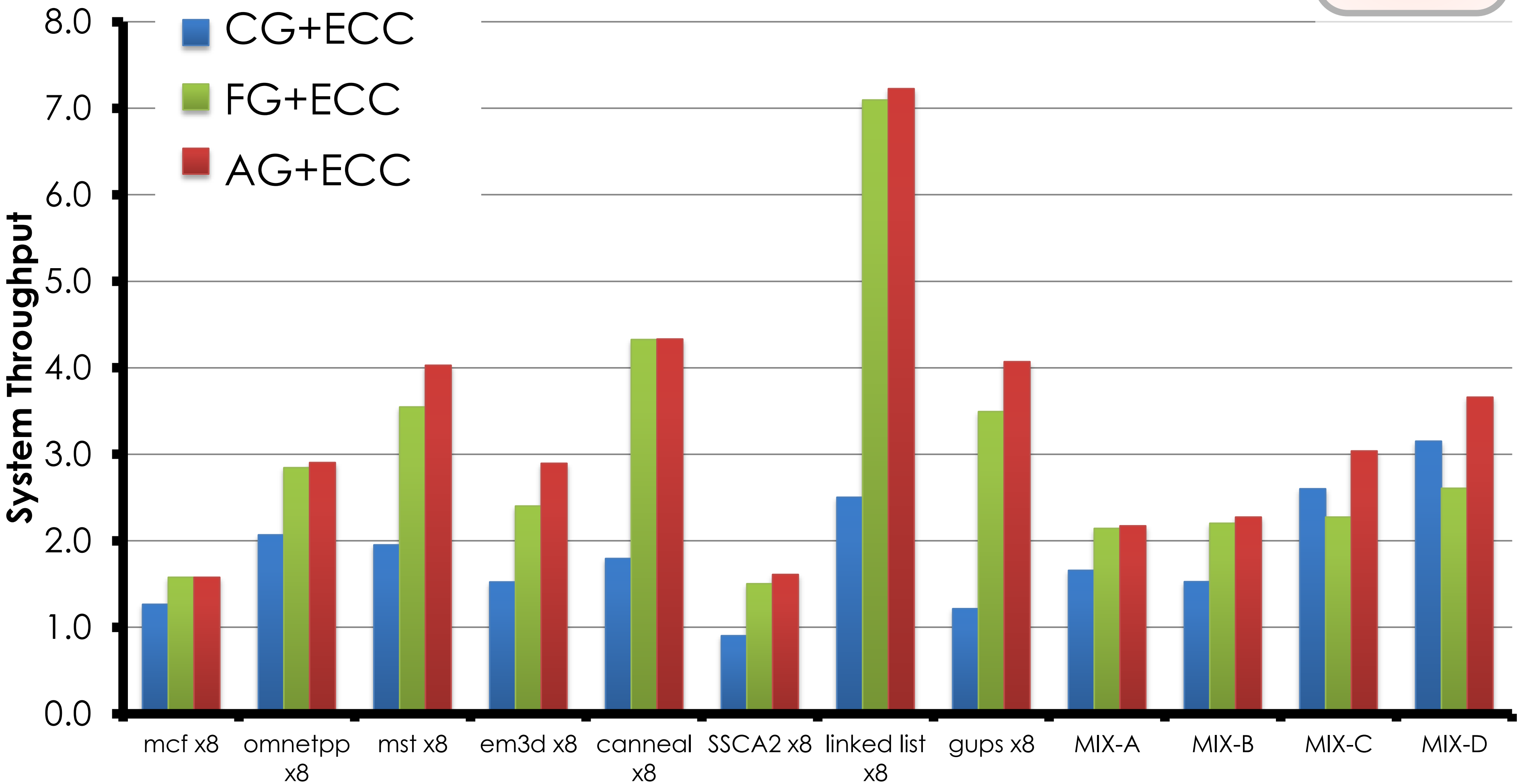
Sector cache
(8 8B sectors per line)

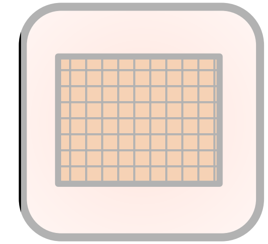
Co-scheduling CG/FG w/ split CG

Sub-Ranked DRAM w/ 2X ABUS
Support both CG and FG



Dynamically **adapt** granularity

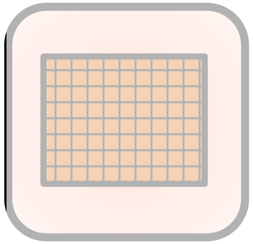




Cost + … → **poor reliability**

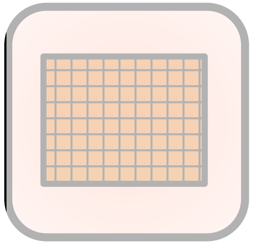
Efficiency + … → **poor reliability**

New + … → **poor reliability**

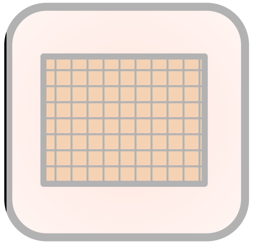


Compensate with error protection

– ECC

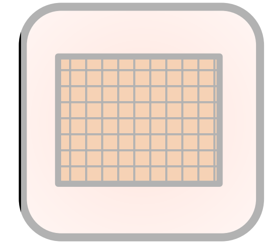


Compensate with **proportional protection**



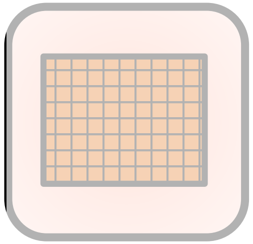
Adaptive reliability

- Adapt the **mechanism**
- Adapt the **error rate**
  - precision (stochastic precision)



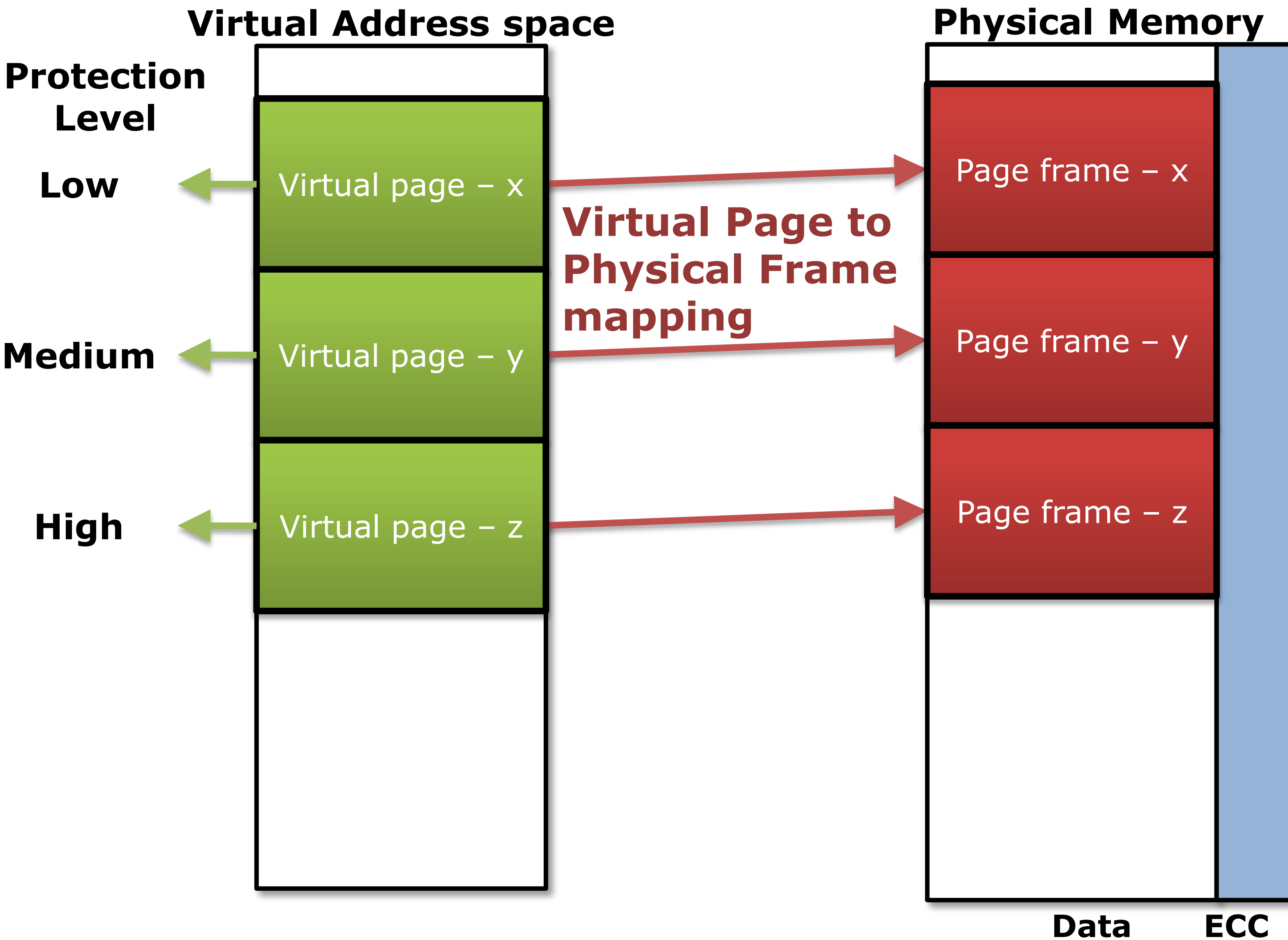
Adaptive reliability

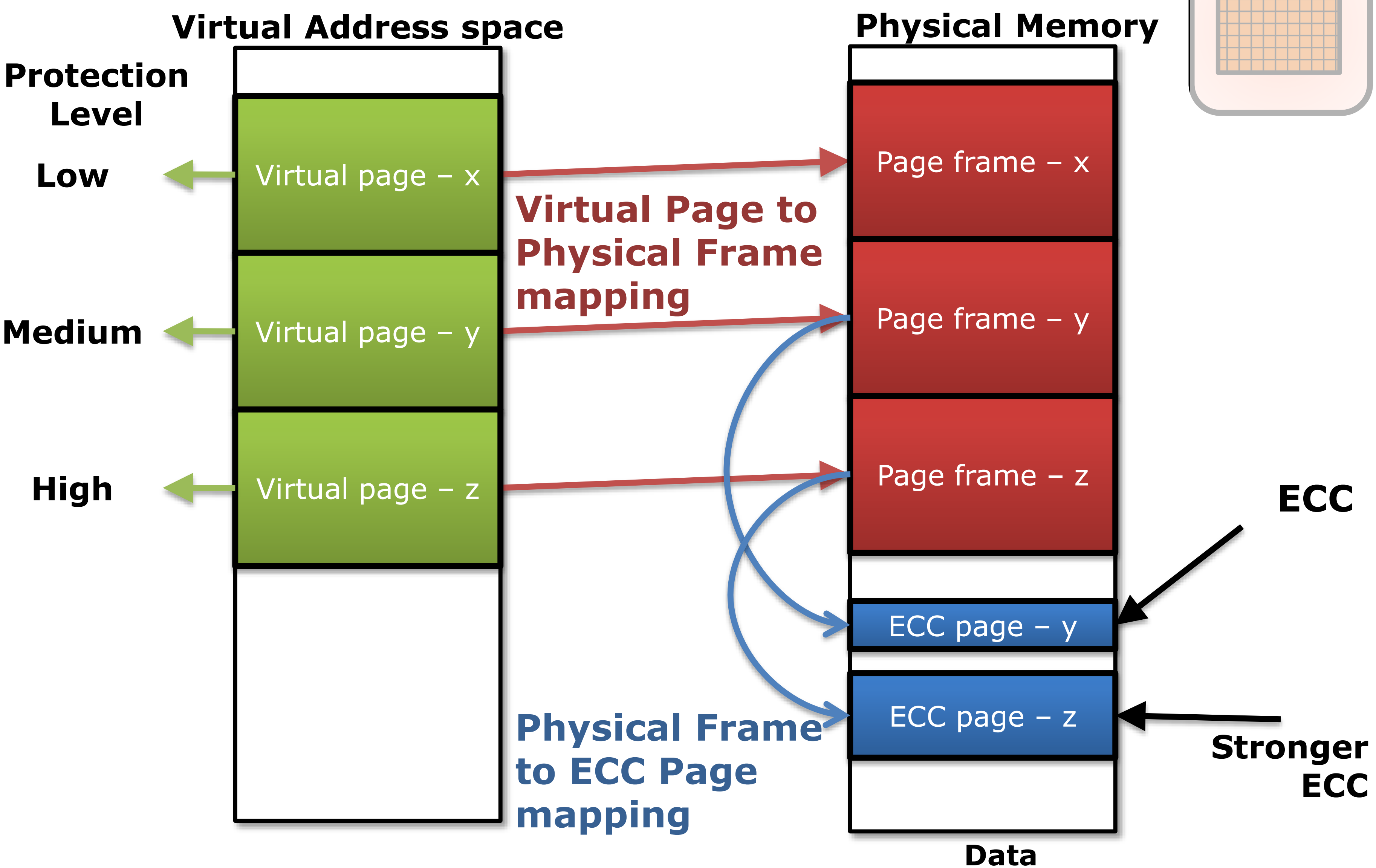
- By **type**
  - Application guided
- By **location**
  - Machine-state guided

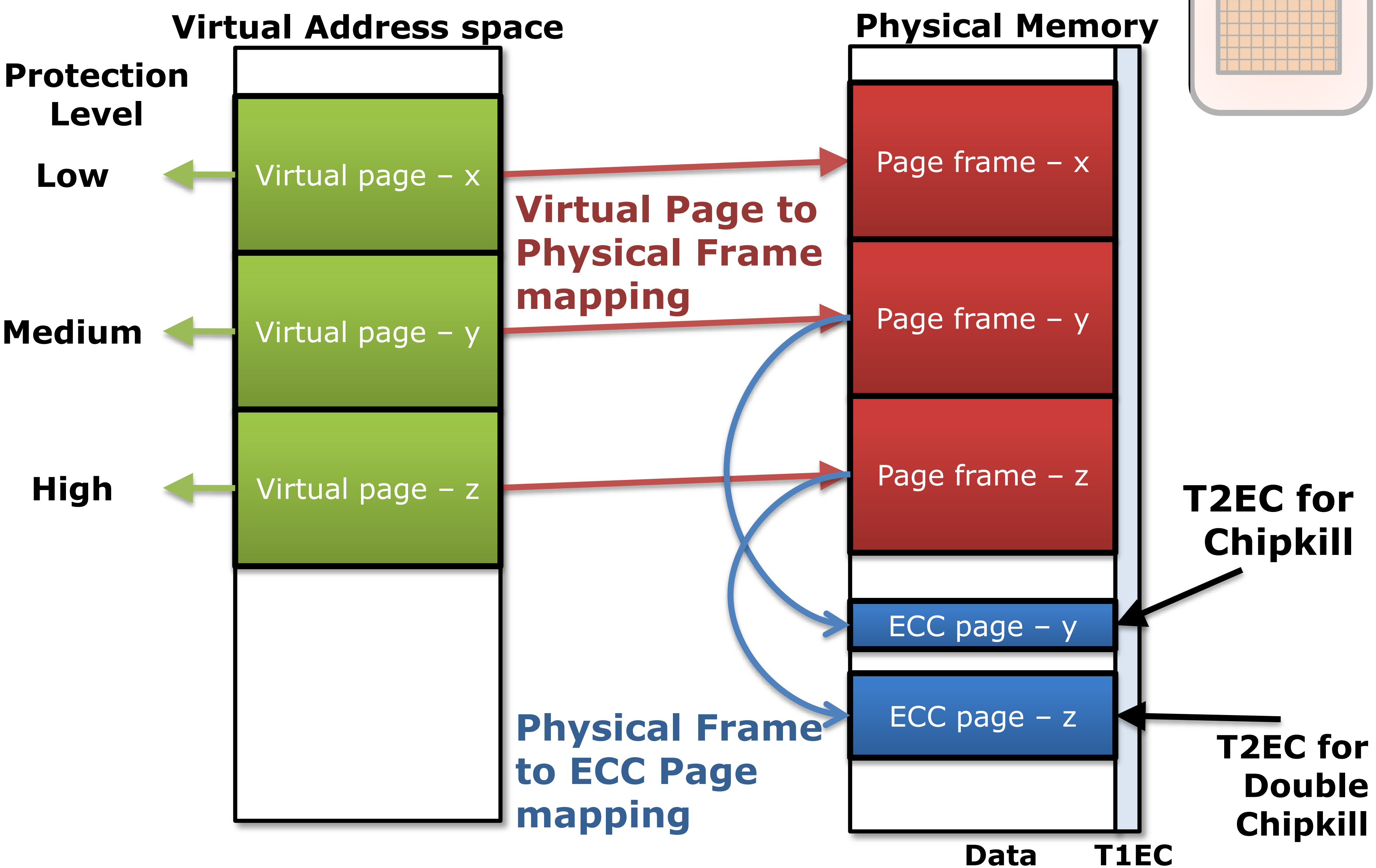


Example: **Virtualized ECC**

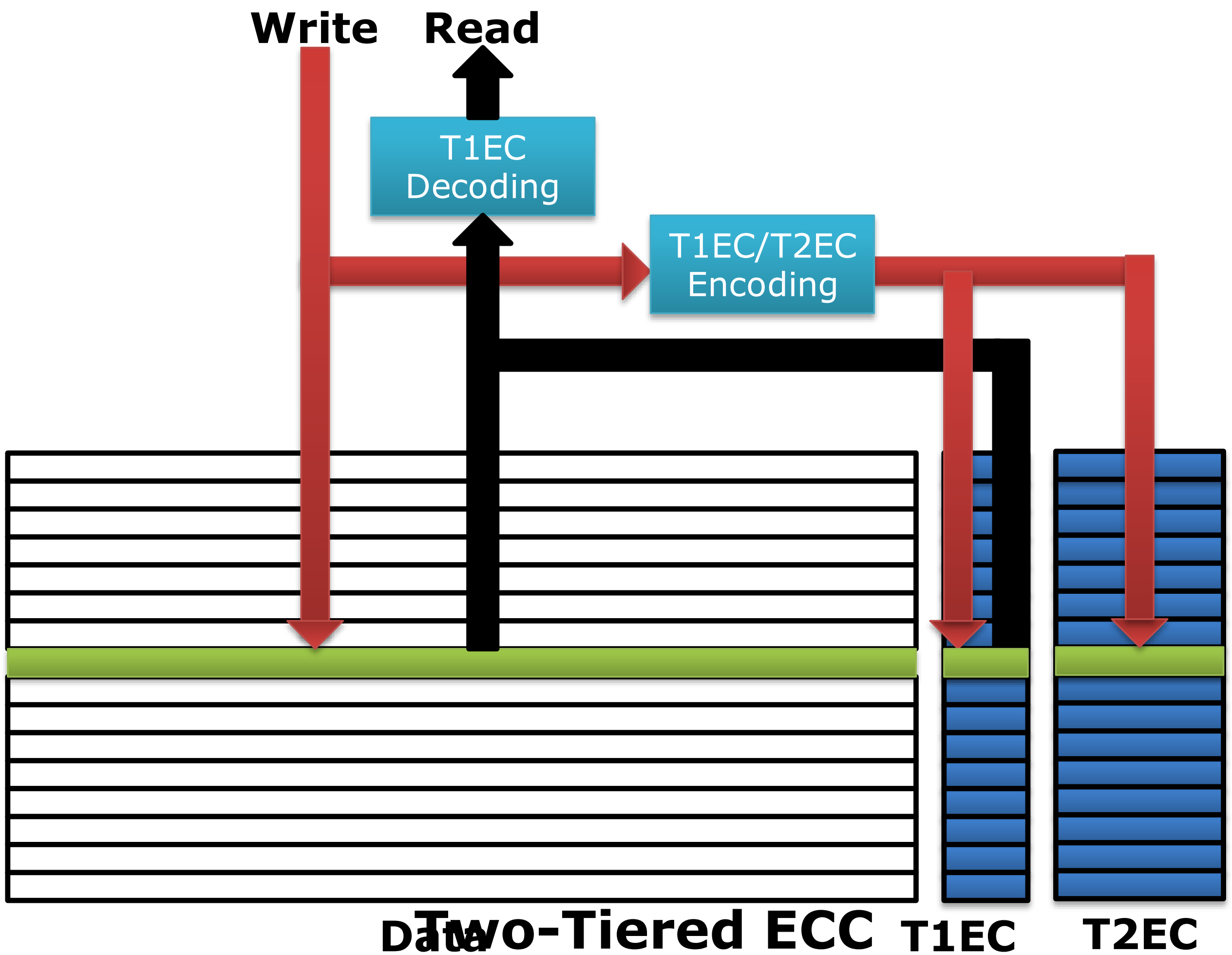
– Adapt the mechanism

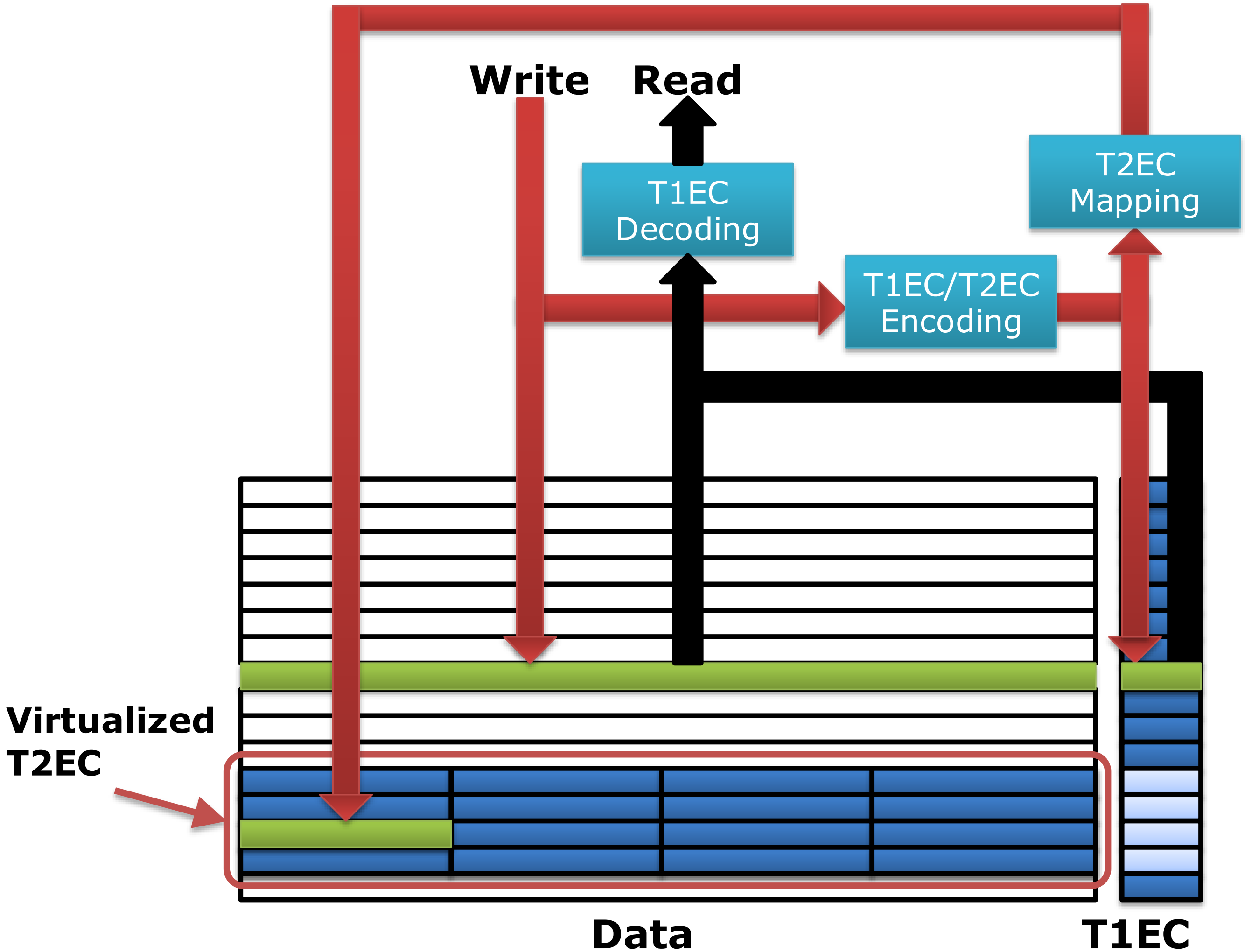

Virtual Address space
Protection Level
Low
Medium
High
Virtual page – x
Virtual page – y
Virtual page – z
Virtual Page to Physical Frame mapping
Physical Memory
Page frame – x
Page frame – y
Page frame – z
Data
ECC


Virtual Address space
Physical Memory
Protection Level
Low
Medium
High
Virtual page – x
Virtual page – y
Virtual page – z
Virtual Page to Physical Frame mapping
Page frame – x
Page frame – y
Page frame – z
ECC page – y
ECC page – z
ECC
Stronger ECC
Physical Frame to ECC Page mapping
Data

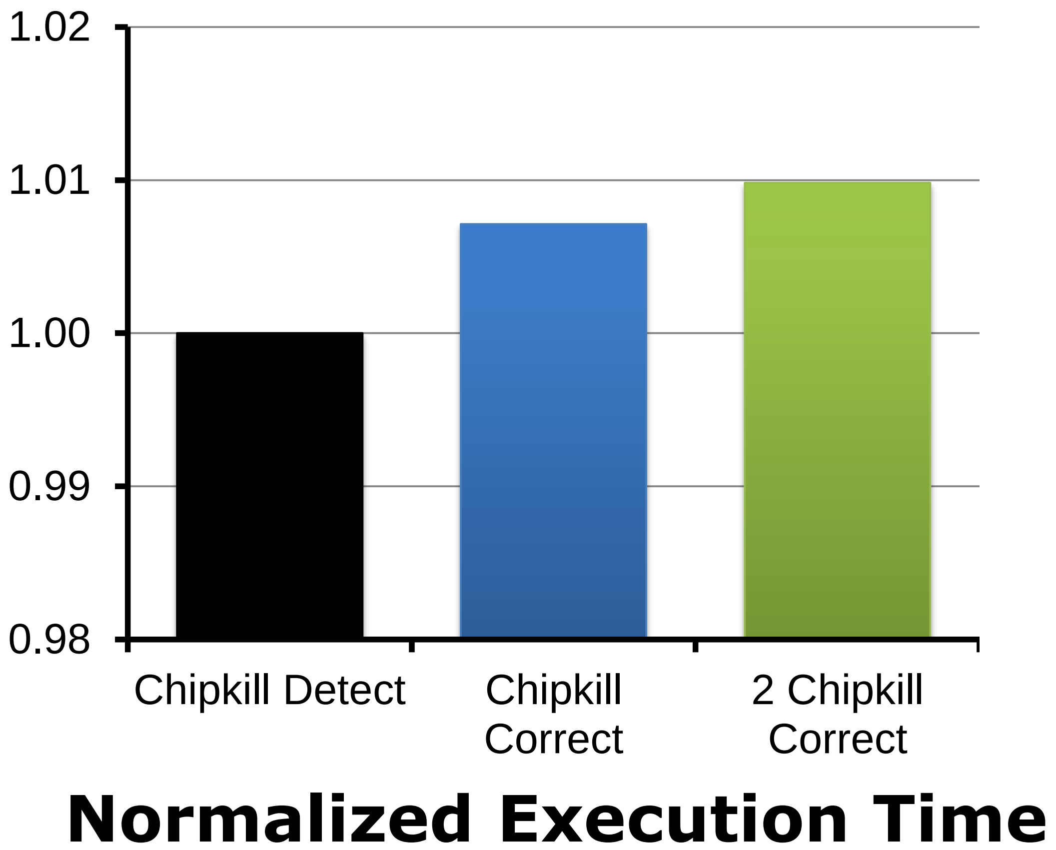

Virtual Address space
Physical Memory
Protection Level
Low
Medium
High
Virtual page – x
Virtual page – y
Virtual page – z
Virtual Page to Physical Frame mapping
Page frame – x
Page frame – y
Page frame – z
ECC page – y
ECC page – z
Physical Frame to ECC Page mapping
T2EC for Chipkill
T2EC for Double Chipkill
Data
T1EC

Two-Tiered ECC

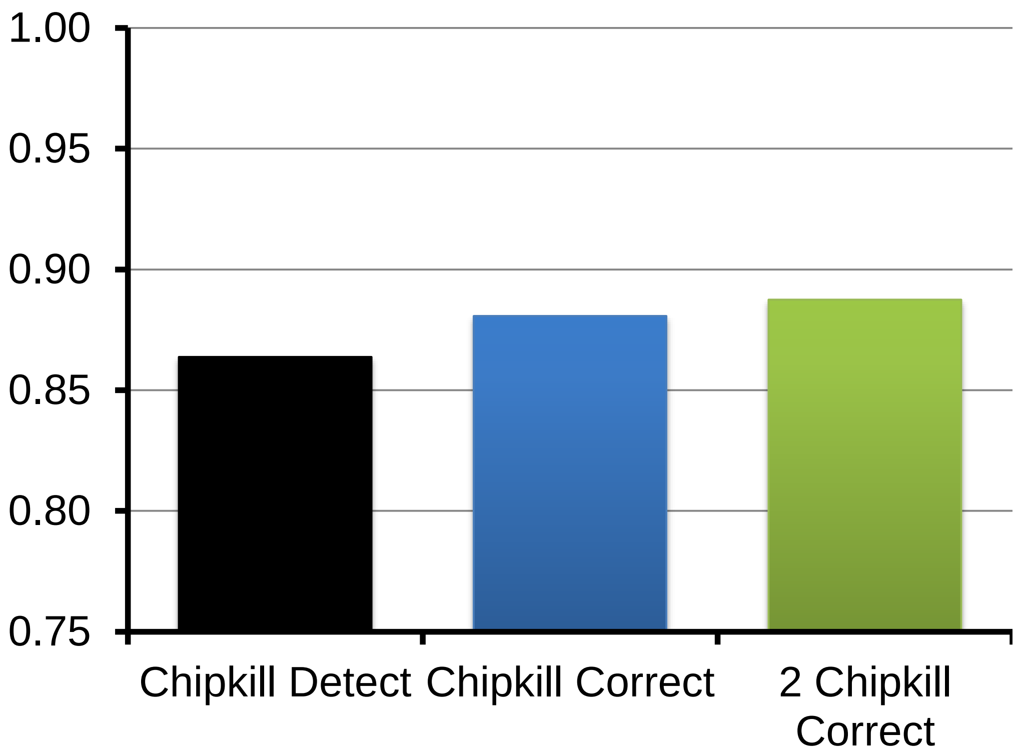



Write
Read
T1EC
Decoding
T2EC
Mapping
T1EC/T2EC
Encoding
Virtualized
T2EC
Data
T1EC

Caching work great



Bonus: **flexibility** of ECC

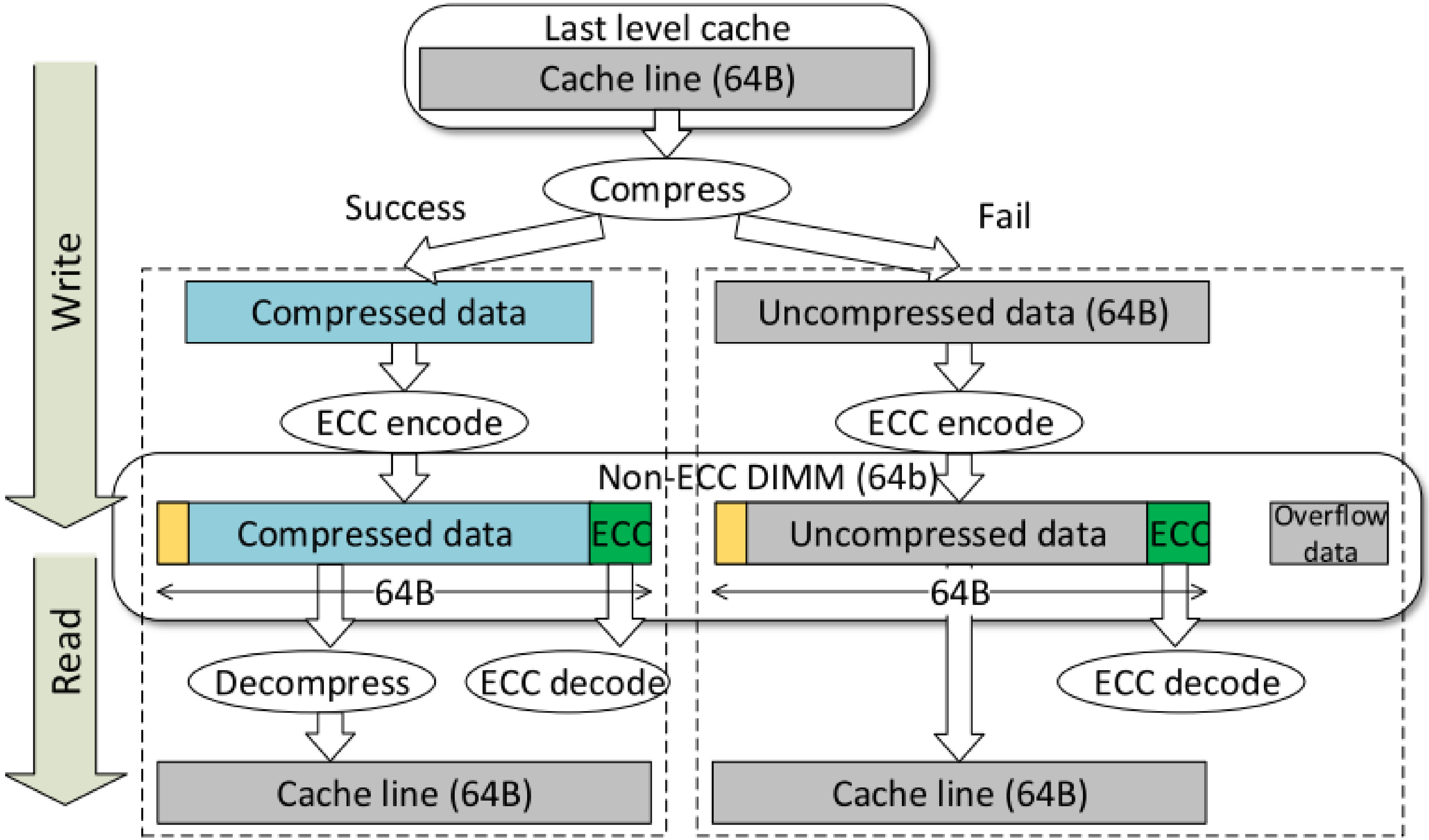


Normalized Energy-Delay Product

Can we do even better?

– Hide second-tier

FrugalECC = Compression + VECC



Adapt resilience scheme or adapt storage?

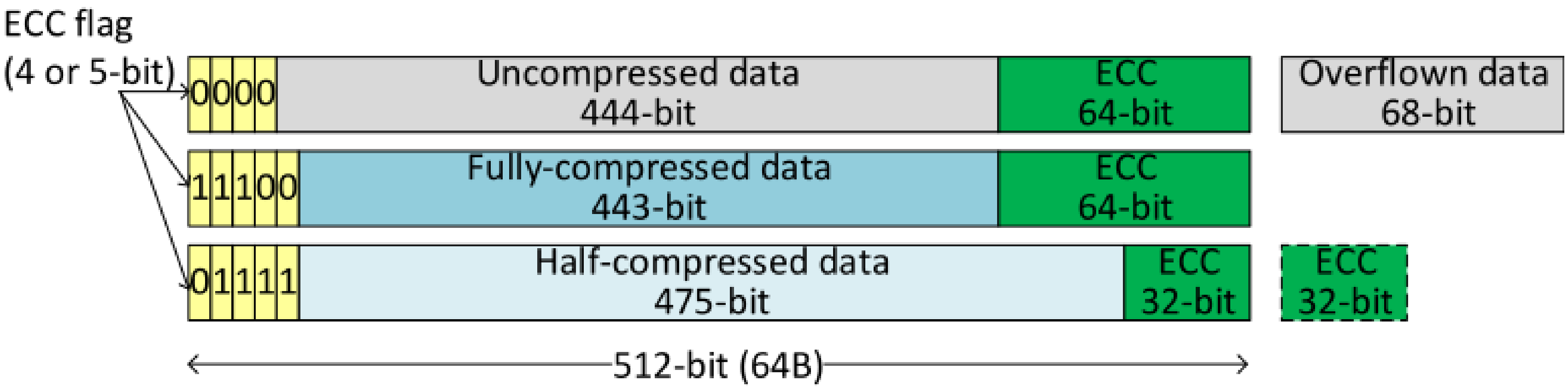
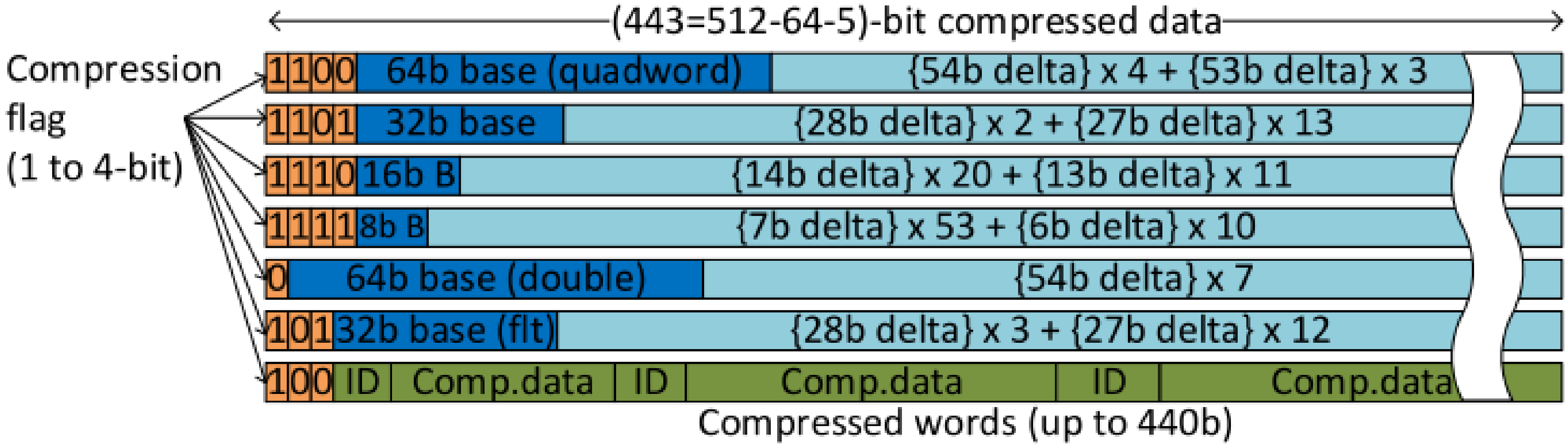
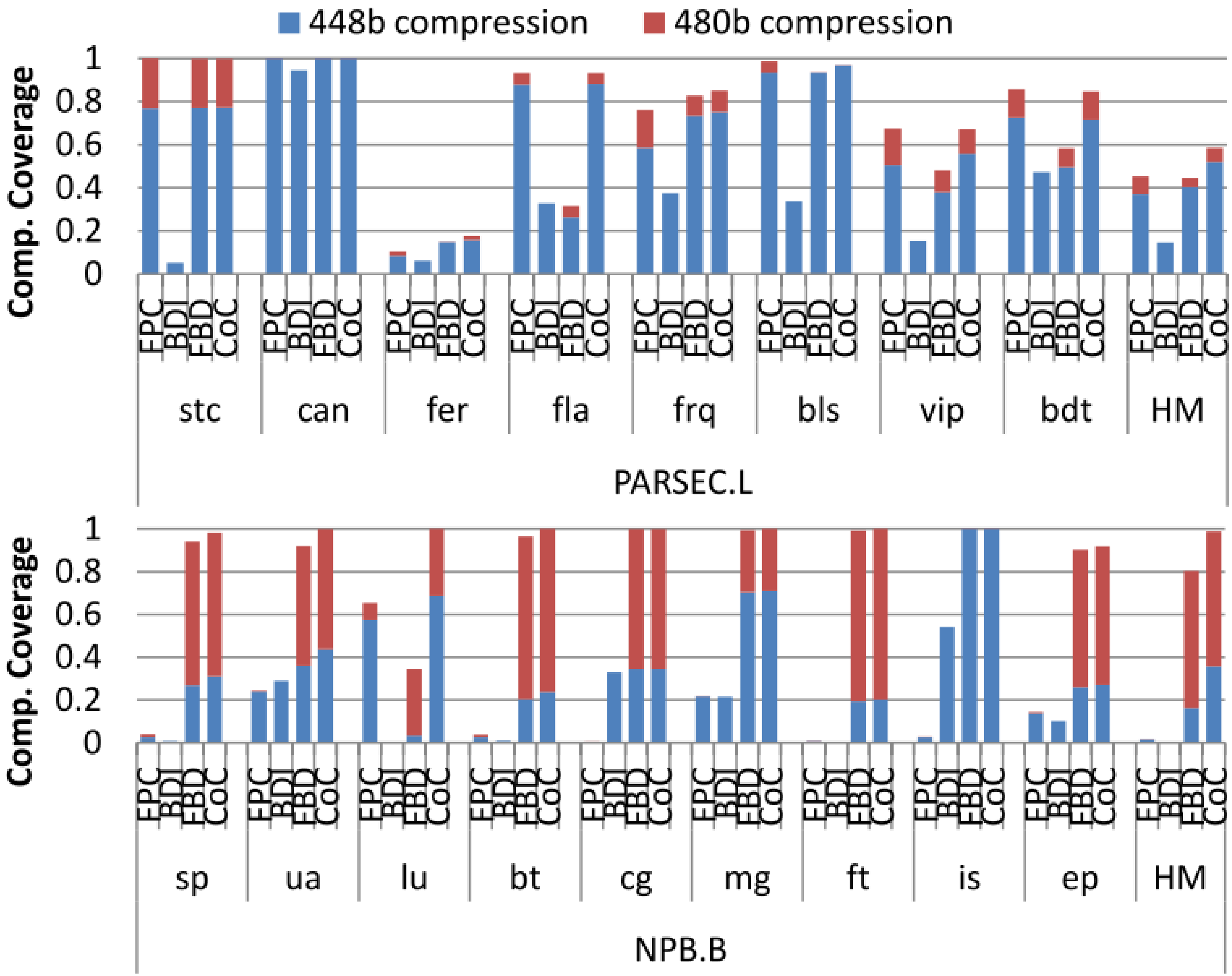
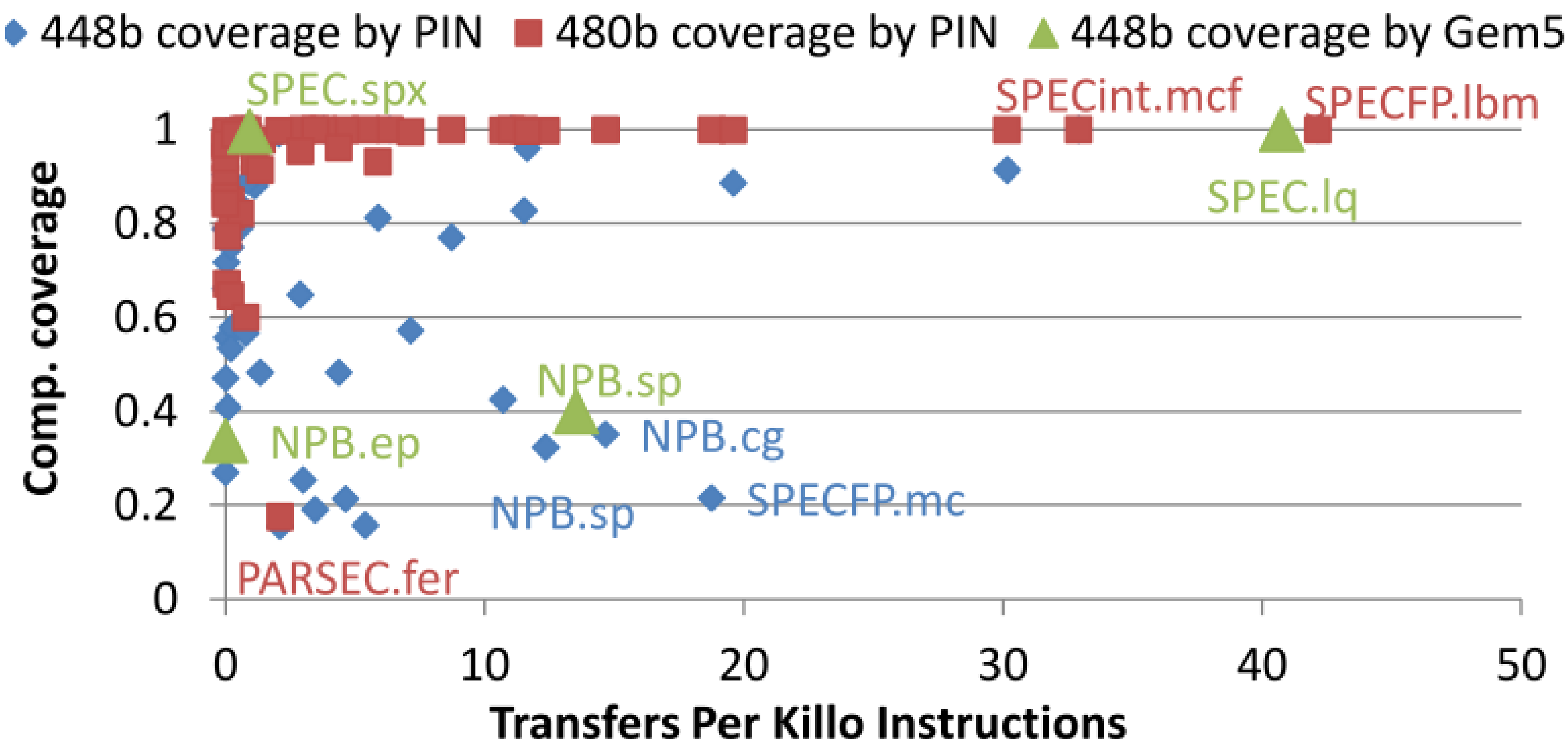


Figure 3: Frugal ECC layout for chipkill-correct (64-bit redundancy).

Adapt compression scheme too! Coverage-oriented-Compression (CoC)

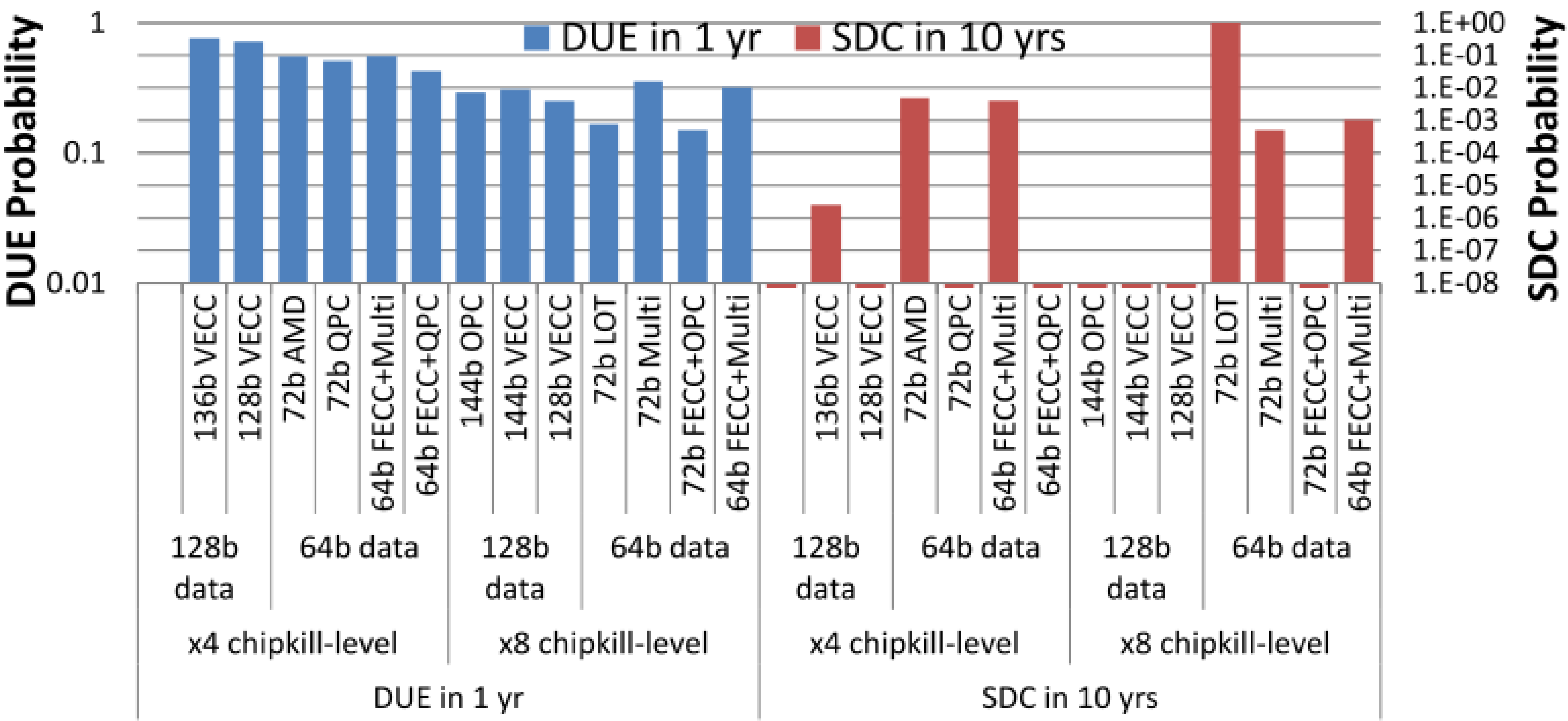


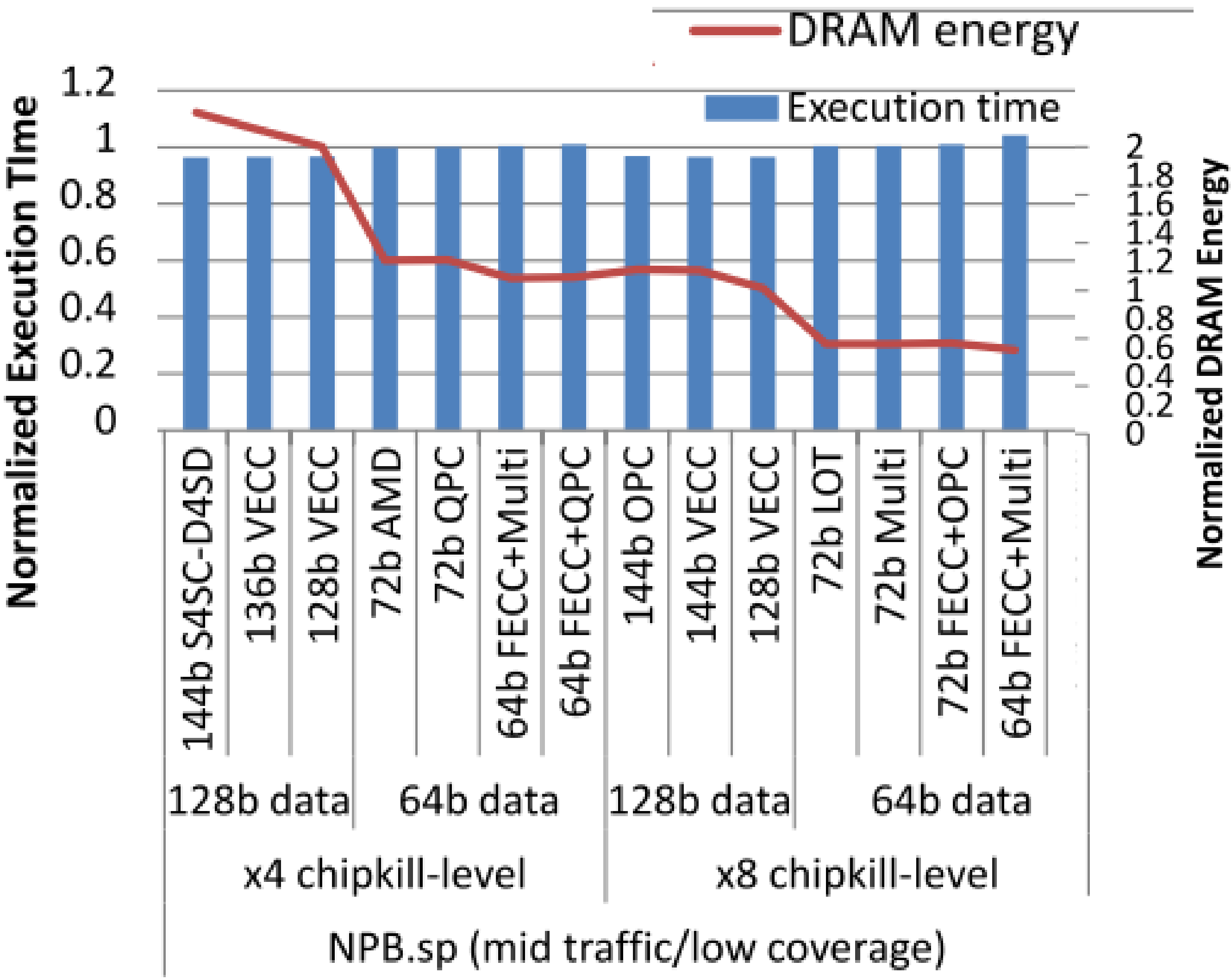

448b compression
480b compression
Comp. Coverage
1
0.8
0.6
0.4
0.2
0
FPC
BDI
FBD
CoC
stc
can
fer
fla
frq
bls
vip
bdt
HM
PARSEC.L
sp
ua
lu
bt
cg
mg
ft
is
ep
HM
NPB.B


448b coverage by PIN
480b coverage by PIN
448b coverage by Gem5
SPEC.spx
SPECint.mcf
SPECFP.lbm
SPEC.lq
NPB.sp
NPB.cg
NPB.ep
SPECFP.mc
NPB.sp
PARSEC.fer
Comp. coverage
1
0.8
0.6
0.4
0.2
0
0
10
20
30
40
50
Transfers Per Killo Instructions

Experiments promising:

- Match or exceed reliability
- Improve power
- Minimal impact on performance


DUE Probability
SDC Probability
DUE in 1 yr
SDC in 10 yrs
1
0.1
0.01
1.E+00
1.E-01
1.E-02
1.E-03
1.E-04
1.E-05
1.E-06
1.E-07
1.E-08
136b VECC
128b VECC
72b AMD
72b QPC
64b FECC+Multi
64b FECC+QPC
144b OPC
144b VECC
128b VECC
72b LOT
72b Multi
72b FECC+OPC
64b FECC+Multi
128b data
64b data
x4 chipkill-level
x8 chipkill-level


DRAM energy
Execution time
Normalized Execution Time
1.2
1
0.8
0.6
0.4
0.2
0
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
Normalized DRAM Energy
144b S4SC-D4SD
136b VECC
128b VECC
72b AMD
72b QPC
64b FECC+Multi
64b FECC+QPC
144b OPC
144b VECC
128b VECC
72b LOT
72b Multi
72b FECC+OPC
64b FECC+Multi
128b data
64b data
128b data
64b data
x4 chipkill-level
x8 chipkill-level
NPB.sp (mid traffic/low coverage)

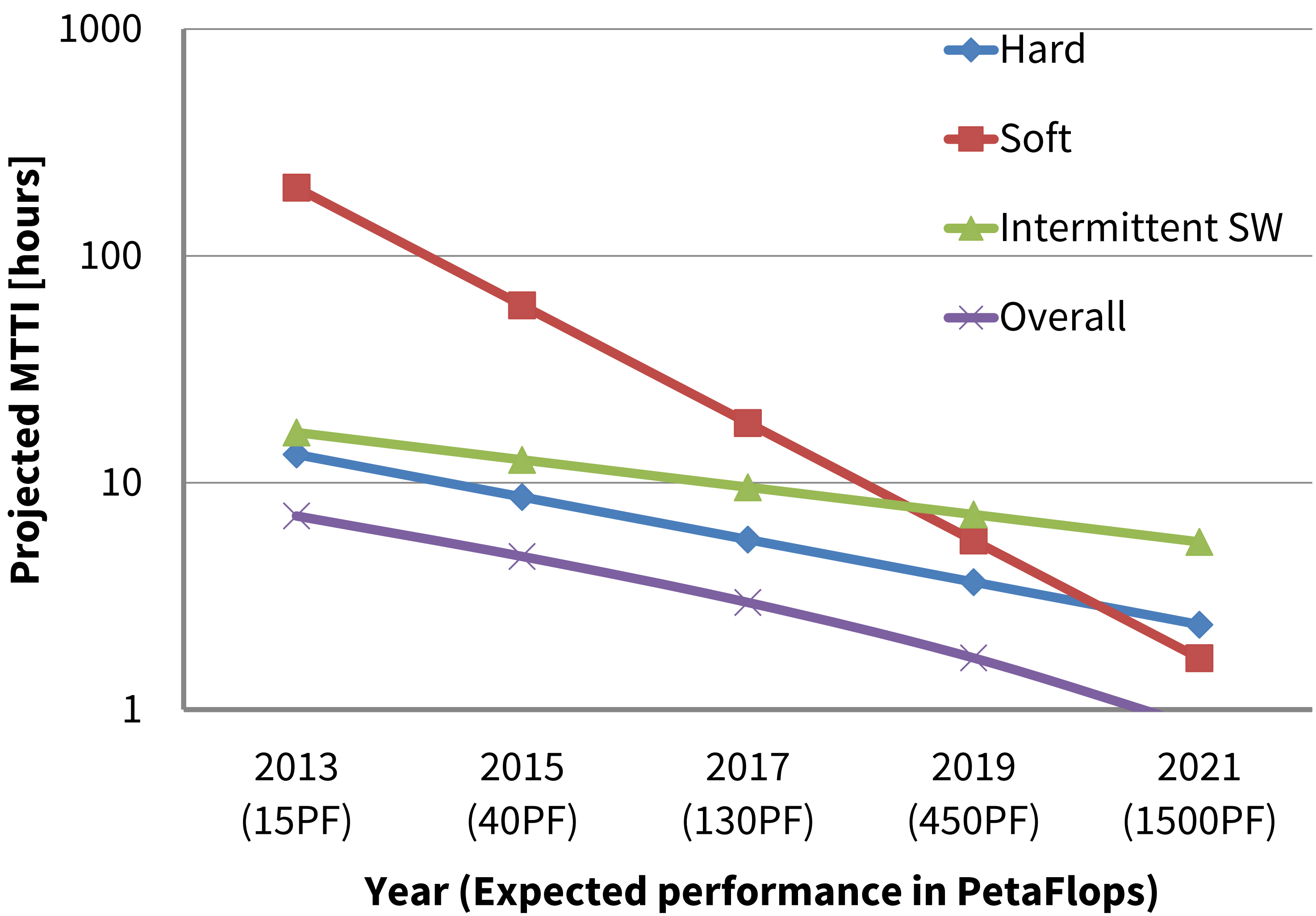
Architecture goals:

- Balance possible and practical
- Enable generality
- Don't hurt the common case

Architecture so far:

- Proportionality
  - Adaptivity
  - SW-HW co-tuning
  - Heterogeneity
- Locality
- Parallelism
- Hierarchy

The **reliability** challenge



Detect → Contain → Repair → Recover

Today's techniques not good enough

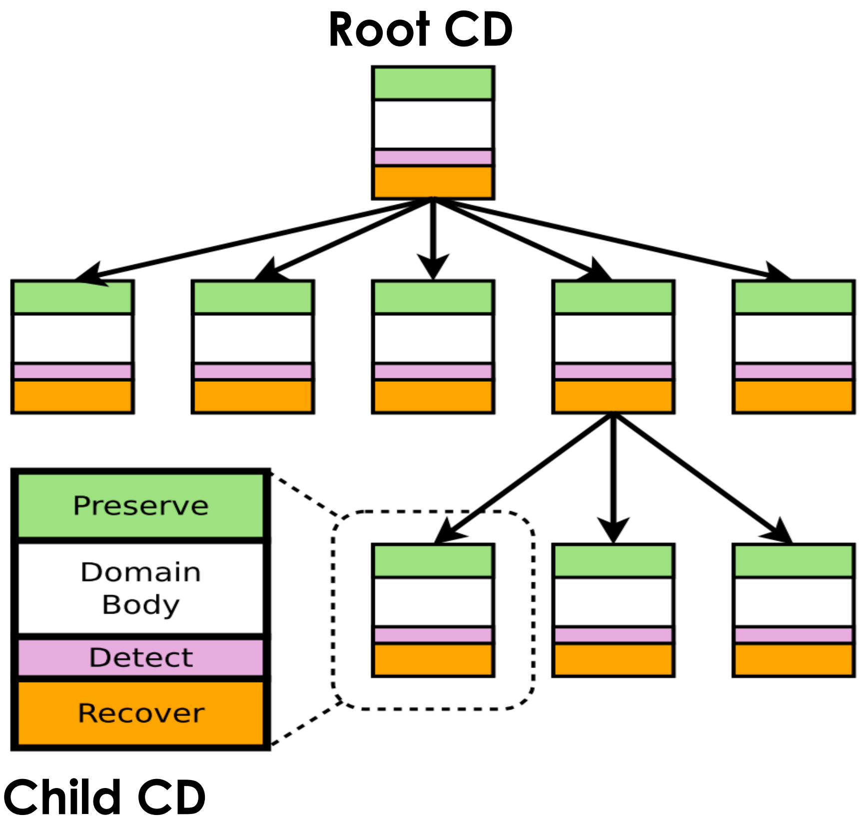
- Hardware support expensive
- Checkpoint-restart doesn't scale

Exascale resilience **must be**:

- Proportional
- Parallel
- Adaptive
- Hierarchical
- Analyzable

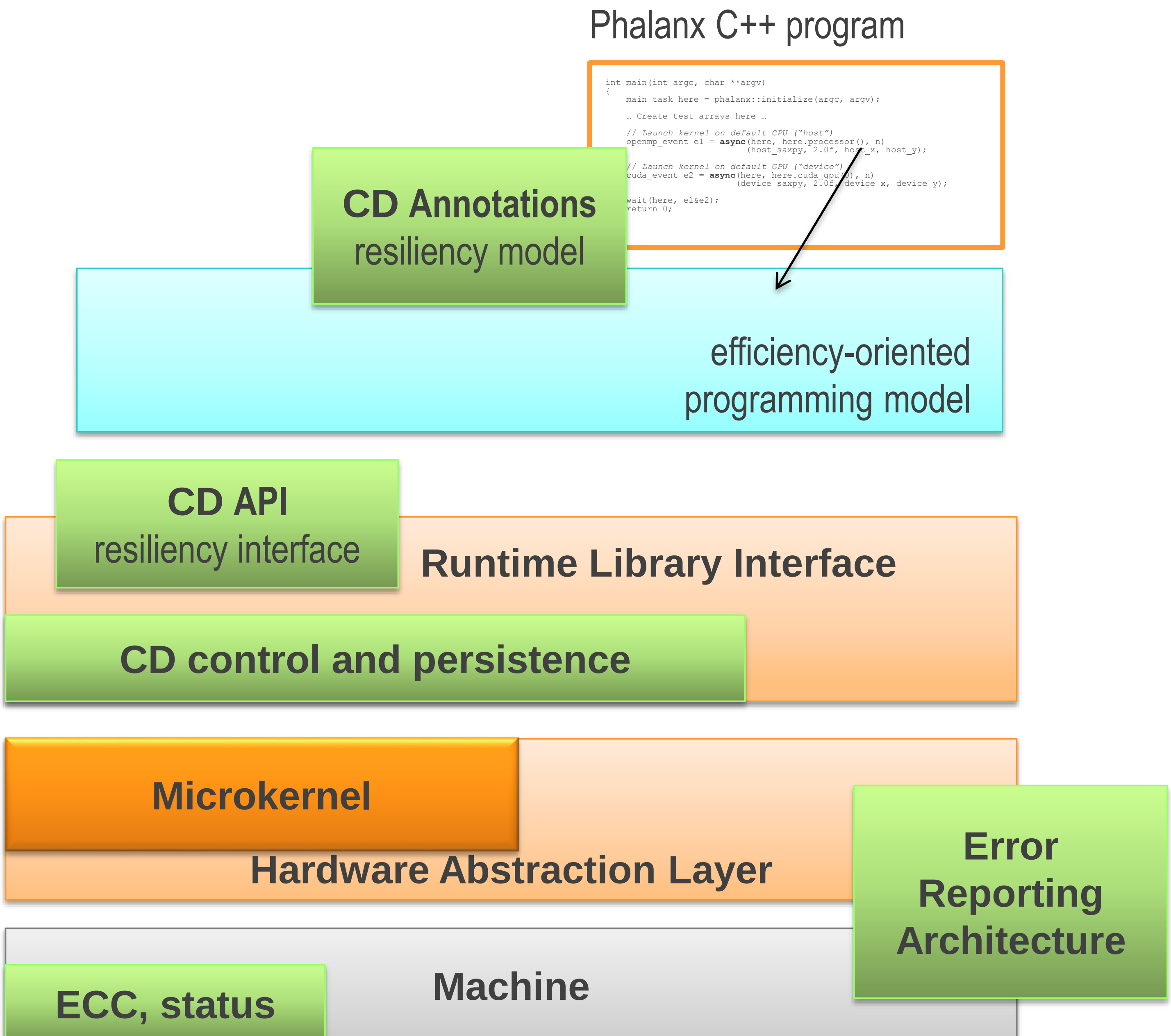
Containment domains

- Embed resilience within application
- Match system hierarchy and characteristics
- Analyzable abstraction consistent across layers



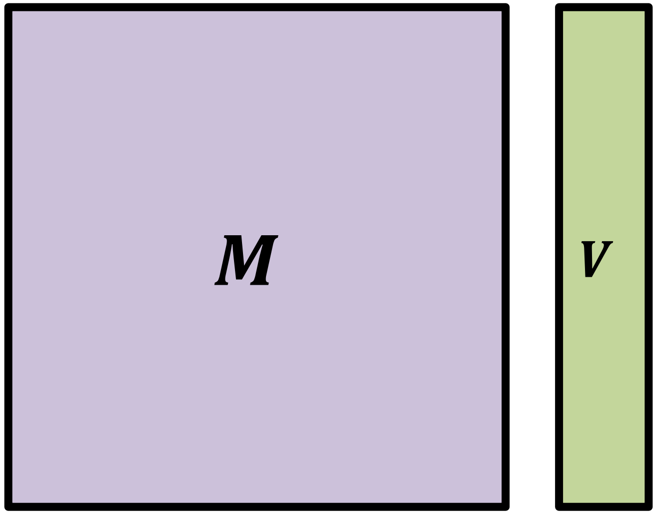
CDs don't communicate erroneous data

CDs have means to recover


Phalanx C++ program
int main(int argc, char **argv)
{
main_task here = phalanx::initialize(argc, argv);
… Create test arrays here …
// Launch kernel on default CPU ("host")
openmp_event e1 = async(here, here.processor(), n)
(host_saxpy, 2.0f, host_x, host_y);
// Launch kernel on default GPU ("device")
cuda_event e2 = async(here, here.cuda_gpu(0), n)
(device_saxpy, 2.0f, device_x, device_y);
wait(here, e1&e2);
return 0;
CD Annotations
resiliency model
efficiency-oriented
programming model
CD API
resiliency interface
Runtime Library Interface
CD control and persistence
Microkernel
Hardware Abstraction Layer
Error
Reporting
Architecture
Machine
ECC, status

101, 106

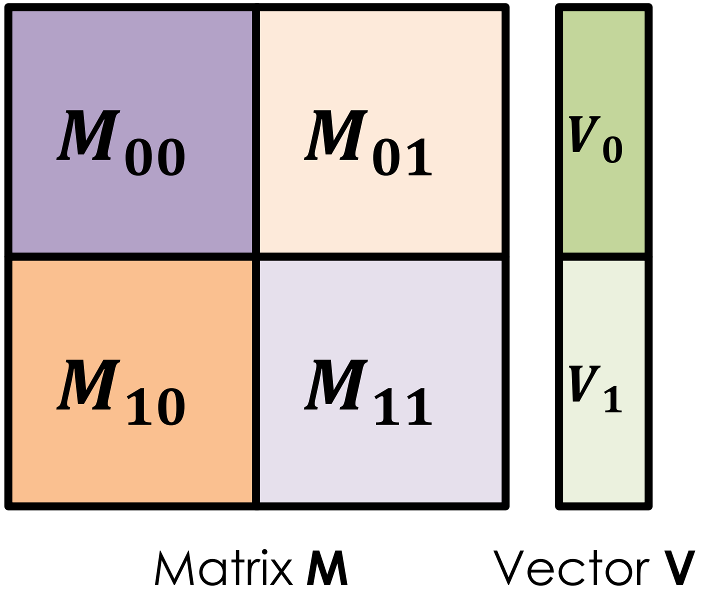
Mapping example: SpMV



Matrix **M** Vector **V**

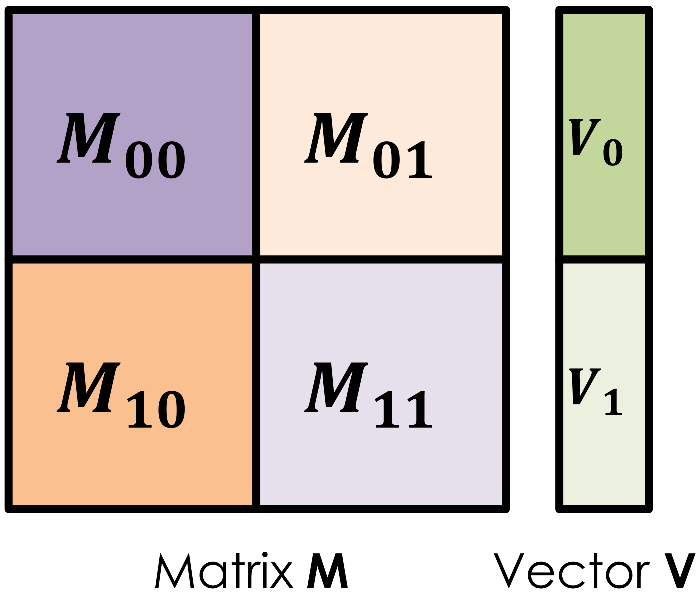
```
void task<inner> SpMV( in M, in V_i,
out R_i){
    forall(…) reduce(…)
        SpMV(M[…],V_i[…],R_i[…]);
}

void task<leaf> SpMV(…){
   for r=0..N
     for c=rowS[r]..rowS[r+1] {
         res_i[r]+=data[c]*V_i[cIdx[c]];
         prevC=c;
     }
}
```

Mapping example: SpMV

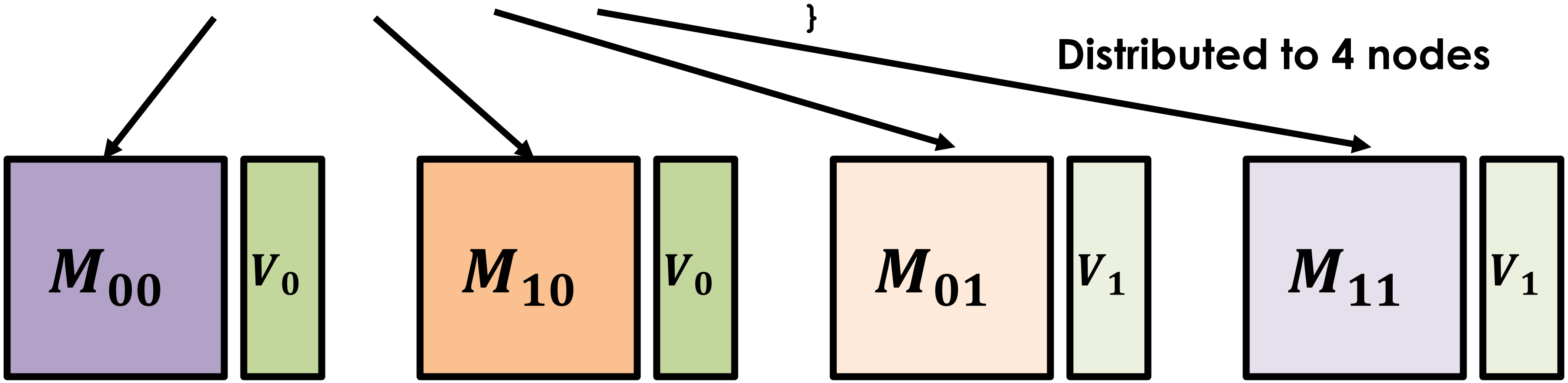



```
void task<inner> SpMV( in M, in Vi,
out Ri){
    forall(…) reduce(…)
        SpMV(M[…],Vi[…],Ri[…]);
}

void task<leaf> SpMV(…){
   for r=0..N
     for c=rowS[r]..rowS[r+1] {
         resi[r]+=data[c]*Vi[cIdx[c]];
         prevC=c;
     }
}
```

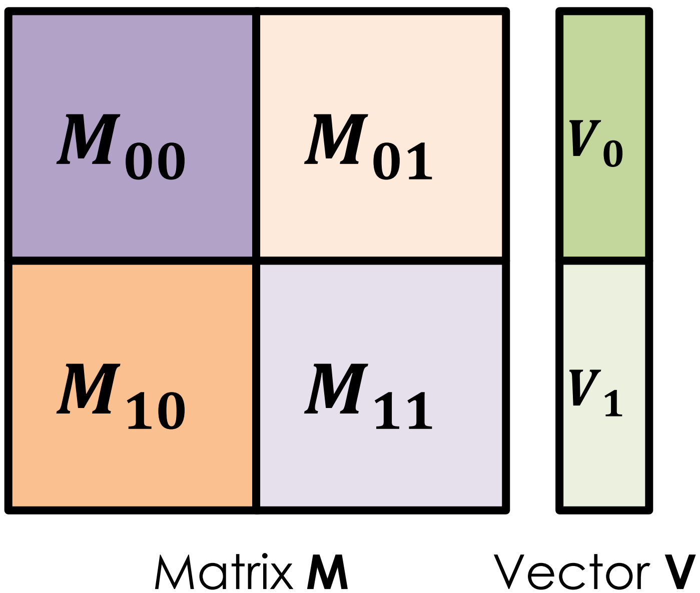
Mapping example: SpMV




```
void task<inner> SpMV( in M, in V_i,
out R_i){
    forall(…) reduce(…)
       SpMV(M[…],V_i[…],R_i[…]);
}

void task<leaf> SpMV(…){
   for r=0..N
     for c=rowS[r]..rowS[r+1] {
        res_i[r]+=data[c]*V_i[cIdx[c]];
        prevC=c;
     }
}
```

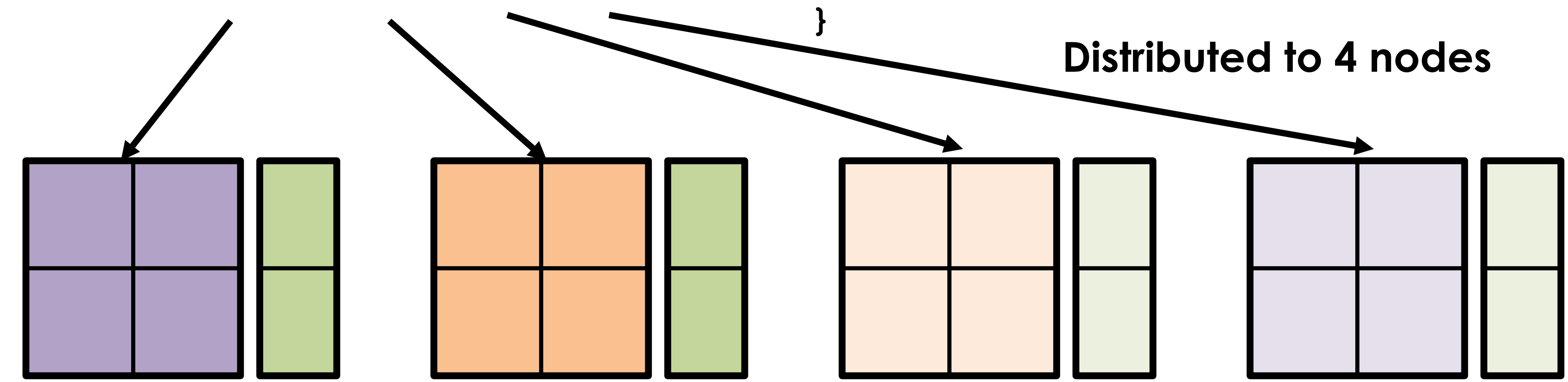


Mapping example: SpMV

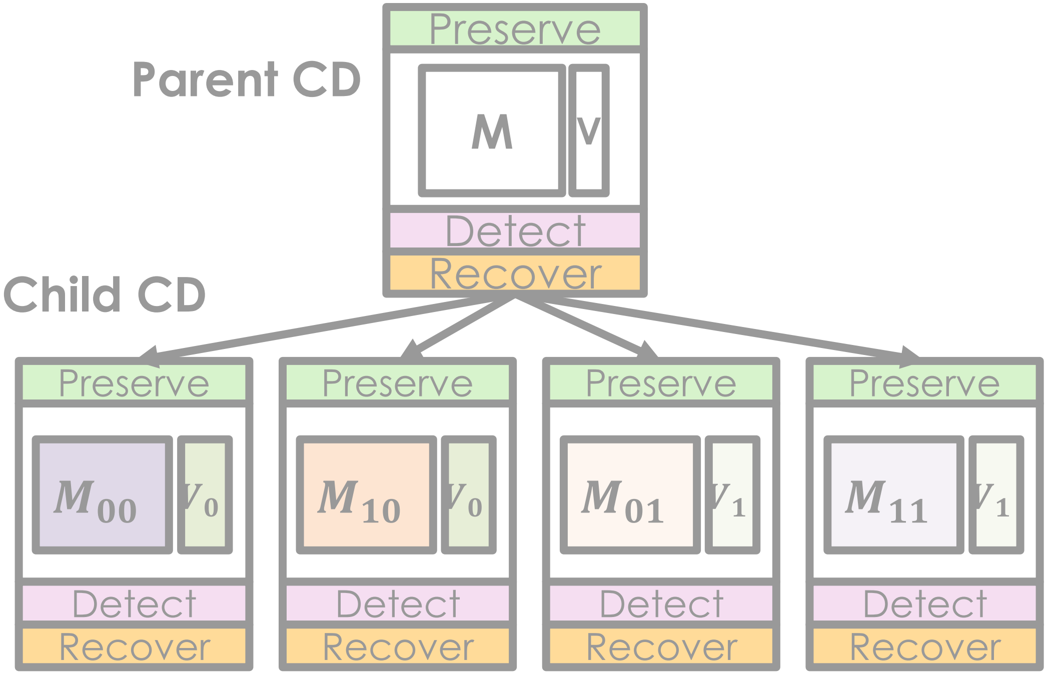


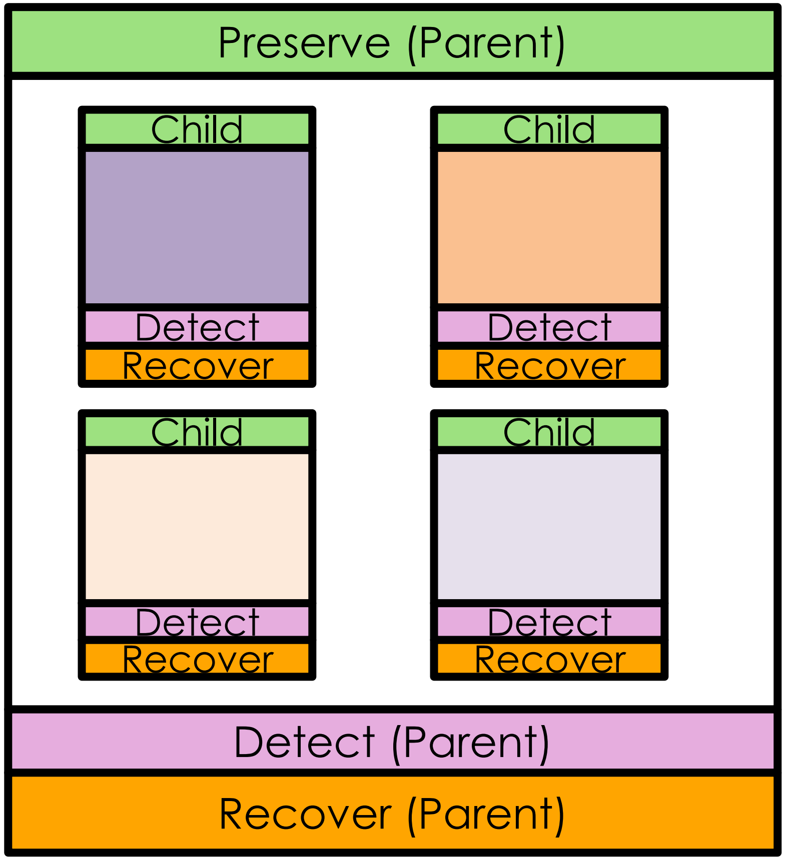

```
void task<inner> SpMV( in M, in V_i, out R_i){
    forall(…) reduce(…)
       SpMV(M[…],V_i[…],R_i[…]);
}

void task<leaf> SpMV(…){
  for r=0..N
    for c=rowS[r]..rowS[r+1] {
       res_i[r]+=data[c]*V_i[cIdx[c]];
      prevC=c;
    }
}
```

Distributed to 4 nodes



Mapping example: SpMV

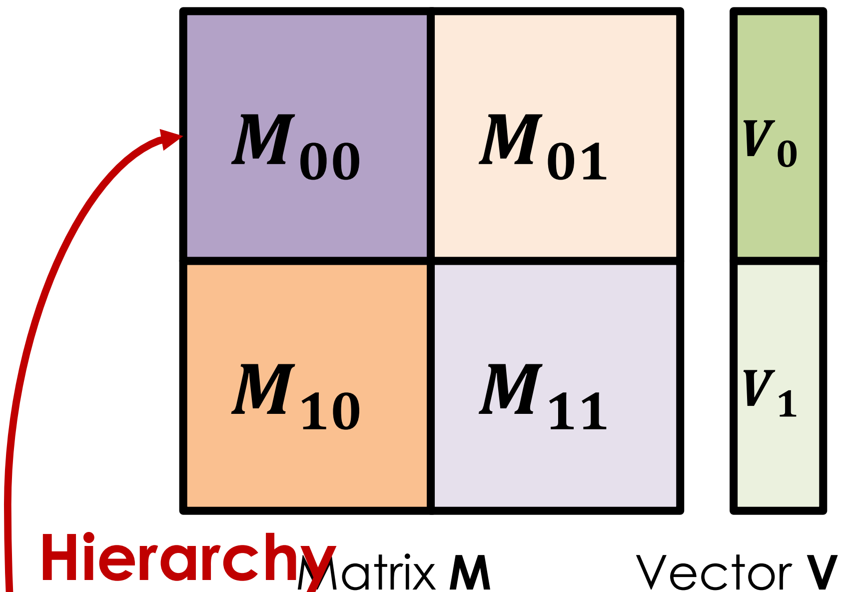




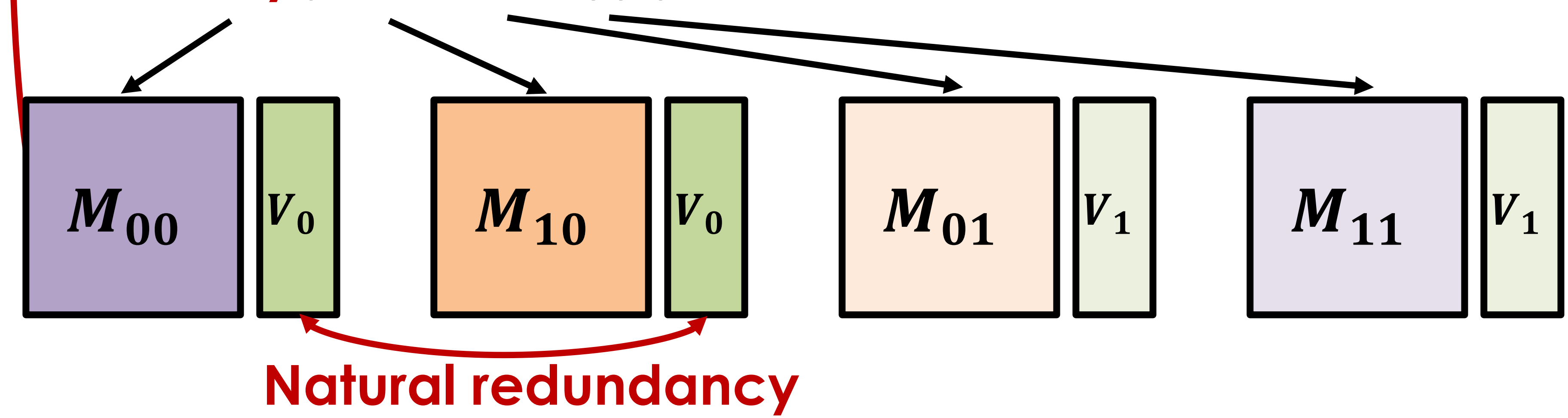
```
void task<inner> SpMV(in M, in Vi, out Ri) {
 cd = begin(parentCD);
 preserve_via_copy(cd, matrix, …);
 forall(…) reduce(…)
   SpMV(M[…],Vi[…],Ri[…]);
 complete(cd);
}

void task<leaf> SpMV(…) {
 cd = begin(parentCD);
 preserve_via_copy(cd, matrix, …);
 preserve_via_parent(cd, veci, …);
 for r=0..N
  for c=rowS[r]..rowS[r+1] {
    resi[r]+=data[c]*Vi[cIdx[c]];
    check {fault<fail>(c > prevC);}
    prevC=c;
  }
 complete(cd);
}
```

API
begin
configure
preserve
check
complete

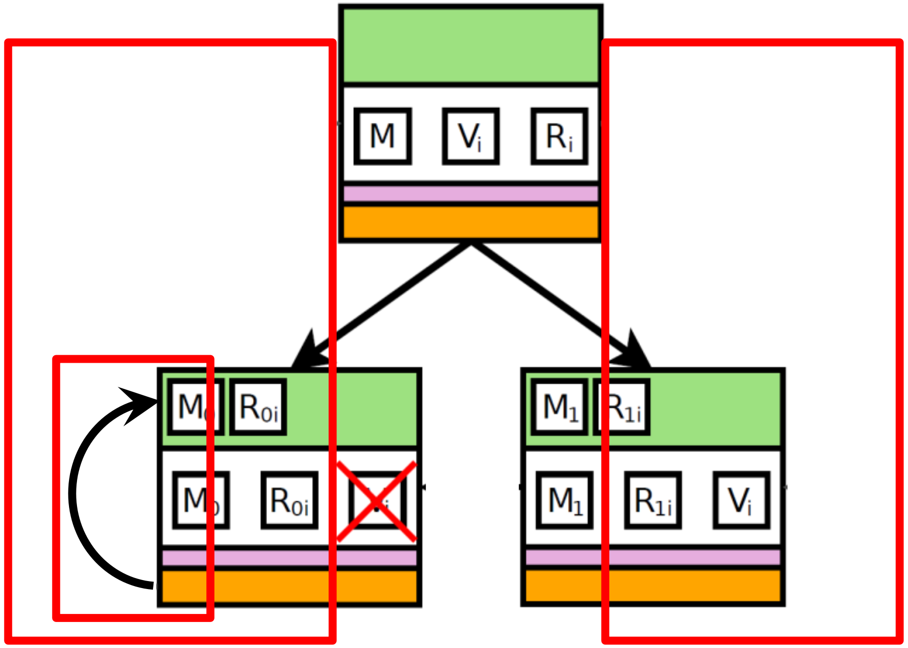
SpMV preservation tuning




```
void task<leaf> SpMV(…) {
 cd = begin(parentCD);
 preserve_via_copy(cd, matrix, …);
 preserve_via_parent(cd, vec_i, …);
 for r=0..N
  for c=rowS[r]..rowS[r+1] {
    res_i[r]+=data[c]*V_i[cIdx[c]];
    check {fault<fail>(c > prevC);}
    prevC=c;
  }
 complete(cd);
}
```

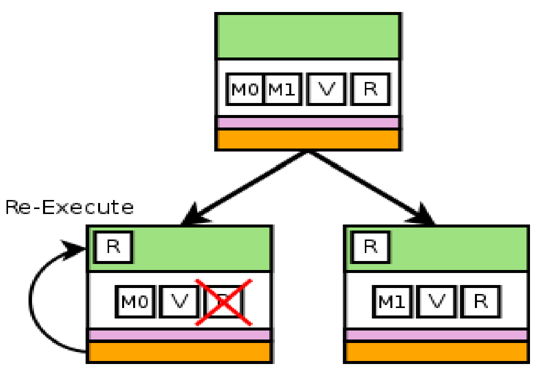


Concise abstraction for complex behavior

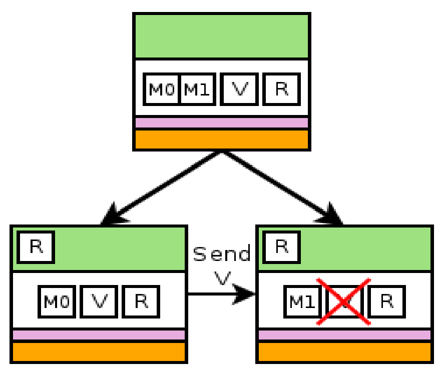



```
void task<leaf> SpMV(…) {
 cd = begin(parentCD);
 preserve_via_copy(cd, matrix, …);
 preserve_via_parent(cd, vec_i, …);
 for r=0..N
  for c=rowS[r]..rowS[r+1] {
    res_i[r]+=data[c]*V_i[cIdx[c]];
    check {fault<fail>(c > prevC);}
    prevC=c;
  }
 complete(cd);}
```

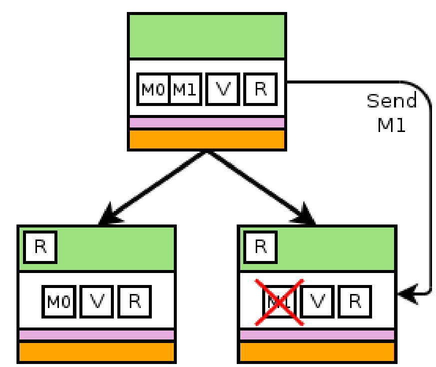
Local copy or regen



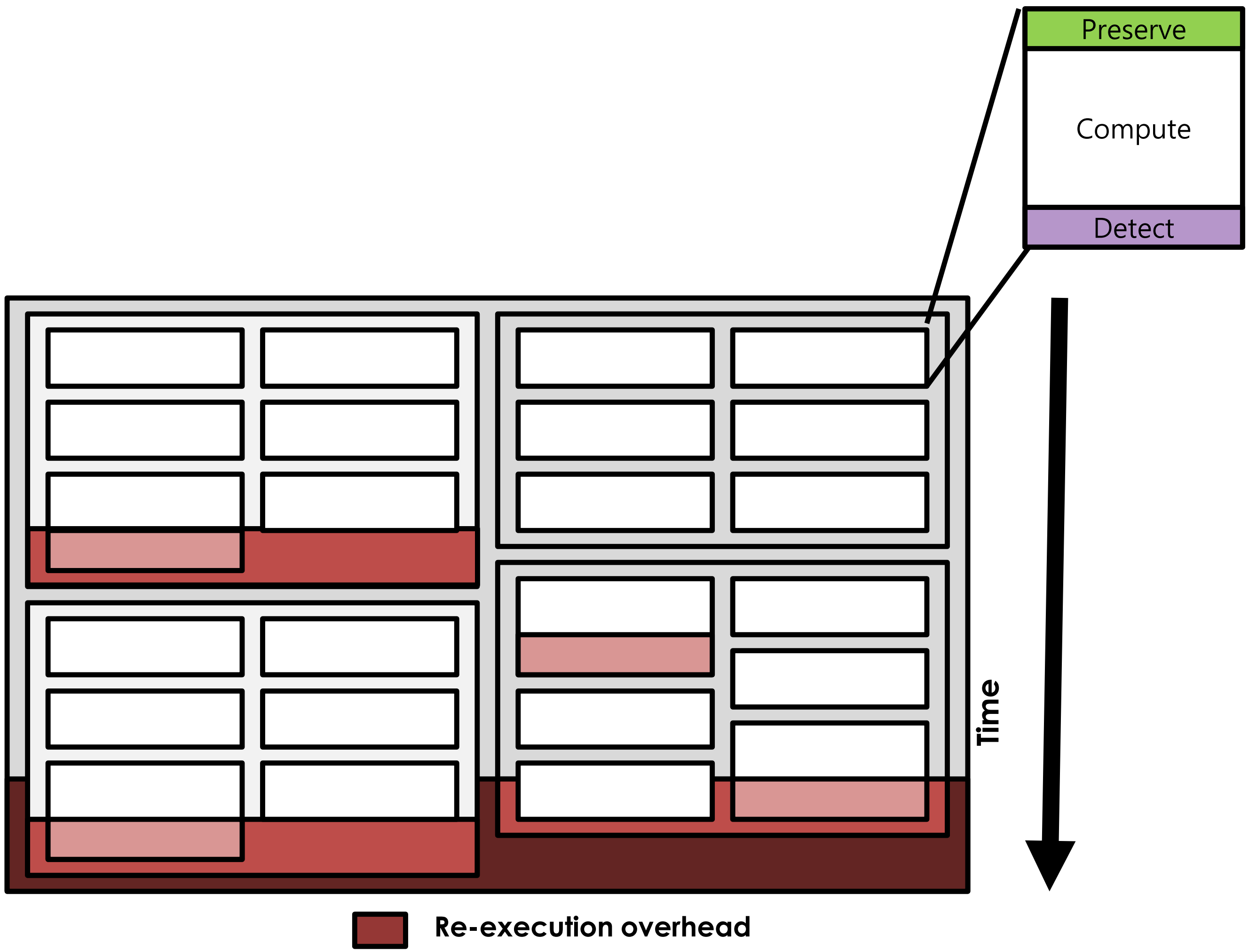
Sibling



Parent (unchanged)

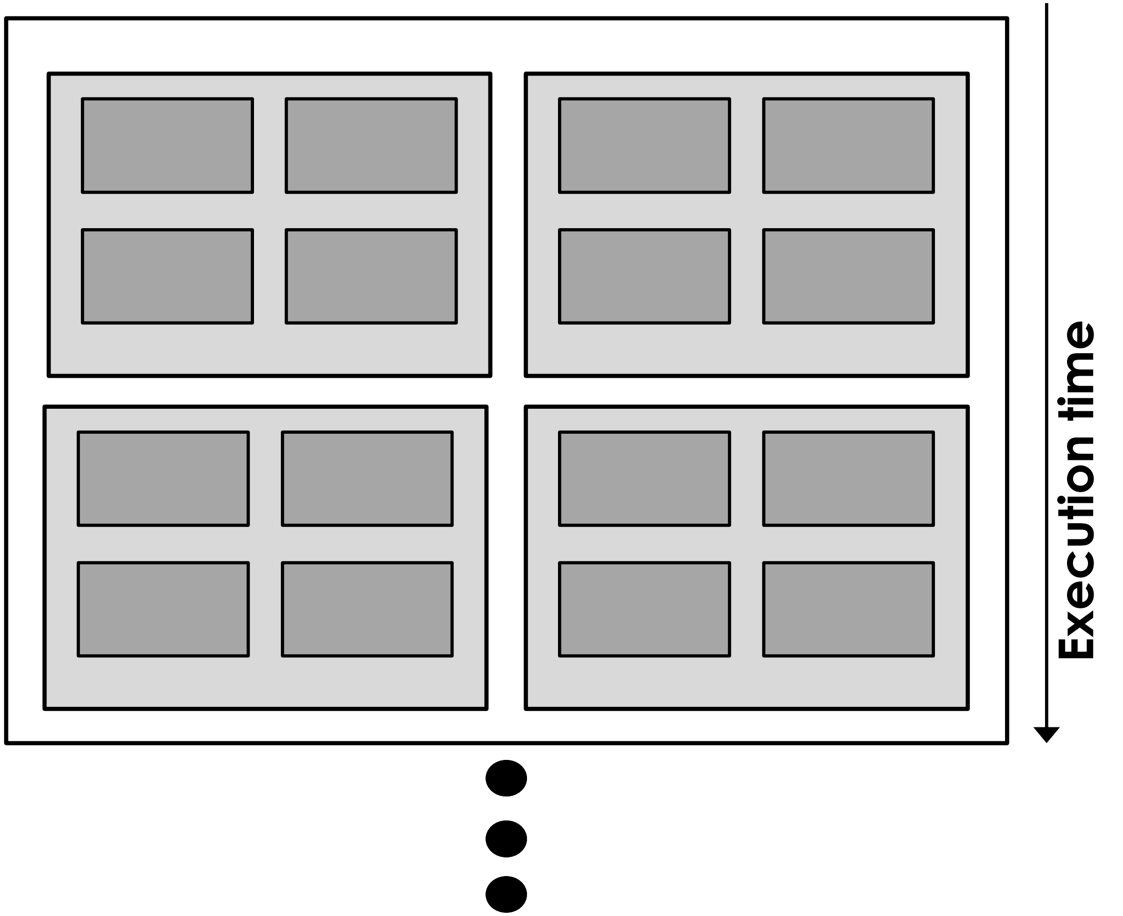


Recovery: concurrent, hierarchical, adaptive



Analyzable

- Application model (tree)
- Overhead model
- Fault model



Analyzable

$q[0,m,n] = (1-p_c)^{m \cdot n}$

- Probability that all child CDs experienced no failures

$q[x,m,n] = \left(\sum_{i=0}^{x}\left(\binom{i+m-1}{i}p_c^i(1-p_c)^m\right)\right)^n$

- Probability that all child CDs experienced at most x failures in the m serial CDs

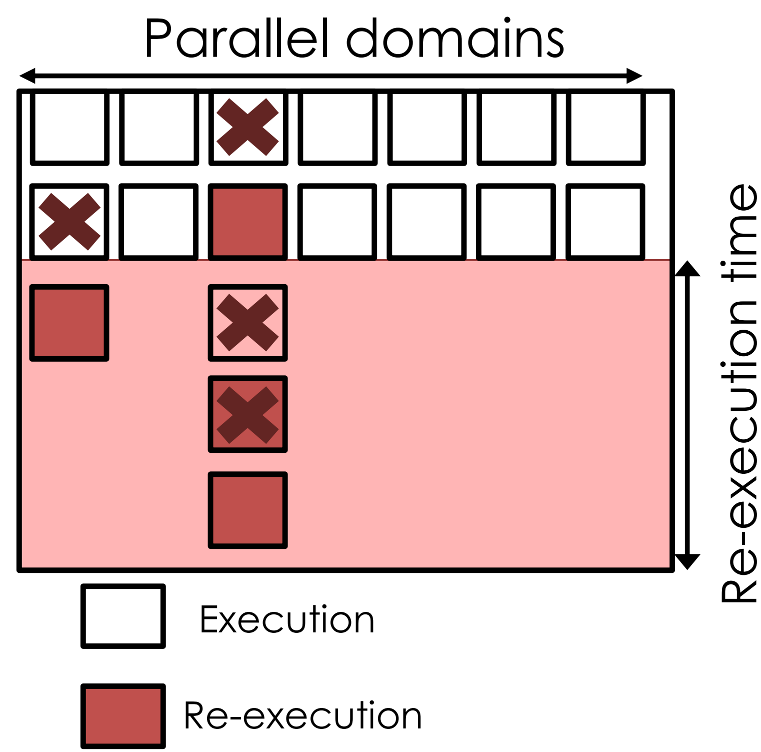
$d[0,m,n] = q[0,m,n]$

- Probability that all child CDs experienced exactly 0 failures

$d[x,m,n] = q[x,m,n] - q[x-1,m,n]$

- Probability that sibling with the most failure experiences exactly x failures

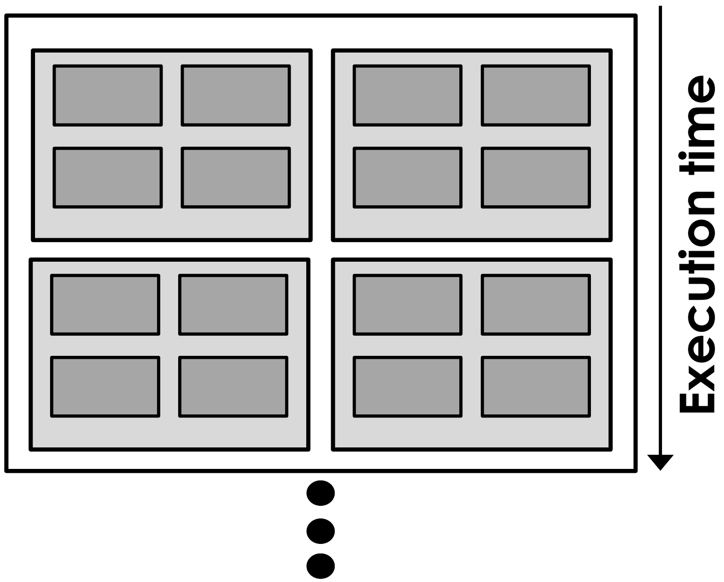
$T_{parent}[x,m,n] = \sum_{i=0}^{\infty}(i+m)T_c d[i,m,n]$

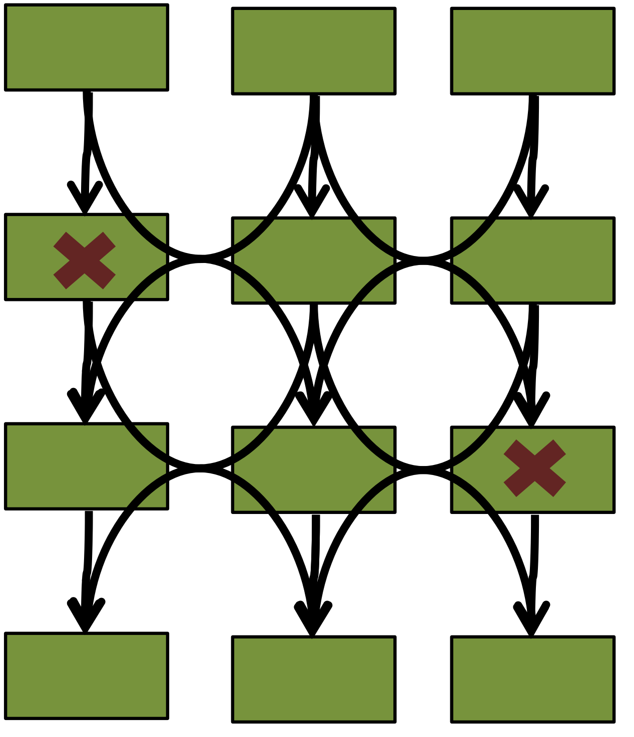


Heterogeneous CDs

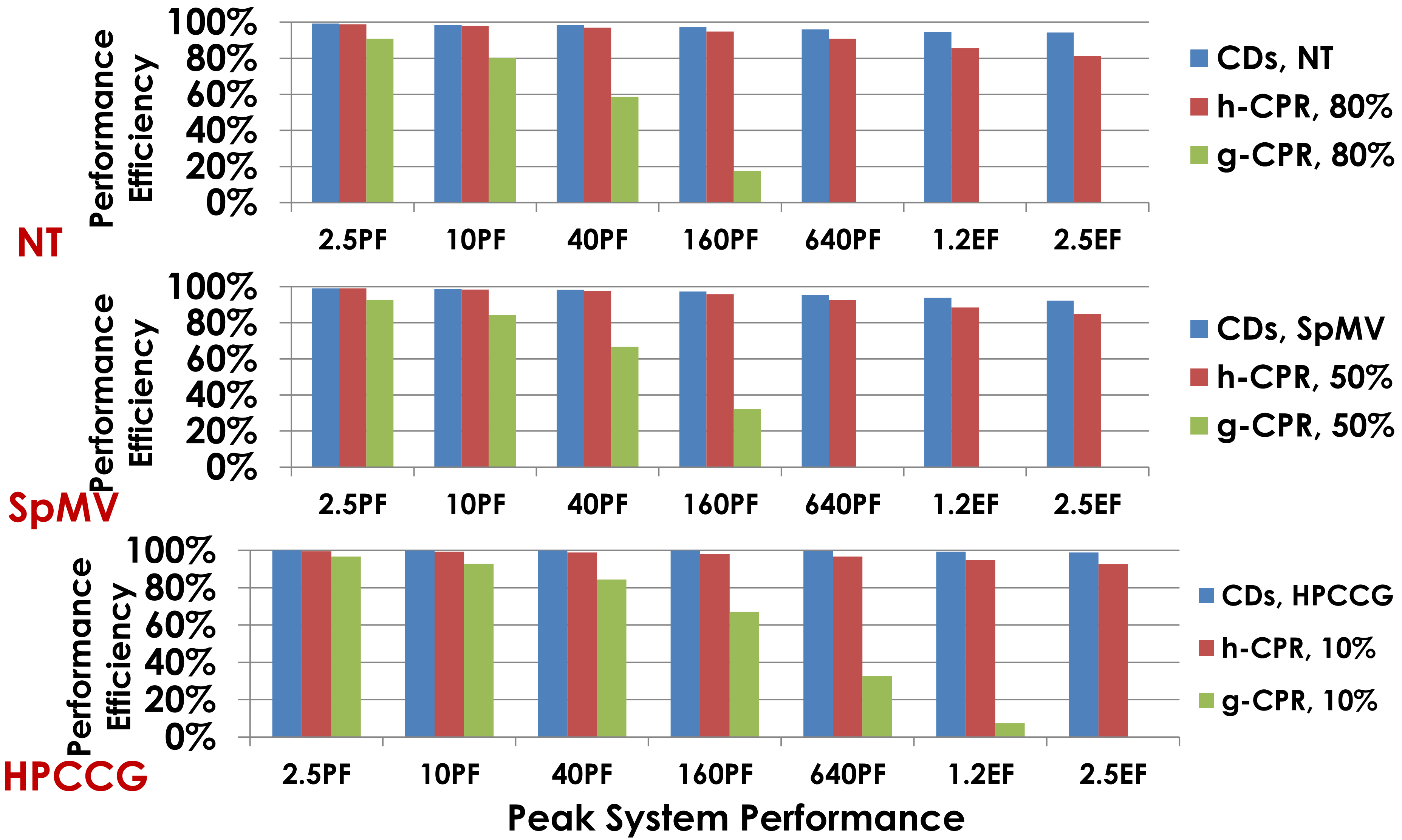
Asymmetric preservation and restoration

Analyzable?

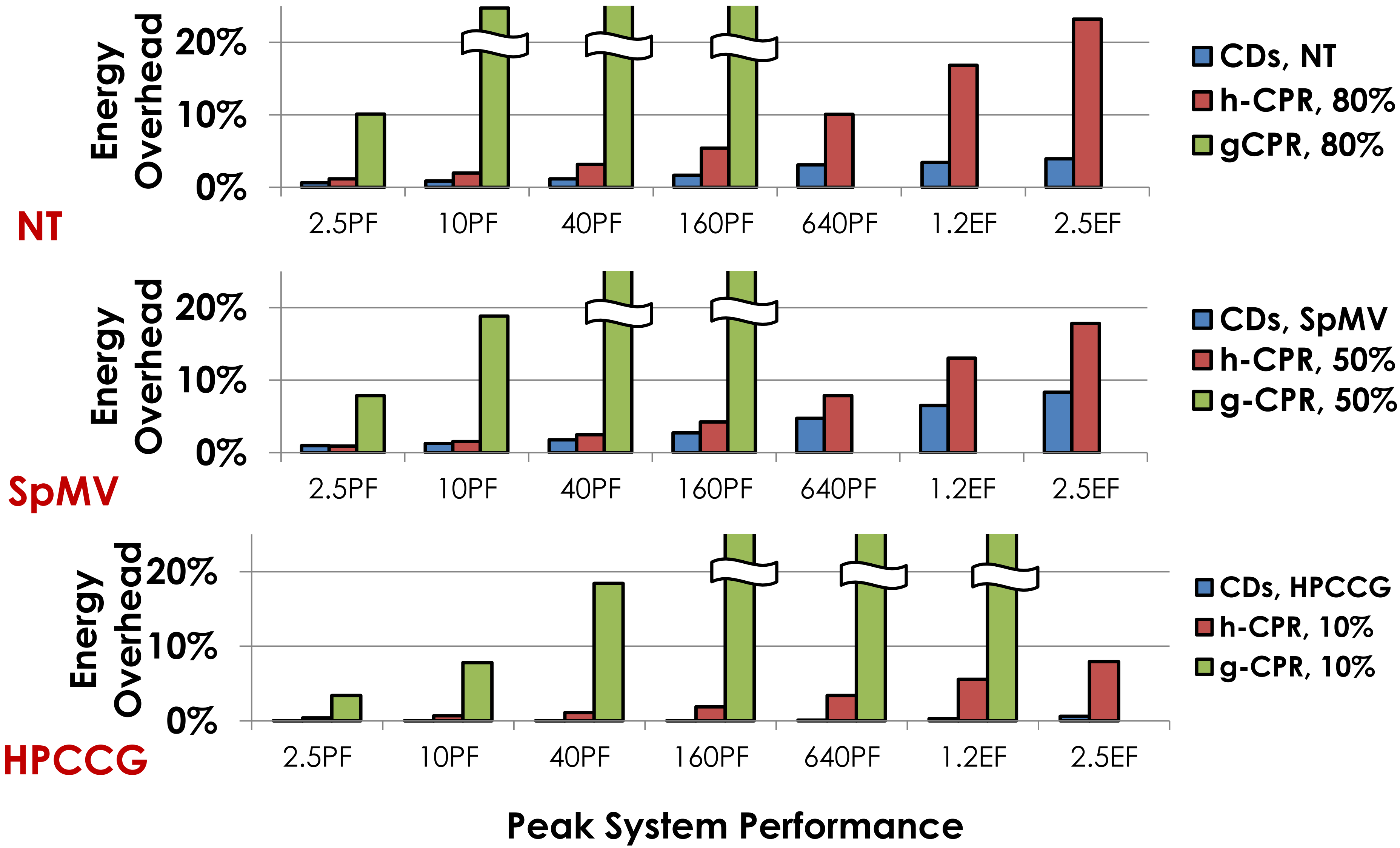




CDs are scalable



CDs are proportional and efficient



Architecture goals:

- Balance possible and practical
- Enable generality
- Don’t hurt the common case

It’s all about the algorithm

Architecture goals:

- Balance possible and practical
- Enable generality
- Don't hurt the common case

Architecture so far:

- Proportionality
  - Adaptivity
  - SW-HW co-tuning **+ analyzability**
  - Heterogeneity
- Locality
- Parallelism
- Hierarchy

Arithmetic
Control
Memory
Reliability
Programming

Exascale computers are **lunacy**

Exascale computers are lunacy **science**

Harbingers of things to come

Discovery awaits

Enable and promote **open innovation**

Math +
Systems +
Engineering +
Technology