Karlsruhe Institute of Technology

Institut für Technische Informatik
Chair for Embedded Systems – Prof. Dr. J. Henkel

Vorlesung im SS 2016

Reconfigurable and Adaptive Systems (RAS)

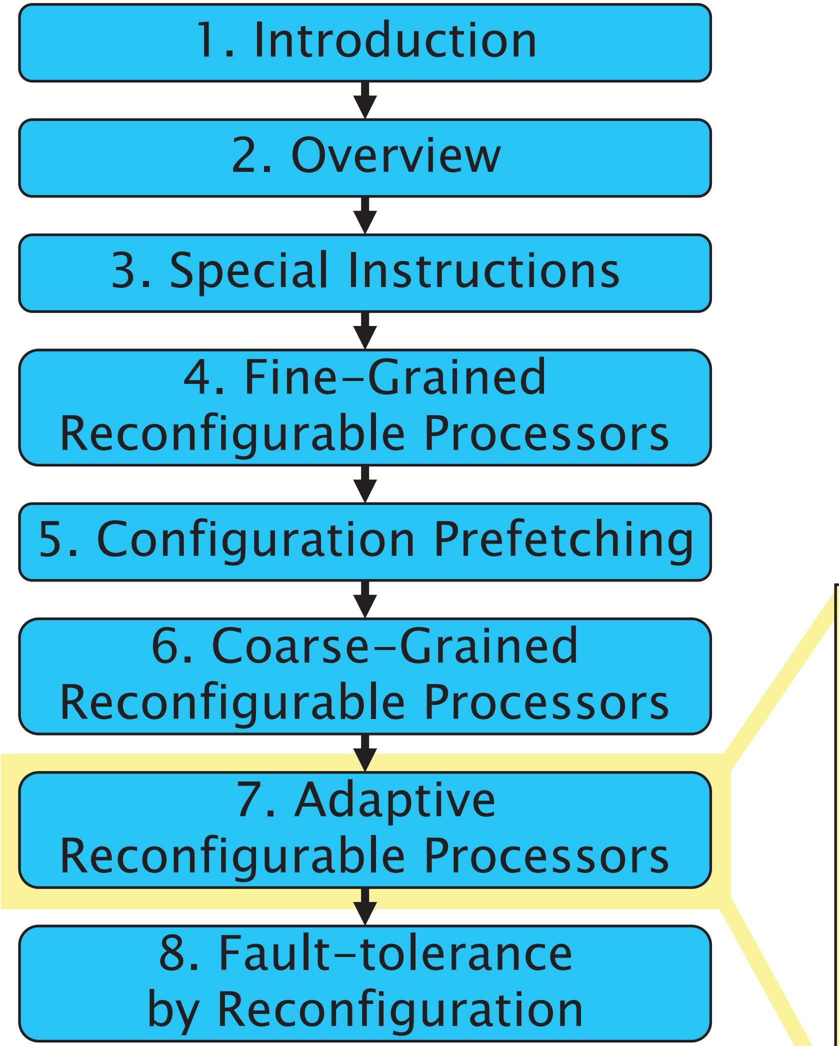
Marvin Damschen, Lars Bauer, Jörg Henkel

CES

Reconfigurable and Adaptive Systems (RAS)

7. Adaptive Reconfigurable Processors

RAS Topic Overview



1. Introduction

2. Overview

3. Special Instructions

4. Fine-Grained Reconfigurable Processors

5. Configuration Prefetching

6. Coarse-Grained Reconfigurable Processors

7. Adaptive Reconfigurable Processors

8. Fault-tolerance by Reconfiguration

- RISPP
- WARP
- Dynamic Instruction Merging (DIM)
- Further relevant architectures / domains

7.1 RISPP: Rotating Instruction Set Processing Platform

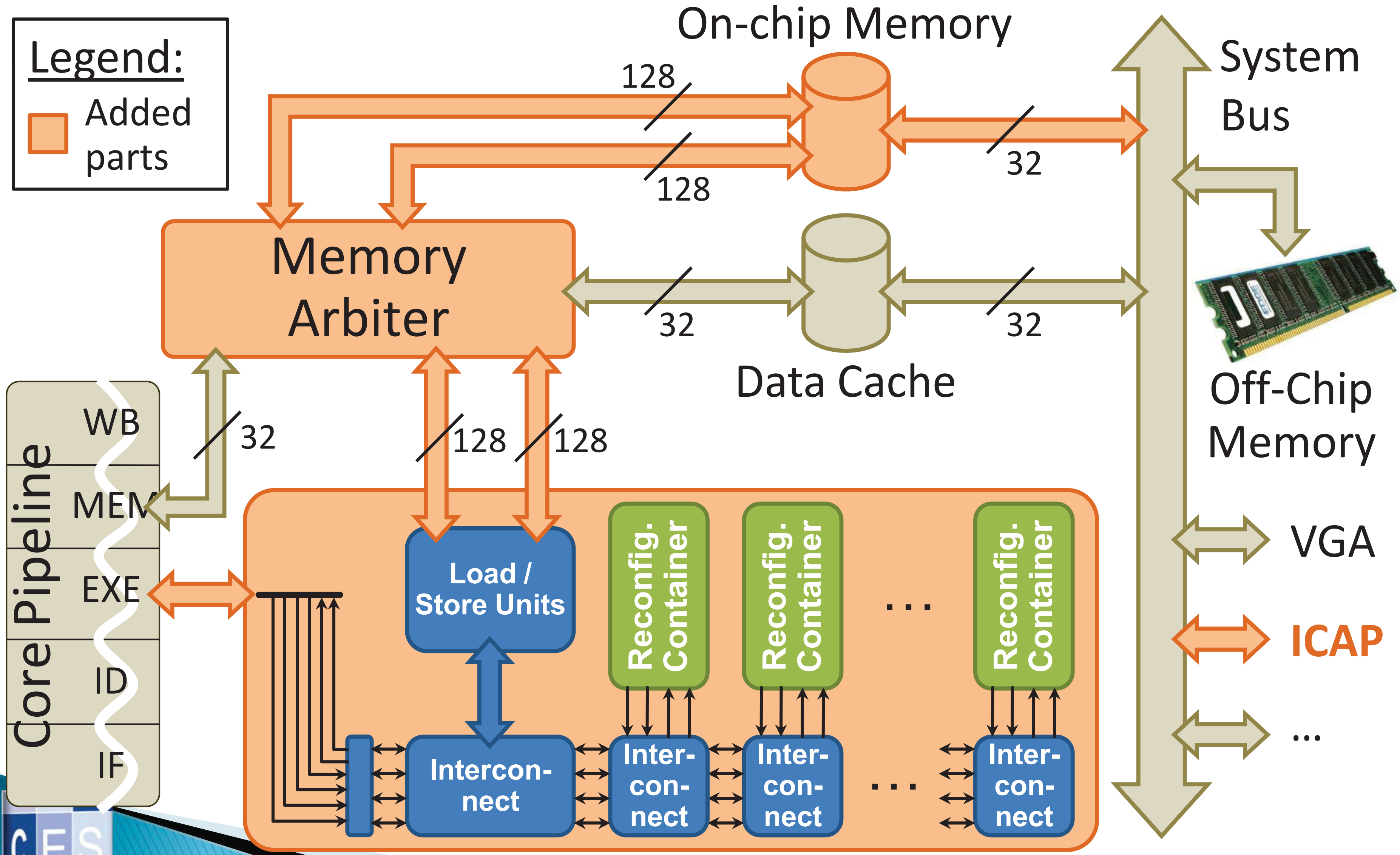
Overview

- Developed at CES, KIT
- Tightly-coupled fine-grained reconfigurable fabric
- Introduces and implements modular SIs
  - Provide different performance/area trade-offs at runtime
- Realizes high runtime adaptivity, i.e. a runtime system decides which reconfigurations shall be performed and when they shall be performed

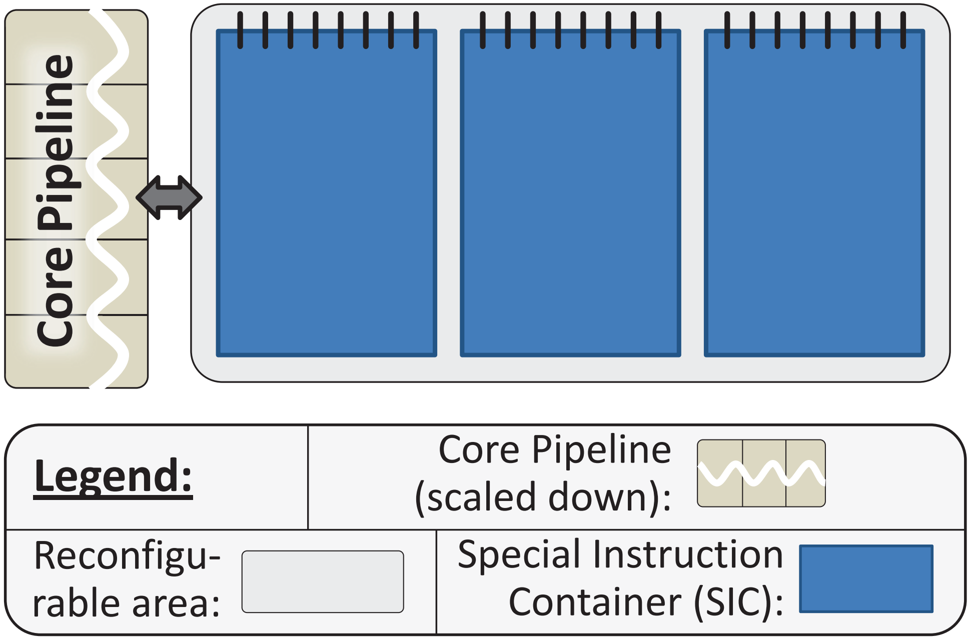
RISPP Recall

- Some parts were already introduced as case-study in previous lectures
- Instruction Format (up to 4 read and 2 write registers, immediate values, 10-bit virtual opcode)
- Using the core ISA (cISA) to implement SIs when their reconfiguration is not completed yet (trap handler)
- Special Instructions have access to main memory and to a fast on-chip scratch-pad memory
  - Using two independent 128-bit ports
  - Pipeline stalls when SI executes in hardware
- Dynamic Prefetching (called 'Forecasting') using weighted error-back propagation

RISPP HW Architecture Overview



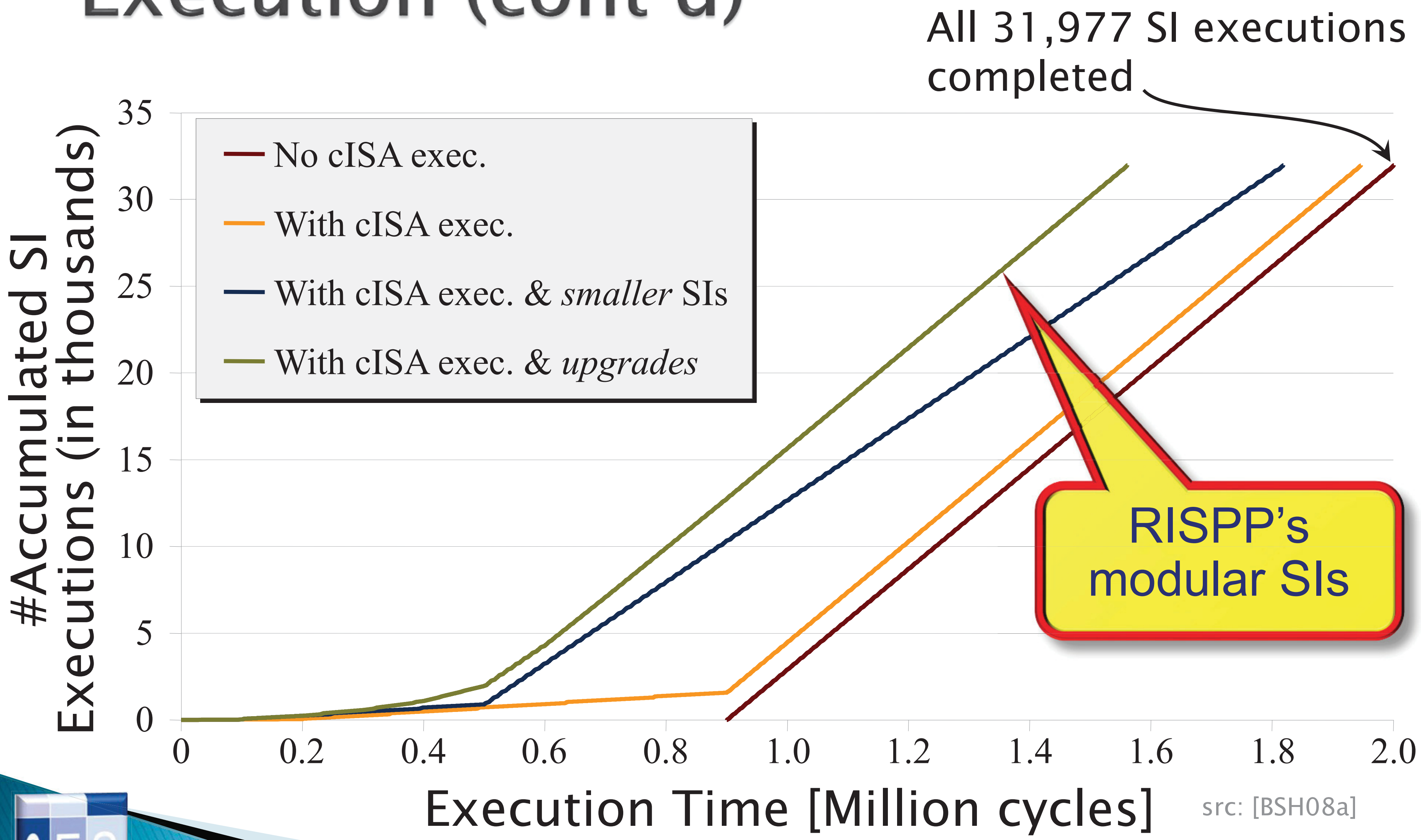
Analysis of Special Instruction Execution


Core Pipeline
Legend:
Core Pipeline (scaled down):
Reconfigu-rable area:
Special Instruction Container (SIC):

Corresponds to OneChip, Chimaera, Proteus, …

- Partition the reconfigurable fabric into so-called SI Containers
  - aka ‘Reconfigurable Functional Unit’
- An SI may be loaded into any free Container
- Problems:
  - Relatively long reconfiguration time
  - Limited Resource Sharing
  - Fragmentation (not the entire available space may be usable)

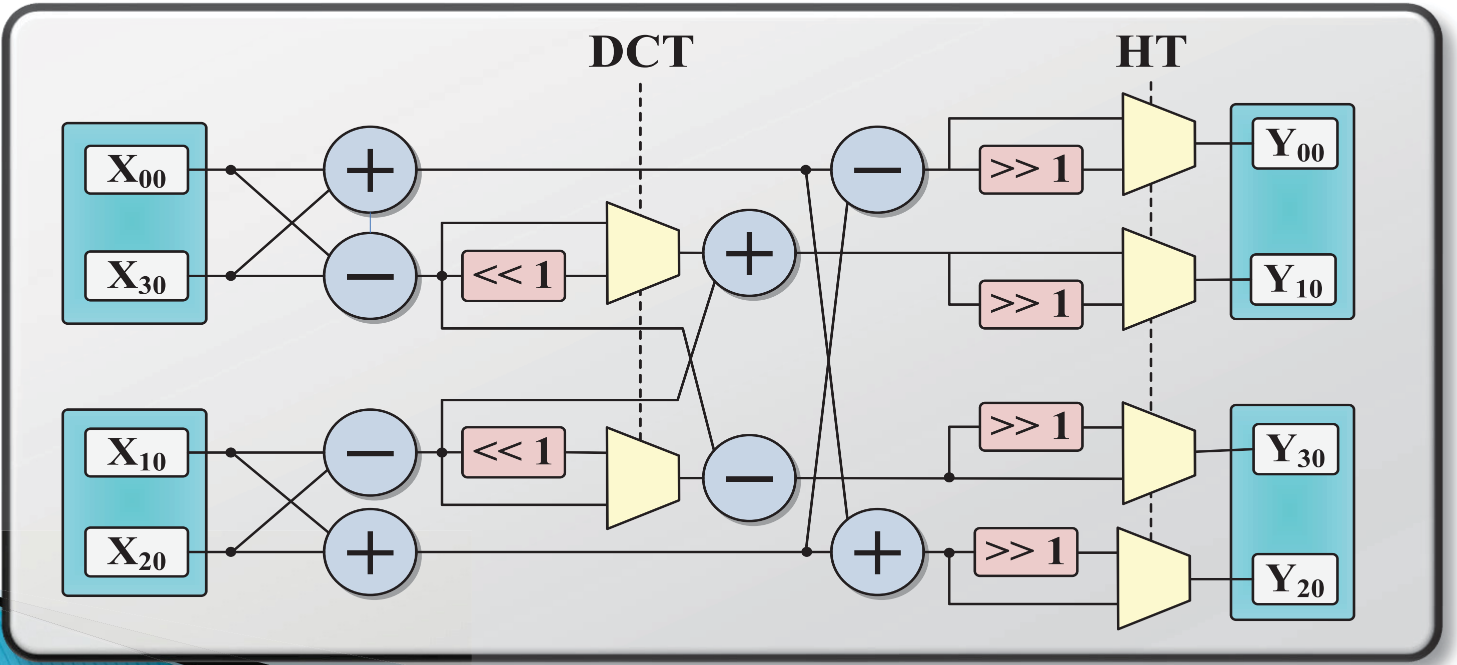
Analysis of Special Instruction Execution (cont'd)



src: [BSH08a]

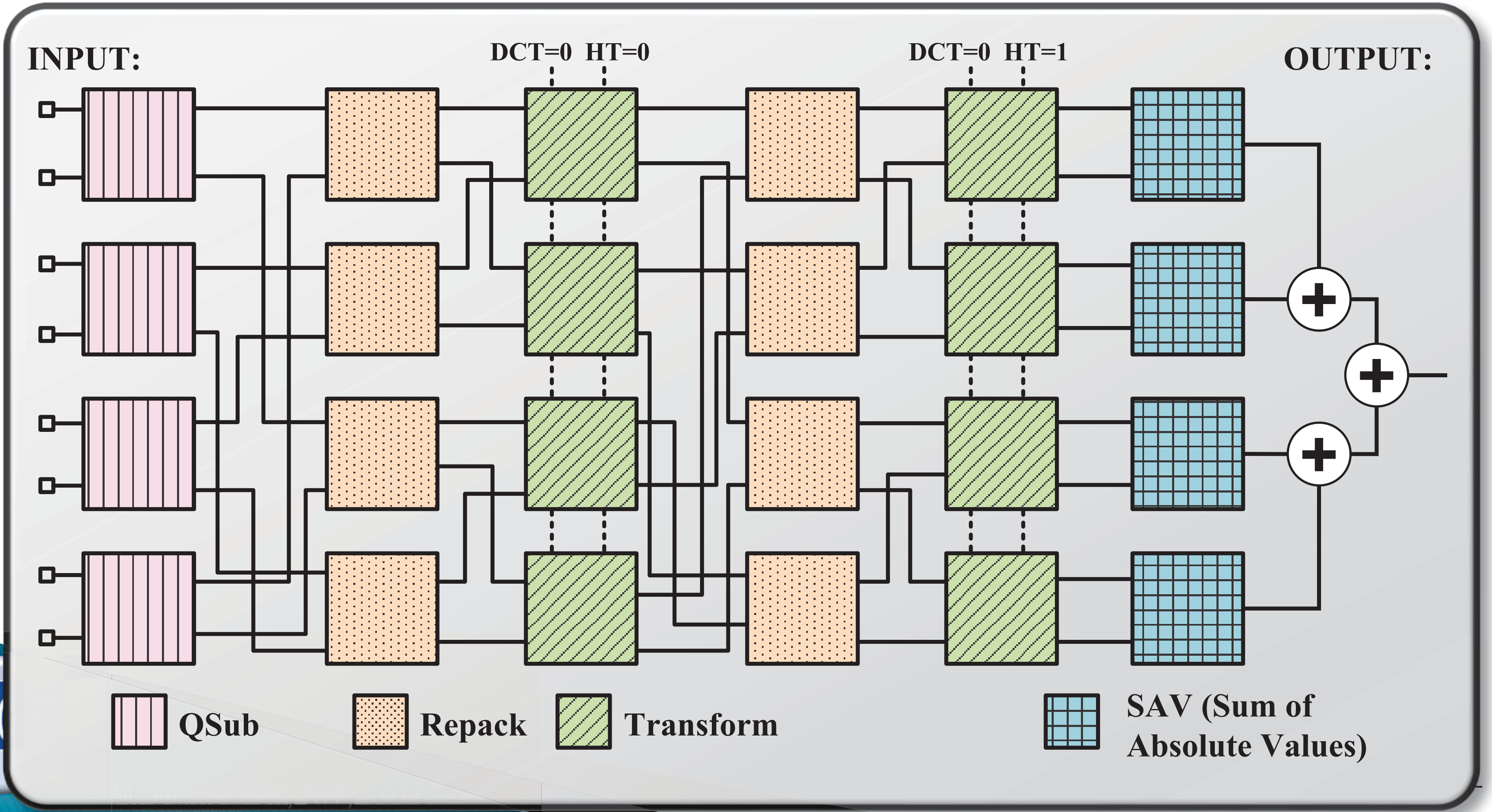
Fundamental Processor Extension: Atom / Molecule Model

- Definition Atom:
  - A computational data path
  - Smallest block that can be reconfigured (‘atomic’ in that sense)
- Example: Transform Atom



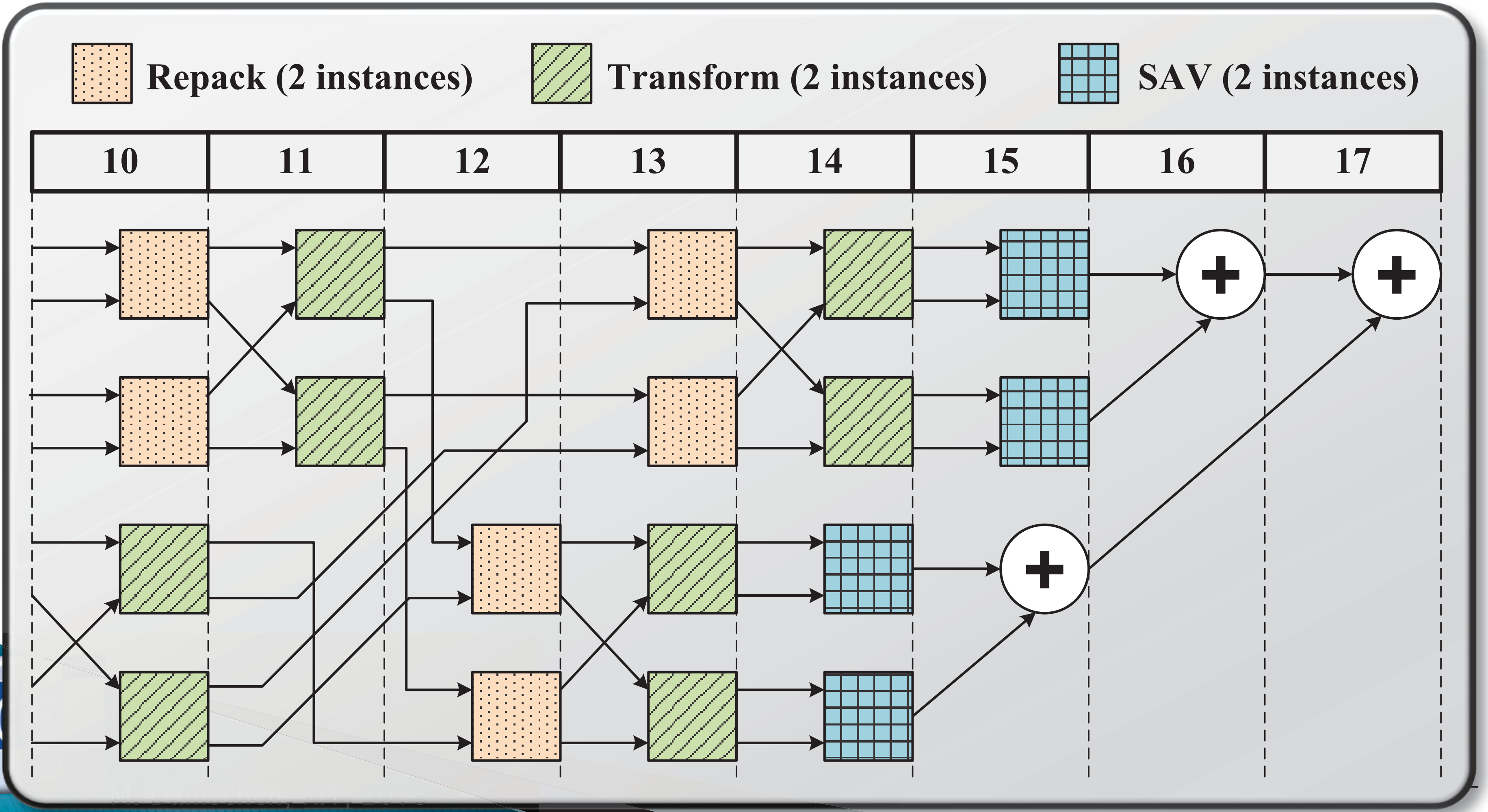
Fundamental Processor Extension: Atom / Molecule Model

- Definition Special Instruction:
  - An assembly instruction
  - Dataflow graph of Atoms
- Example: Sum of Absolute Transformed Differences (SATD)

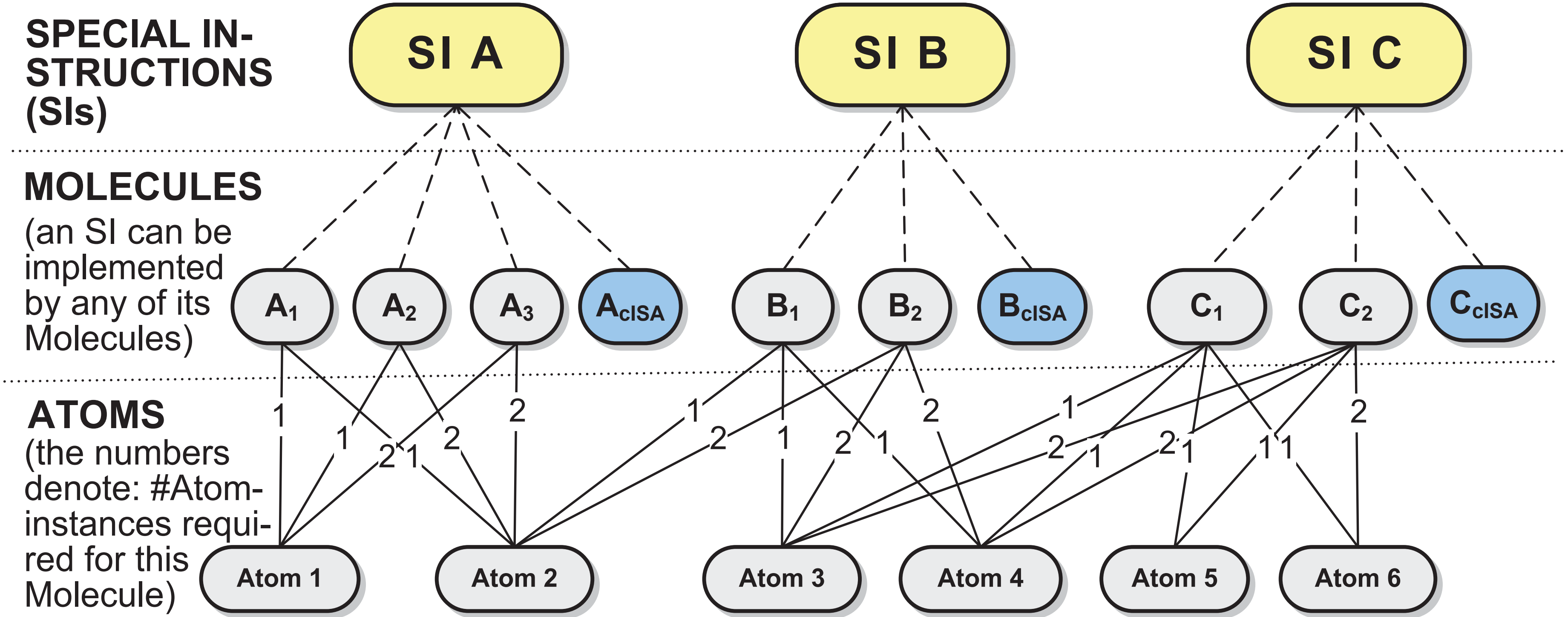


Fundamental Processor Extension: Atom / Molecule Model

- Definition Molecule:
  - Implementation of an SI
  - Using the available (i.e. at that time reconfigured) Atoms
  - Similar to HLS scheduling after allocating a certain number of Atoms



Fundamental Processor Extension: Atom / Molecule Model



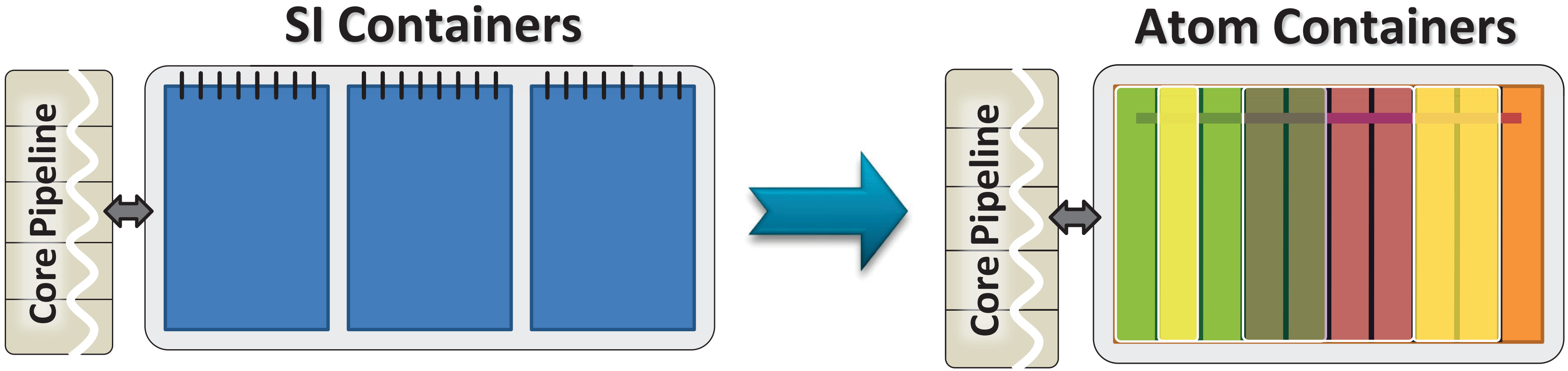
SPECIAL INSTRUCTIONS (SIs)

MOLECULES (an SI can be implemented by any of its Molecules)

ATOMS (the numbers denote: #Atom-instances required for this Molecule)

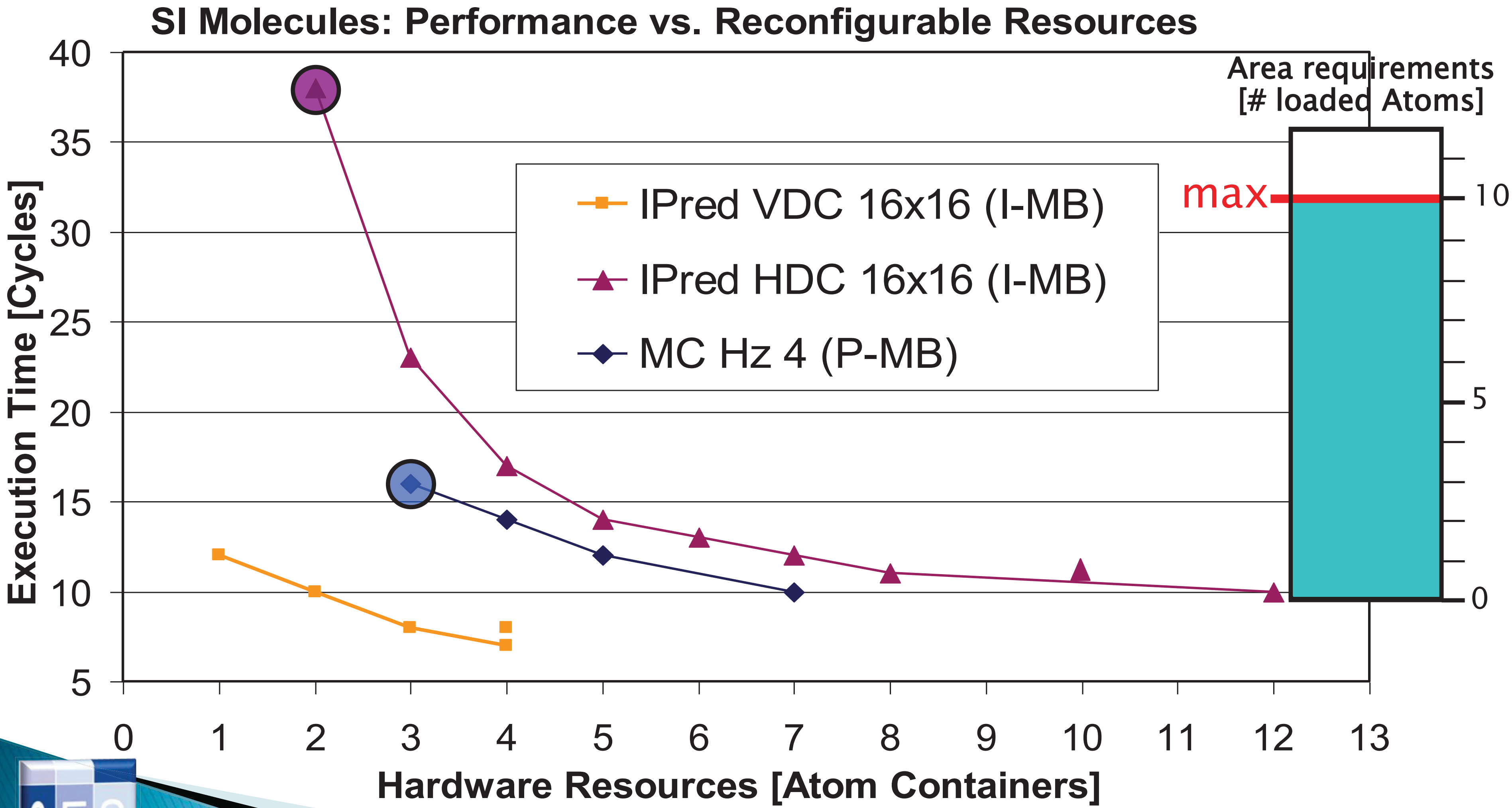
- For each SI there are different implementations (Molecules)
  - There is one Molecule that does not need any Atom (Software Implementation with core-ISA: cISA)
  - Atoms can be shared among different Molecules and SIs
- Implementation of a particular SI can be gradually upgraded by loading more Atoms

Difference to SI Containers



- Multiple SIs may share common Atoms
- There is no predetermined maximum of supported SIs
- But: it is not possible/easy to execute two SIs at the same time (as they are no longer independent)
  - Not necessarily a problem, see Molen (single controller unit) and OneChip (memory coherency problems)
- SIs can be *upgraded* (step-by-step by loading more Atoms)

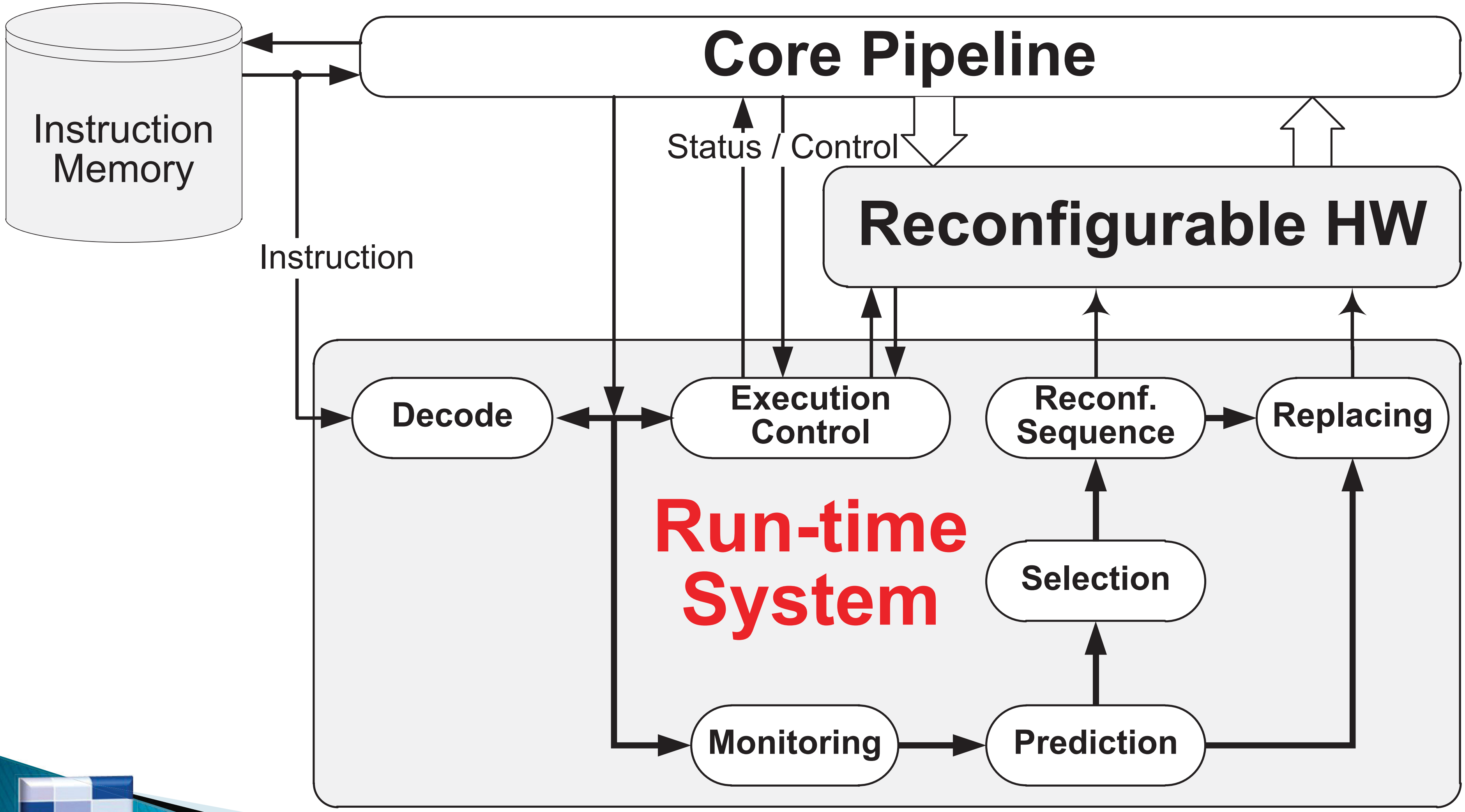
Adaptivity Through Dynamic Performance vs. Area Trade-off



Summary Modular SIs

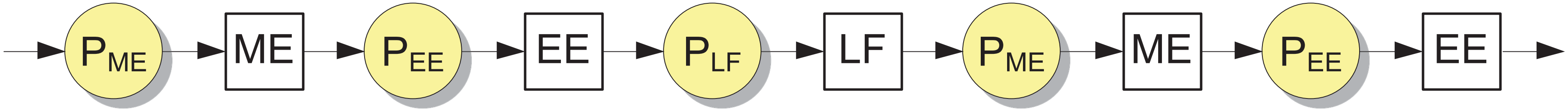
- Concept improves the efficiency and flexibility
  - Atom sharing
  - Reduced fragmentation
  - Reduced reconfiguration overhead (due to SI upgrading)
- Decision how many Atom Containers shall be spend for which SI can be adapted at runtime
- However, this adaptivity demands a runtime system that determines the decision and that implies overhead (to execute it)

Runtime System: Simplified Overview



Runtime System: Simplified Overview (cont'd)

- Decode: detects SIs and Forecasts (for prefetching) and sends them to the execution controls (only SIs) and Monitoring (SIs and Forecasts)
- Execution Control: executes SIs by determining their fastest currently available Molecule (state is maintained in a look-up table) and triggers the hardware execution (using the Atoms) or the software emulation (using the trap handler)
- Monitoring: Counts the executions for each SI
- Prediction: Fine-tunes the Forecasts (recall: dynamic prefetching; see below) and resets the monitoring values

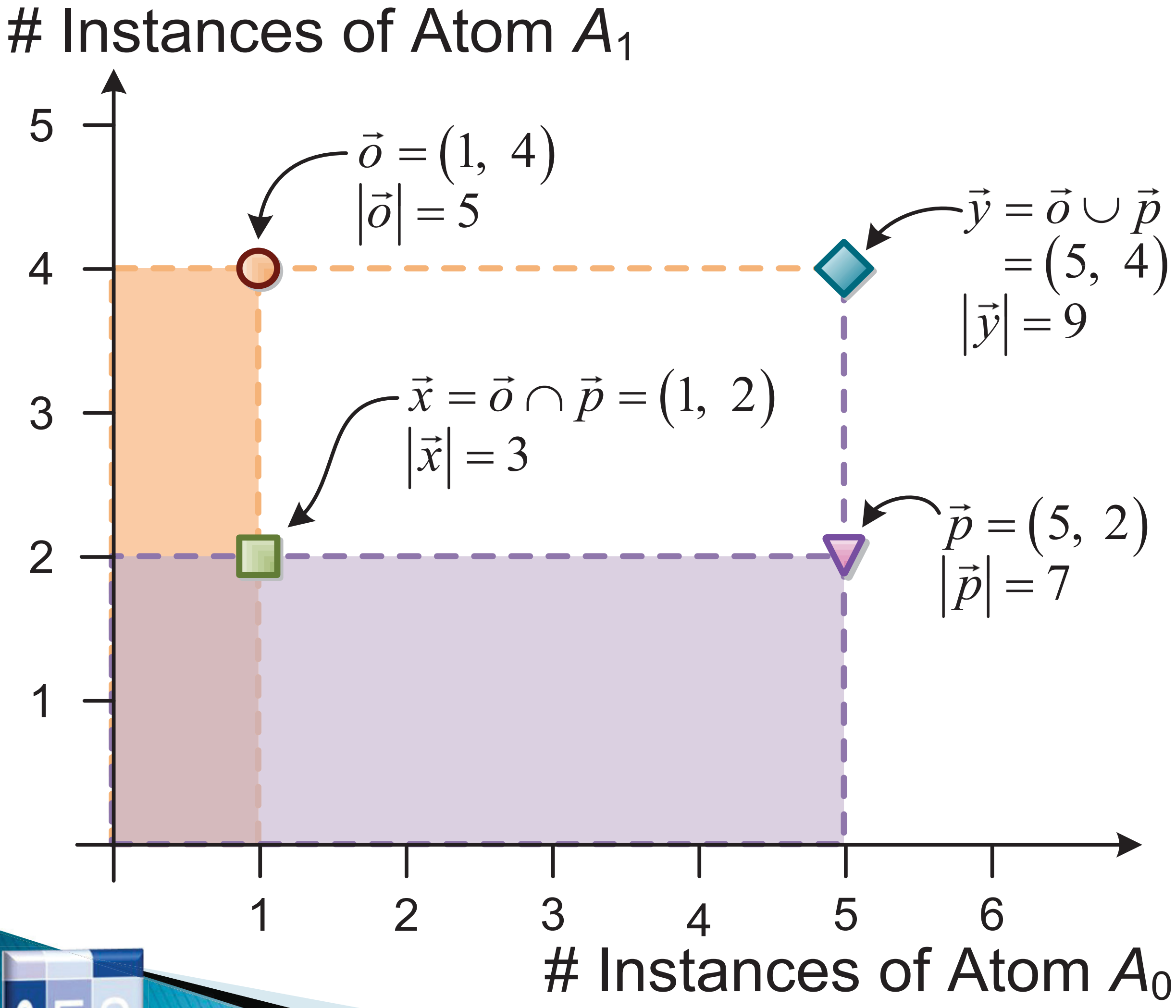


P: Prefetching Point
ME: Motion Estimation
EE: Encoding Engine
LF: Loop Filter

Runtime System: Simplified Overview (cont’d)

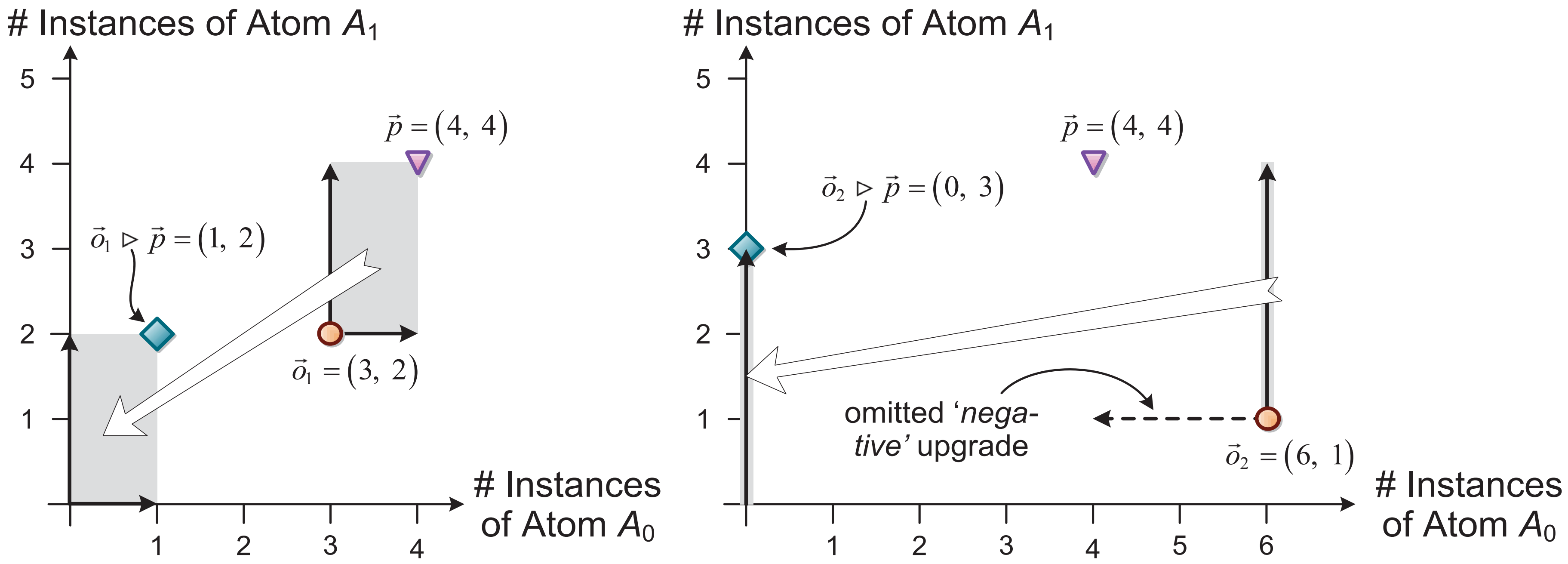
- Selection: Select Molecules to implement the forecasted SIs
- Reconfiguration Sequence Scheduling: Determine the reconfiguration sequence of the Atoms that are required to implement the selected Molecules
- Replacing: Determines, which currently configured Atom shall be replaced by a new Atom that is scheduled to be reconfigured

Formal Atom/Molecule Model



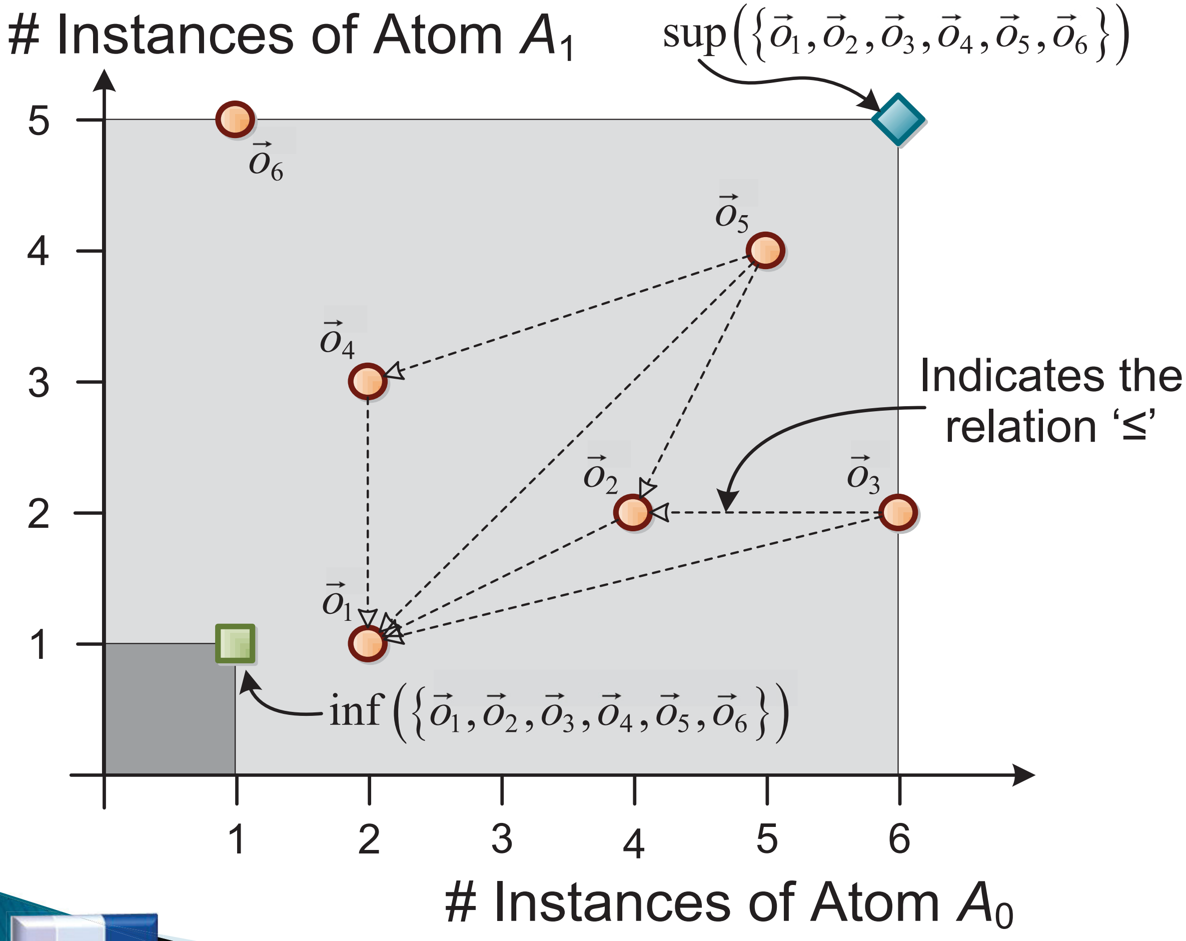
- Representing the Molecules as a vector of Atoms
  - The example only shows 2 Atom Types (A_0 and A_1), thus each vector has 2 entries; in general: $\mathbb{N}^n$
- Basic operators
  - How many Atoms are needed for a Molecule
  - Which Atoms have two Molecules in common
  - Which Atoms are needed to fulfill the demands of two Molecules

Formal Atom/Molecule Model (cont'd)



- Upgrade operator $o \triangleright p$:
  - Given the Atoms of o, which *additional* Atoms are needed to implement p
  - Similarly, the *without* operator: $p \,/\, o := o \triangleright p$

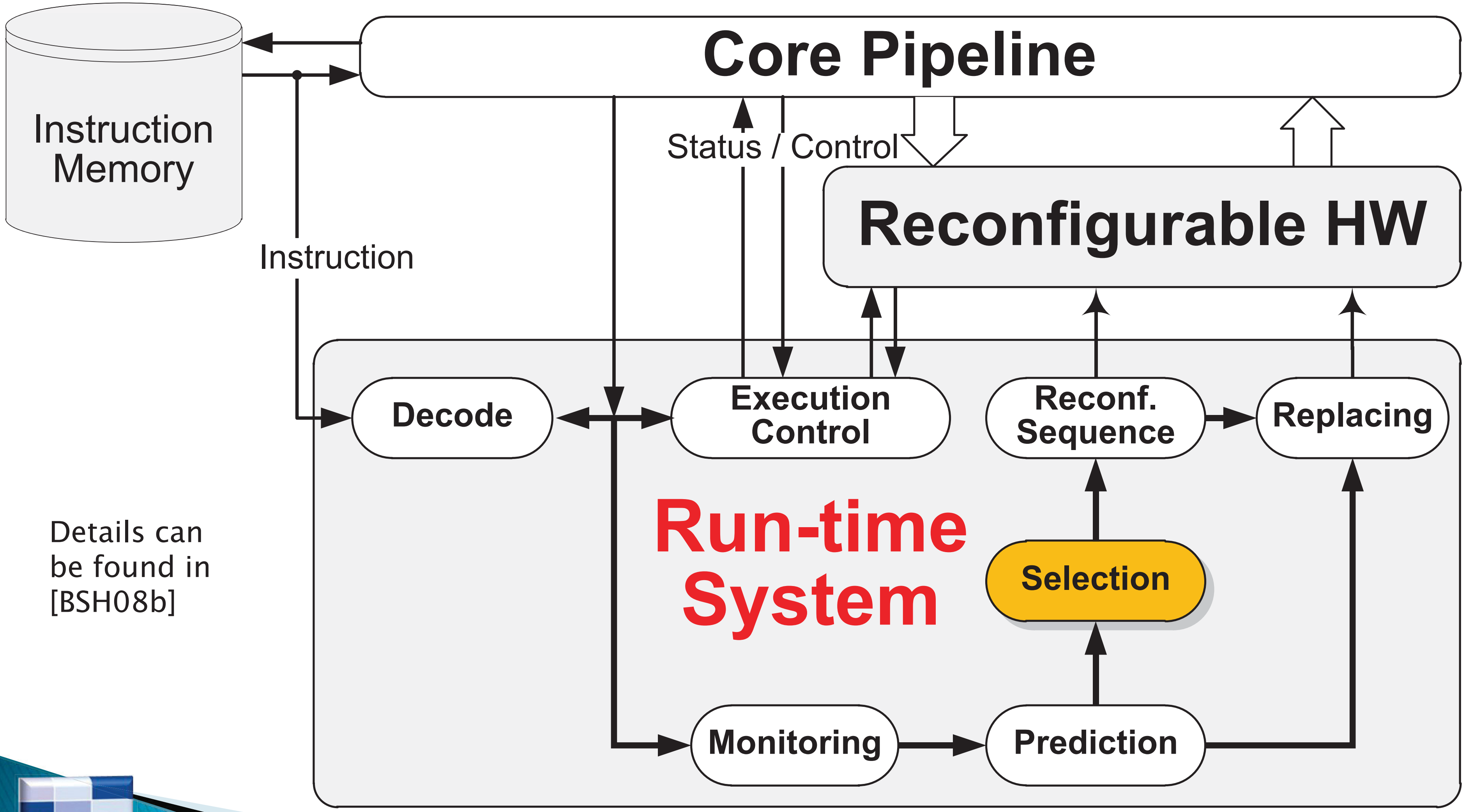
Formal Atom/Molecule Model (cont'd)



# Instances of Atom A_1
$\sup(\{\vec{o}_1,\vec{o}_2,\vec{o}_3,\vec{o}_4,\vec{o}_5,\vec{o}_6\})$
5
4
3
2
1
$\vec{o}_6$
$\vec{o}_5$
$\vec{o}_4$
$\vec{o}_2$
$\vec{o}_3$
$\vec{o}_1$
Indicates the relation '≤'
$\inf(\{\vec{o}_1,\vec{o}_2,\vec{o}_3,\vec{o}_4,\vec{o}_5,\vec{o}_6\})$
1
2
3
4
5
6
# Instances of Atom A_0

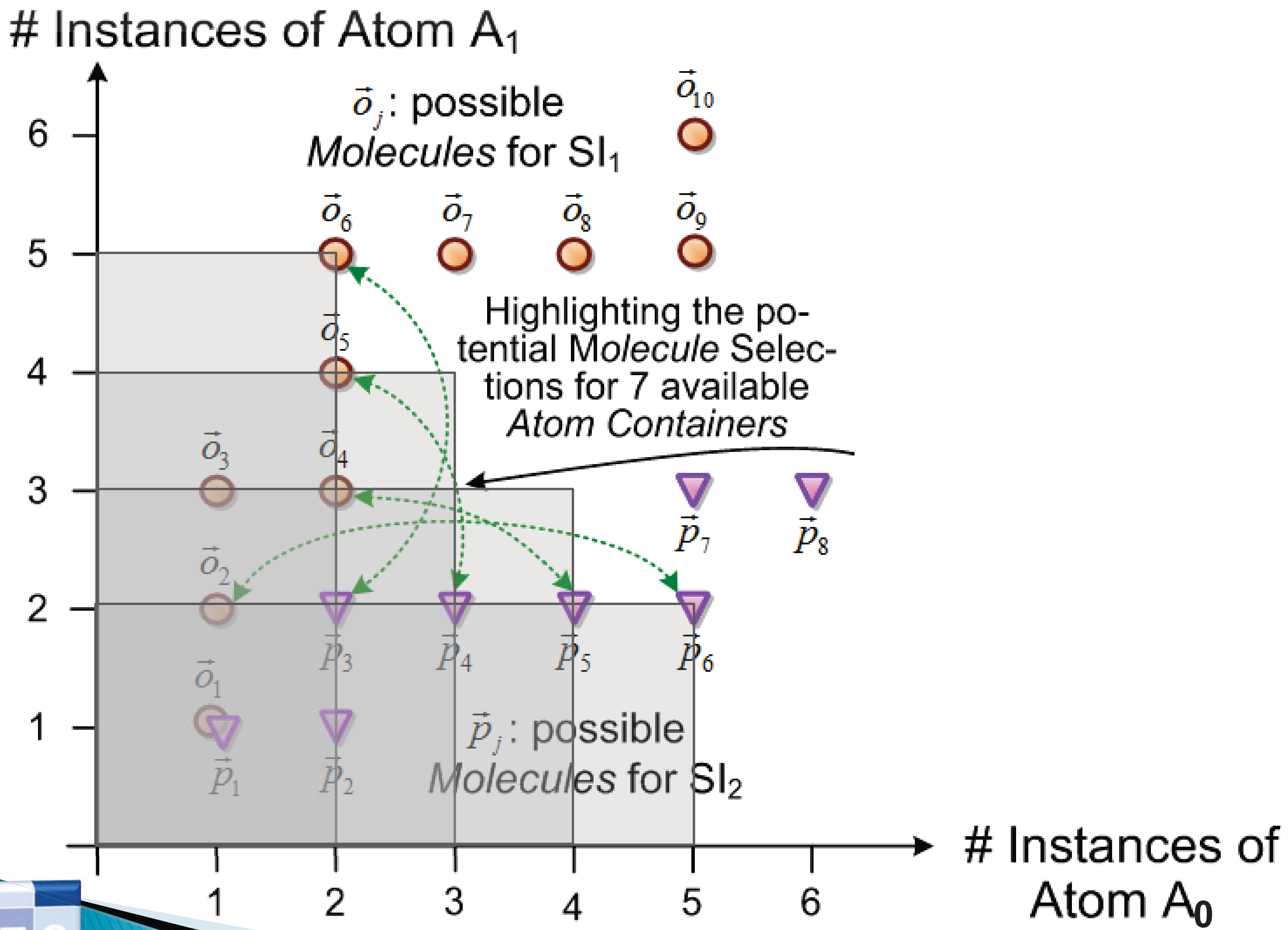
- A relation can be used to compare Molecules with each other
  - Not all Molecules can be compared, e.g. o_4 and o_6
- The relation has a infimum and a supremum
  - Actually it is a complete lattice (vollständiger Verband)

Runtime System: Simplified Overview



Details can be found in [BSH08b]

Molecule Selection: Why at runtime?



# Instances of Atom A_1
$\vec{o}_j$: possible *Molecules* for SI_1
$\vec{o}_1$
$\vec{o}_2$
$\vec{o}_3$
$\vec{o}_4$
$\vec{o}_5$
$\vec{o}_6$
$\vec{o}_7$
$\vec{o}_8$
$\vec{o}_9$
$\vec{o}_{10}$
Highlighting the potential *Molecule* Selections for 7 available *Atom Containers*
$\vec{p}_1$
$\vec{p}_2$
$\vec{p}_3$
$\vec{p}_4$
$\vec{p}_5$
$\vec{p}_6$
$\vec{p}_7$
$\vec{p}_8$
$\vec{p}_j$: possible *Molecules* for SI_2
1
2
3
4
5
6
# Instances of Atom A_0

Formalized Instruction Set Selection

- Input to the Selection: requested SIs and their different Molecules (in the following SI_i will denote one of the requested SIs)

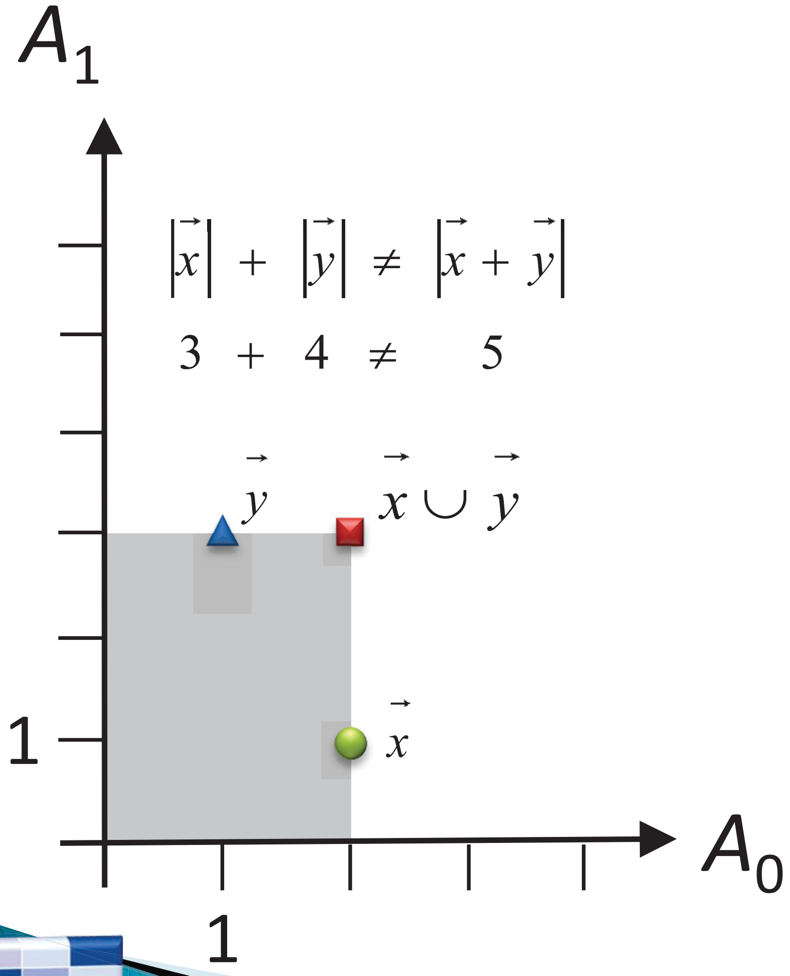
$$\underbrace{\{SI_A,\ SI_B,\ \ldots\}}_{\left\{\vec{b}_1,\ \vec{b}_2,\ \ldots,\ \vec{b}_{cISA}\right\}}$$

Complexity of the runtime Instruction Set Selection

- Similarities to the well-known NP-hard Knapsack Problem

- Given:
  - A Knapsack with the capacity C
  - Elements $E=\{e_i\}$ with weight $w(e_i)$ and profit $p(e_i)$
- Task: choose (multiple) elements such that the accumulated capacity is not violated and the accumulated benefit is maximal
  - Weight and benefit are constants that depend on the capacity (e.g. volume vs. weight) and the situation (e.g. for camping a tent might be more beneficial than a gold bar), respectively

Complexity of the runtime Instruction Set Selection (cont'd)



- Differences to Knapsack: the weight of a Molecule (i.e. the number of required Atoms to implement it) is not constant
  - It depends on the Molecules that are selected additionally and on their Atom requirements (due to Atom sharing between different SIs)
- Instead of accumulating the individual weights we have to combine all Implementations and determine their total weight
- Question: still NP-hard?

NP-hard Selection: Concept of proof

1. Take an arbitrary input of a Knapsack problem, i.e. capacity *C*, Elements e_i with $w(e_i)$ and $p(e_i)$
2. Apply a polynomial-time transformation on the input such that the transformed input describes a corresponding Selection problem
3. Solve the transformed input with an optimal solver for the Selection problem such that the result can be transformed back into the optimal solution for the original Knapsack problem
4. Then: 'Instruction-Set Selection' is at least as hard as 'Knapsack', i.e. Knapsack $\leq_p$ Instruction Set Selection

NP-hard Selection: Idea of proof

- The capacity of the Knapsack determines the number of Atom Containers, i.e. $N := C$
- For each Knapsack element e_i we create one Atom Type A_i
- For each Knapsack element e_i we create one Special Instr. SI_i with 2 Molecules

$$SI_i := \left\{\vec{x}_{i\_cISA}, \vec{x}_{i\_HW}\right\}$$

- The two Molecules represent the decision whether or not the element e_i should be packed into the knapsack
  - Not Packed: Molecule uses no Atoms and has zero profit
  - Packed: Molecule uses Atom Type A_i in a quantity that corresponds to the weight of the element; the Molecule profit corresponds to the element profit

$$\boxed{\begin{aligned} &\vec{x}_{i\_cISA} := (0, \ldots, 0) \\ &\left|\vec{x}_{i\_cISA}\right| = 0 \\ &p\left(\vec{x}_{i\_cISA}\right) := 0 \end{aligned}}$$

$$\boxed{\begin{aligned} &\vec{x}_{\_HW} := \left(0, \underbrace{\ldots, 0, w(e_i), 0, \ldots}_{\#\text{Instances of } A_i}, 0\right) \\ &\left|\vec{x}_{i\_HW}\right| = w(e_i) \\ &p\left(\vec{x}_{i\_HW}\right) := p(e_i) \end{aligned}}$$

NP-hard Selection: Idea of proof (cont'd)

- This SI structure avoids 'Atom sharing' (the main difference between Knapsack and Selection), as each Atom Type is only used by one Molecule
- The solver for the Instruction Set Selection will select one Molecule (cISA or Hardware) for each SI (i.e. element)
  - Selecting the cISA Molecule (with 0 profit and 0 weight) corresponds to not packing the corresponding element into the Knapsack
- Respecting the capacity constraints for the Atom Containers corresponds to respecting the capacity for the Knapsack
- Maximizing the profit for the SIs corresponds to maximizing the profit for the elements

→The optimal solution for the Instruction Set Selection corresponds to the optimal solution for the Knapsack

→Instruction Set Selection is NP-hard

Classical Greedy Implementation

- Instruction Set Selection needs to execute at runtime
  - Limited resources, e.g. memory and computing time
- Typical Heuristic for Knapsack problems: Greedy Algorithm

```
1. Calculate a benefit for each element (profit per weight)
2. Sort the benefits in a descending order
3. Initialize the Knapsack to be empty and its currently available
   space to its full initial capacity
4. Iterate over all sorted elements (starting with the highest
   benefit):
   IF     the element fits into the Knapsack (considering the
          still available space in it)
   THEN   greedily add it to the Knapsack and update its still
          available space
   ELSE   skip it (i.e. not selected) and continue with the next
          element
```

Greedy Implementation: Problems and Modifications

- This greedy approach cannot be directly used for Instruction Set Selection
  - Might choose multiple Molecules per SI
  - Presorting the Molecules does not work, because the weight (i.e. number of additionally required Atoms) changes, depending on which Molecules were previously selected (i.e. which Atoms are already selected)
- Modifications are required to use a greedy approach
  - After a Molecule was selected we remove the further Molecules from the same SI
  - Instead of presorting we have to recalculate the profit
  - Additionally, instead of using a ‘benefit’ (i.e. profit per weight) we can directly use our profit values, as they already contain the reconfiguration time (and thus indirectly the size in form of the additionally required Atoms) as parameter

Specialized Greedy Implementation

- At first, we remove all cISA Molecules: Instead of explicitly selecting them using the greedy algorithm they are afterwards added for each SI for which no hardware Molecule was selected
- Iterate in a loop over all Molecule candidates, calculate their profit, and remember the Molecule with the highest profit
  - Whenever a Molecule is too big (i.e. there are insufficient Atom Containers left to reconfigure its additionally required Atoms) then remove it from the candidate list
- Select the best Molecule Candidate and clean the remaining candidate list, i.e. remove those Molecules that implement the same SI
- Iterate, till the candidate list is empty

Greedy Implementation: Complexity

- Greedy Algorithm for Knapsack:
  - n := Total number of Molecules for all requested SIs
  - Computational complexity: O(n × log n) due to sorting
  - Additional memory: O(n) for storing the sorting result
- Greedy Algorithm for Instruction Set Selection:
  - Computational complexity: $O(n^2)$
    (in extreme case each SI has exactly 1 hardware Molecule and all of them together fit into the capacity
    → In each of the O(n) iterations the best Molecule is determined in O(n) and 1 Molecule is removed)
  - Additional memory: O(1) (to remember the best Molecule)
  - Advantage: After O(n) iterations the first Molecule is selected and reconfiguration may start. While reconfiguration is running, the next Molecules can be selected. So, even though the computational complexity is higher, the reaction time is shorter.

Optimization Goal

- Constraints describe a 'valid' selection;
  what should be considered for a 'good' selection?
- Execution frequency f_i of *SI*$_i$ (more often executed SIs are more 'important')
- Performance improvement of a Molecule in comparison to the cISA performance
  - Note: $\vec{x}_{ij}$ denotes the j^{th} Molecule from *SI*$_i$

$$\boxed{\begin{array}{l}\vec{x}_{i\_cISA}.getLatency() \\ -\ \vec{x}_{ij}.getLatency()\end{array}}$$

- Reconfiguration time of the Molecules
  - Considering 'how long' the reconfiguration lasts and 'when' the SI is needed (i.e. executed) the first time
- Potentially more parameters, but the above parameters turned out to be the most important ones

Optimization Goal (cont'd)

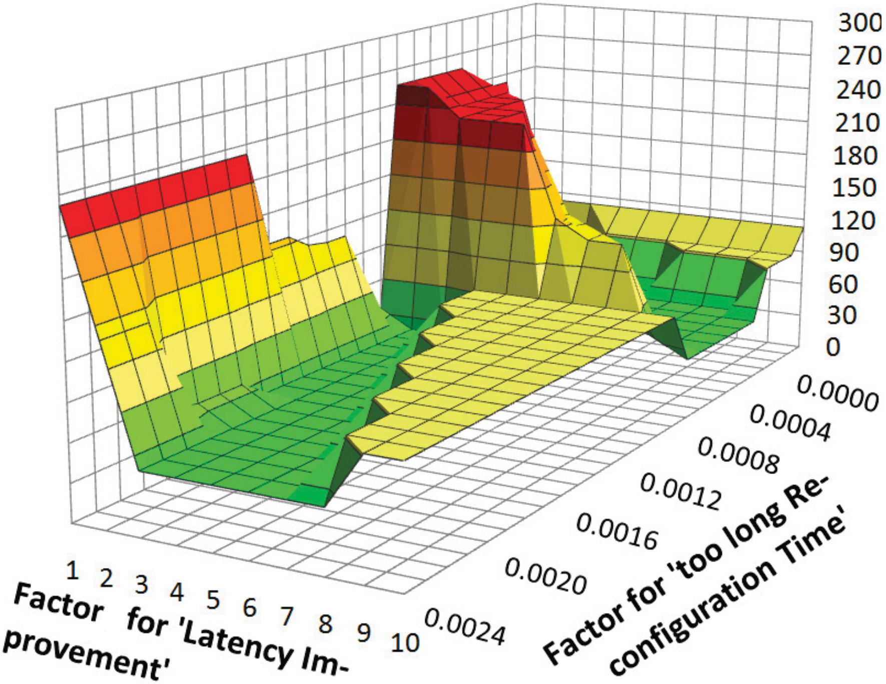
$$p\left(\vec{x}_{ij}\right) := f_i \cdot \left(\begin{array}{l} L \cdot \left(\begin{array}{l} \vec{x}_{i\_cISA}.getLatency() \\ - \vec{x}_{ij}.getLatency() \end{array} \right) - \\ R \cdot \max\left(0, \begin{array}{l} t_{reconf}\left(\vec{x}_{ij}\right) - \\ t_{firstExec}\left(\vec{x}_{ij}.getSI()\right) \end{array} \right) \end{array} \right)$$

- Selection factors *L* and *R* are used to scale the parameters
  - *L*: Latency Improvement
  - *R*: Too long reconfiguration time

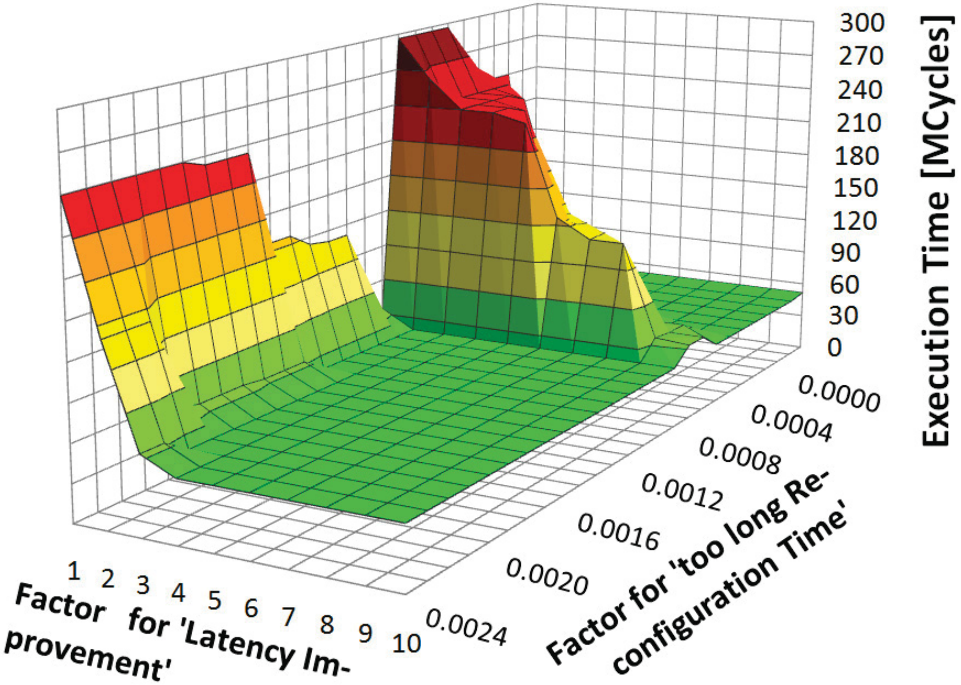
Comparing Greedy vs. Optimal

Capacity:
5 Atom Container

Greedy:

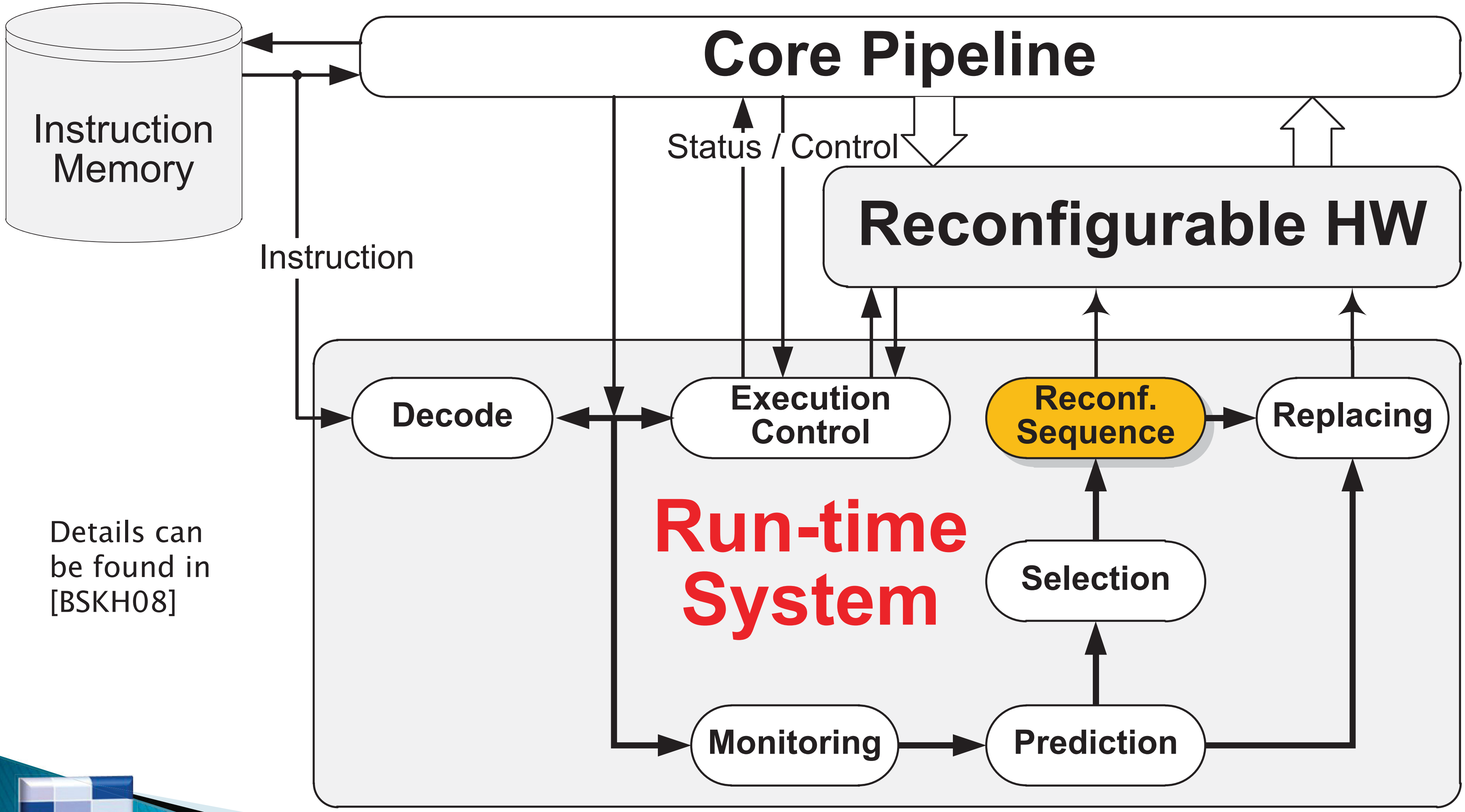


Optimal:



- For many parameter pairs, Greedy finds the same solution
- In some (not relevant) cases, Greedy finds a solution that leads to a faster execution time → Note: optimally solving Selection does not necessarily lead to the fastest execution time (e.g. due to sub-optimal prediction/forecasting/scheduling/replacement etc.)

Runtime System: Simplified Overview



Details can be found in [BSKH08]

Determining Atom loading sequence

- After Selection, we have a set of Molecules that shall be reconfigured

$$S = \{\vec{x}_i\}$$

- Altogether we need a certain set of Atoms to realize all Molecules in this set (supremum)

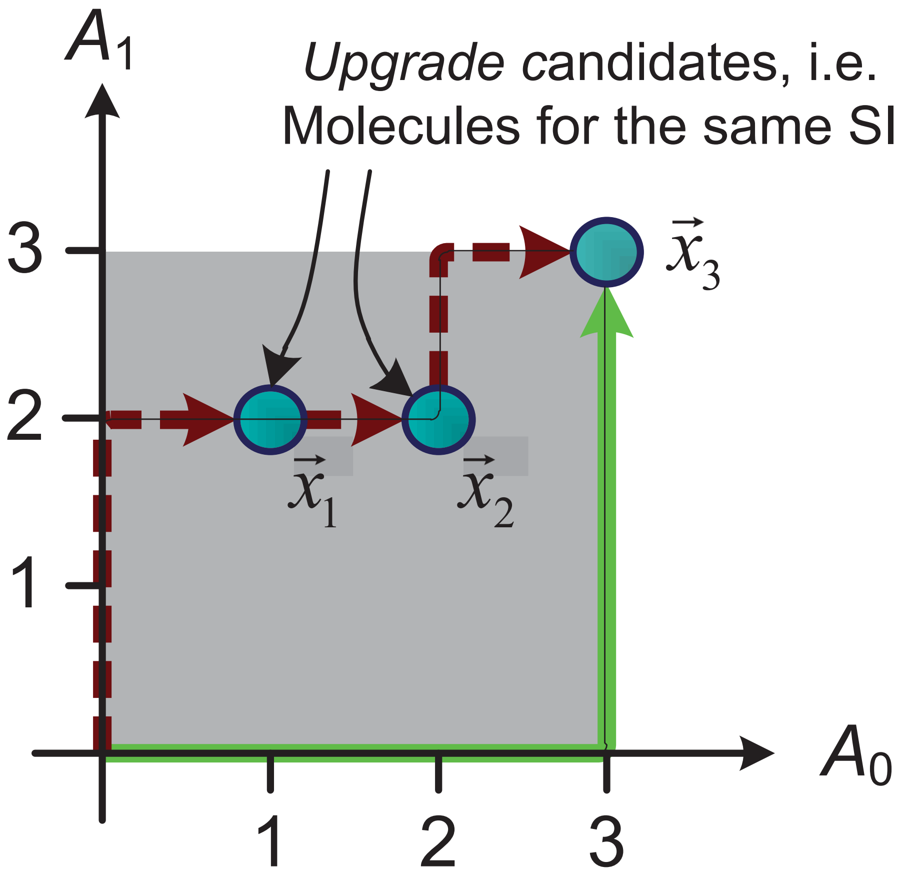
$$\sup(S) = \bigcup_{\forall \vec{x} \in S} \vec{x}$$

- Initially, some Atoms may already be available in hardware and we only need to reconfigure the remaining Atoms

$$\vec{a} \triangleright \sup(S)$$

- Problem: The reconfiguration is rather slow and we have to perform one reconfiguration after the other
- Question: in which sequence shall the reconfigurations be performed?

Determining Atom loading sequence (cont’d)

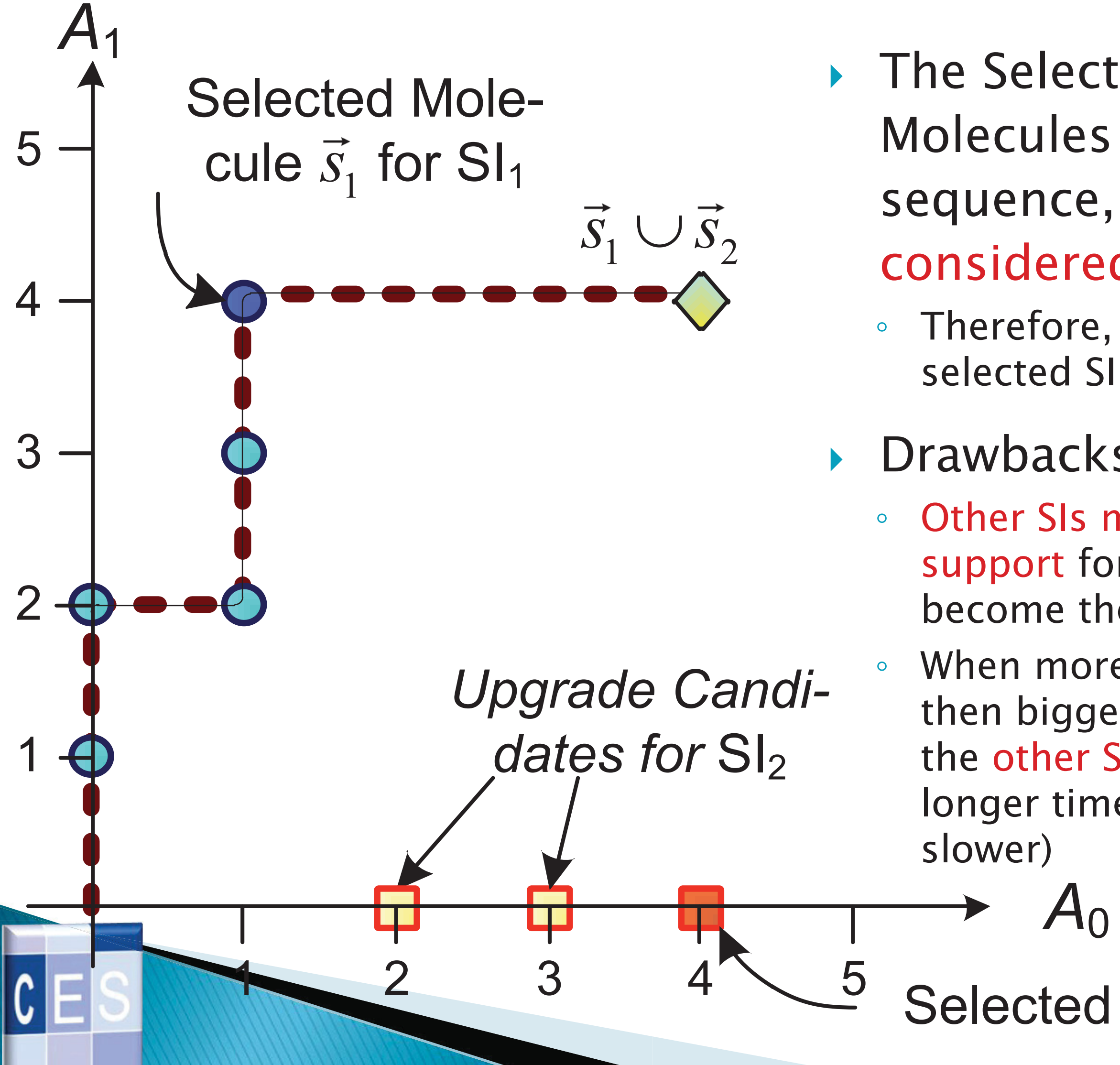


Upgrade candidates, i.e. Molecules for the same SI

# loaded Atoms	fastest available Molecule	
	⇢	→
1	—	—
2	—	—
3	$\vec{x}_1$	—
4	$\vec{x}_2$	—
5	$\vec{x}_2$	$\vec{x}_2$
6	$\vec{x}_3$	$\vec{x}_3$

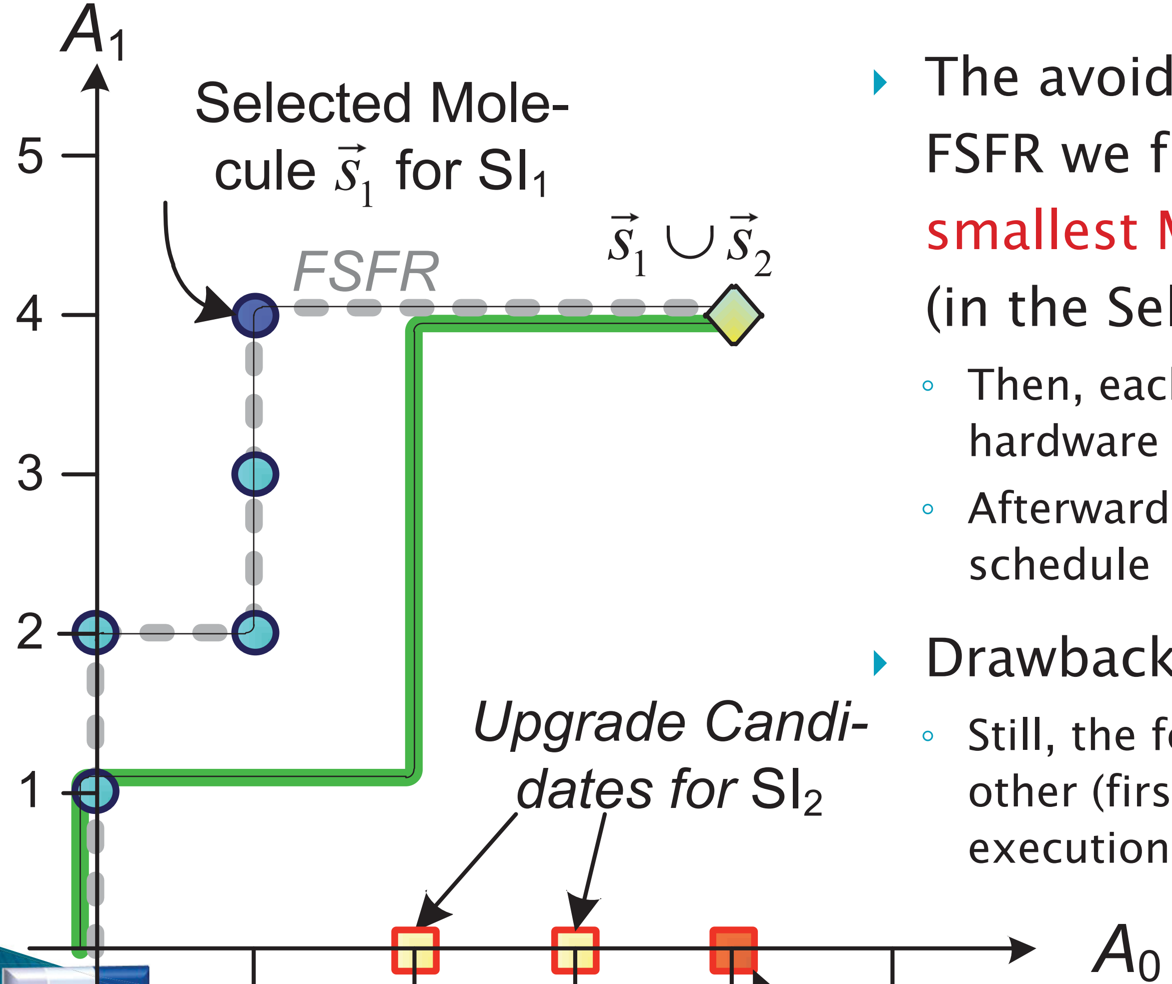
- **Note:** typically the starting point (here: (0,0)) and the ending point (here: (3,3)) vary between different Selections/Schedules

Scheduling Molecules
FSFR: First Select First Reconfigure



- The Selection determines the Molecules of the SIs in a certain sequence, i.e. more relevant SIs are considered first
  - Therefore, the Molecules of the first selected SI should be reconfigured first
- Drawbacks:
  - Other SIs may not achieve any hardware support for a noticeable time and therefore become the major bottleneck
  - When more Atom Containers are available then bigger Molecules will be selected and the other SIs are not accelerated for a longer time (overall exec. might become slower)

Scheduling Implementations ASF: Avoid Software First

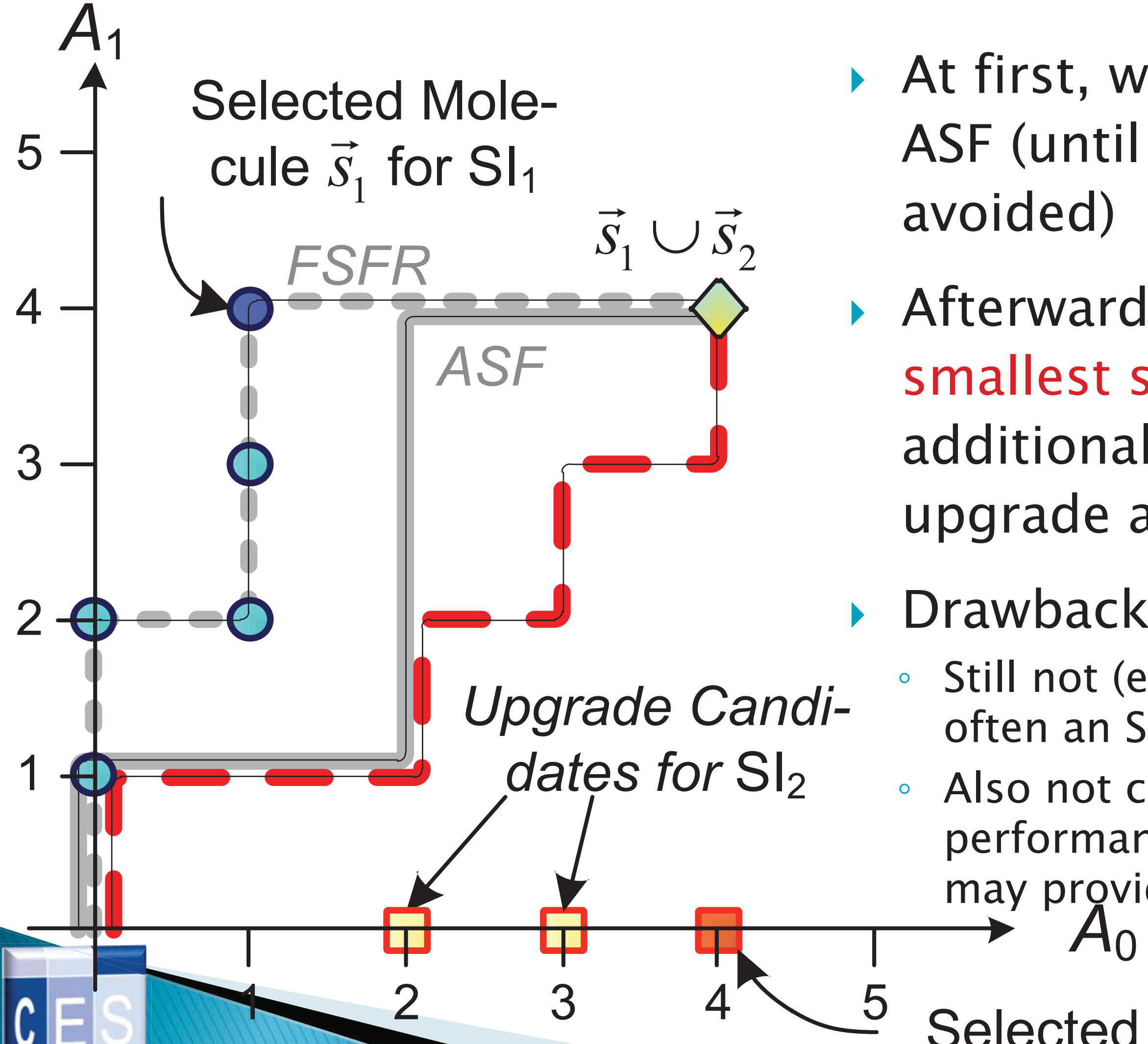


A_1
5
4
3
2
1
Selected Molecule $\vec{s}_1$ for SI_1
FSFR
$\vec{s}_1 \cup \vec{s}_2$
Upgrade Candidates for SI_2
1
2
3
4
5
A_0

Selected Molecule $\vec{s}_2$ for SI_2

- The avoid the drawbacks from FSFR we first schedule the smallest Molecule from each SI (in the Selection sequence)
  - Then, each SI has some degree of hardware acceleration
  - Afterwards we follow the FSFR schedule
- Drawbacks:
  - Still, the focus is on one SI after the other (first for avoiding cISA execution, afterwards for upgrading)

Scheduling Implementations
SJF: Smallest Job First



- At first, we follow the path from ASF (until all cISA executions are avoided)
- Afterwards, we determine the smallest step (i.e. number of additionally required Atoms) to upgrade an SI
- Drawbacks
  - Still not (explicitly) considering how often an SI is expected to execute
  - Also not considering how much performance benefit a certain upgrade may provide

Selected Molecule $\vec{s}_2$ for SI_2

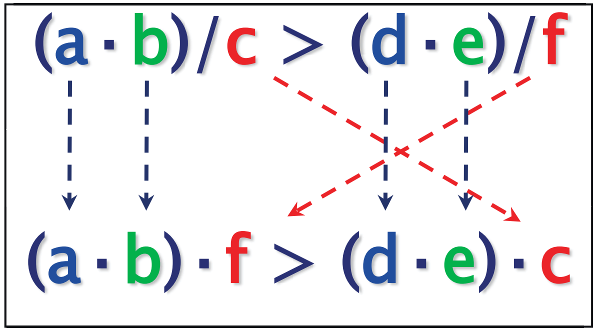
Scheduling Implementations
HEF: Highest Efficiency First

- For determining the next Molecule that shall be scheduled consider the following parameters for a scheduling candidate $\vec{c}$:
  - How often is the corresponding SI executed: f_i
  - What is the performance improvement (in cycles per execution) compared to the currently fastest available Molecule (i.e. after the already scheduled reconfigurations are completed)
  - How many additional Atoms are required (Note: ‘additional’ should never be zero; Molecules with 0 additional Atoms are removed)
- Calculating the ‘efficiency’:

$$f_i \cdot \left(\frac{\vec{c}.getSI().getFastestAvailableMolecules(\vec{a}).getLatency() - \vec{c}.getLatency()}{\left| \vec{a} \triangleright \vec{c} \right|} \right)$$

Scheduling Implementations HEF: Highest Efficiency First

- Calculating the ‘efficiency’ requires a division
  - Divisions require many cycles when executed in software or large area when implemented in hardware
- Optimized calculation:
  - The actual value of the ‘efficiency’ is not required, only the Molecule with the best (biggest) efficiency needs to be determined
  - Thus, only comparison between two values is required

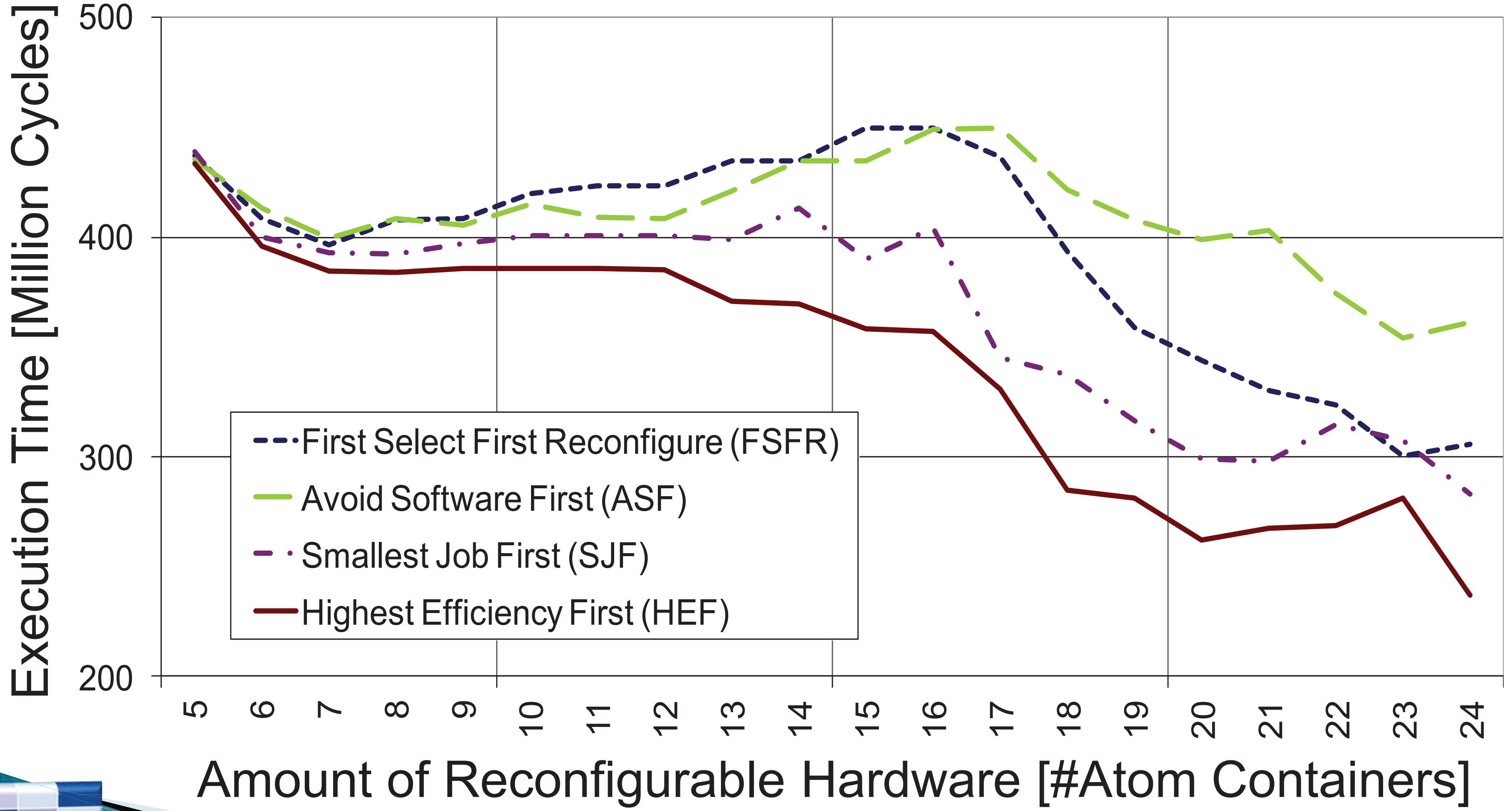


$$(a \cdot b)/c > (d \cdot e)/f$$

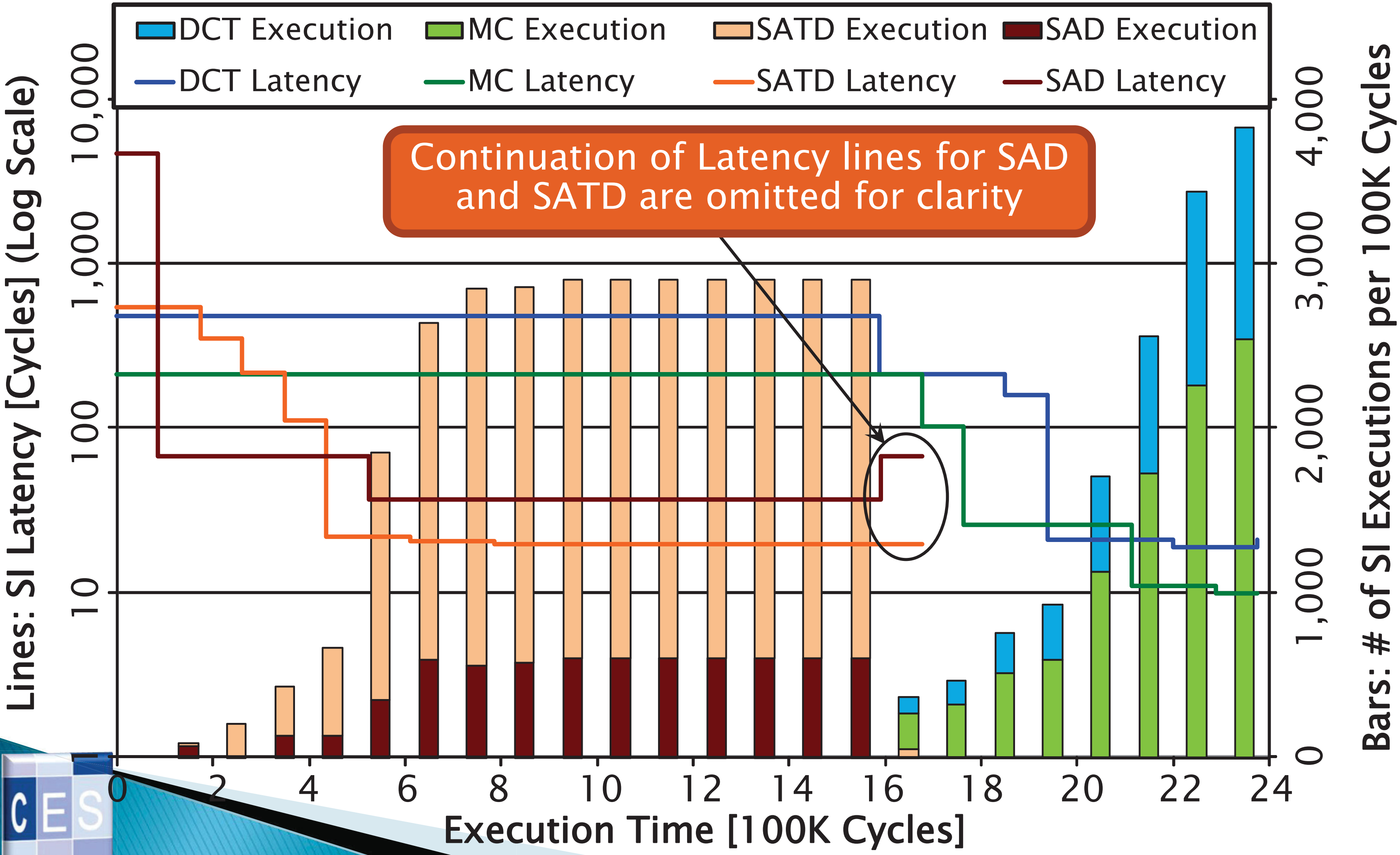
$$(a \cdot b) \cdot f > (d \cdot e) \cdot c$$

  - Store $a \cdot b$ separately to reuse it for the comparisons

Comparing the different Scheduling schemes

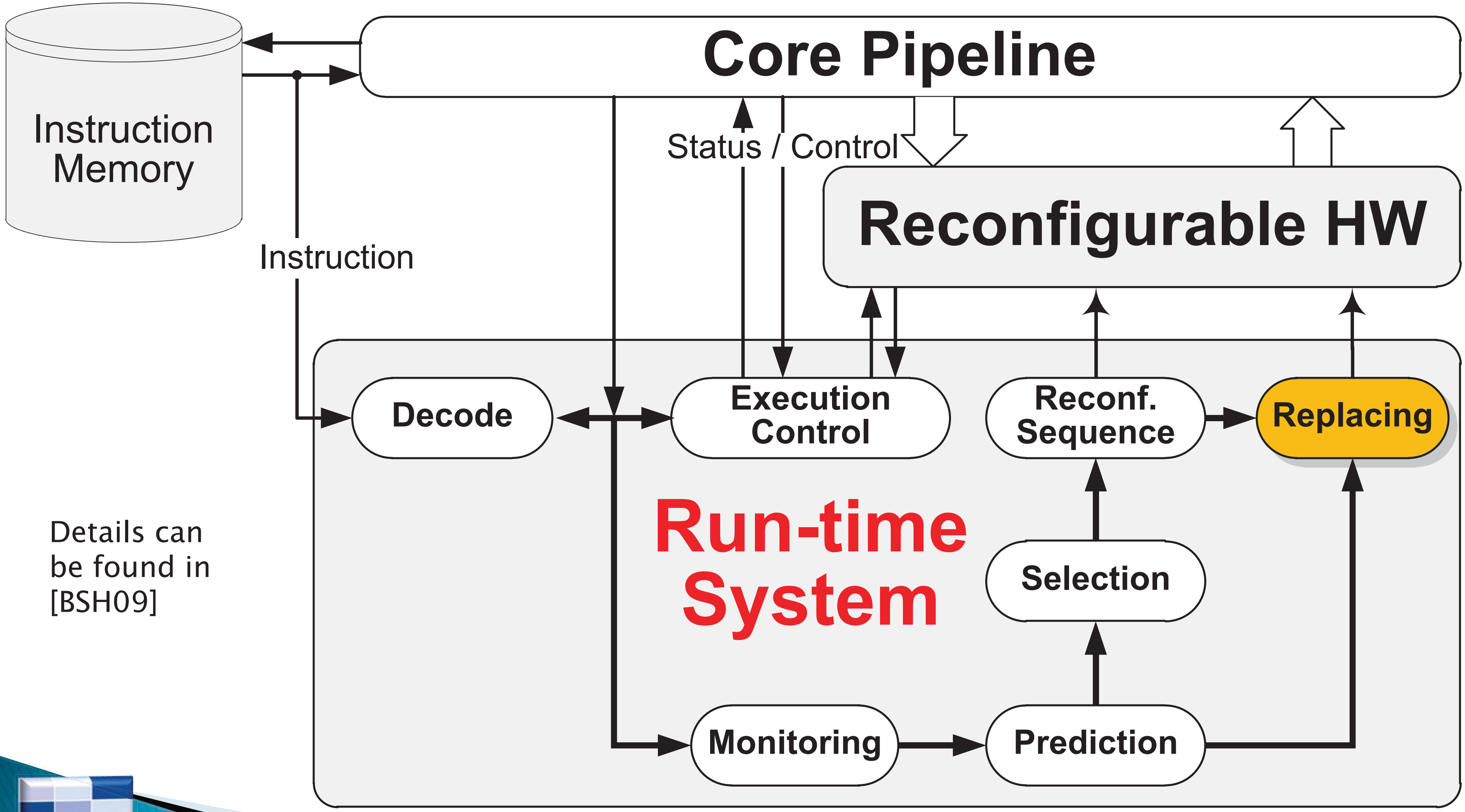


Detailed Analysis of HEF scheduler



Continuation of Latency lines for SAD and SATD are omitted for clarity

Runtime System: Simplified Overview



Details can be found in [BSH09]

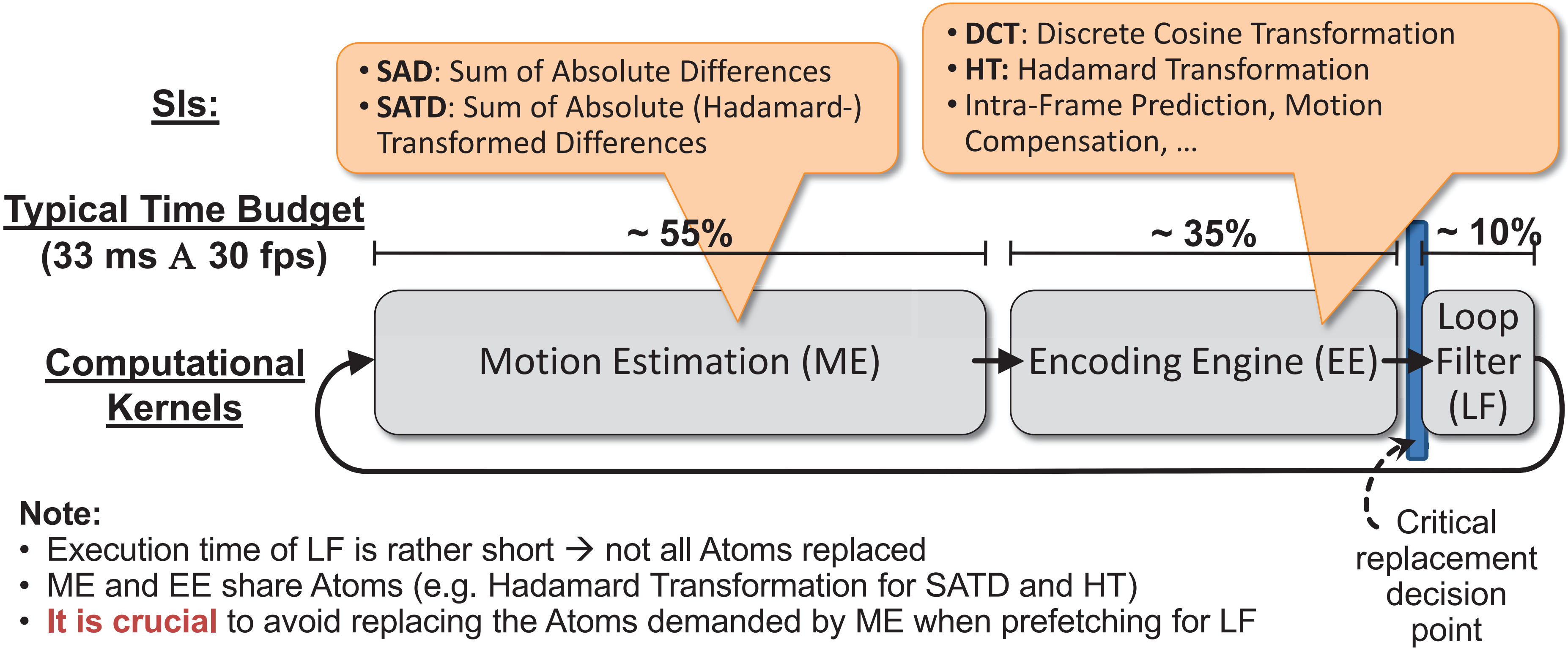
Replacing Atoms

- Whenever all Atom Containers in the reconfigurable fabric are utilized and a new Atom shall be reconfigured (due to Selection and Scheduling) then an existing Atom needs to be replaced
- This Atom may be required again (as typically the different hot spots of the application are executed in a loop)
- We should avoid replacing those Atoms that are required soon
- Optimal solution for memory pages (aka Bélády's replacement): replace that page that is not required for the longest time
  - Drawback: future knowledge required
  - Actual Atom usage is hard to predict due to Atom sharing and as it depends on the Selection
  - Even if future knowledge would be available, Bélády's replacement would not be optimal for Atom replacement. Difference: memory pages are really 'required' and the system has to be stalled until they are fetched; Atoms are not required, they just speed up the computation

Typical replacement policies

Policy	Description	Examined Information
LRU	Least Recently Used	When was it used?
MRU	Most Recently Used	
LFU	Least Frequently Used	How often was it used?
MFU	Most Frequently Used	
FIFO	First In First Out	When was it reconfigured?
LIFO	Last In First Out	
Second Chance / Clock	Extension of FIFO: Each Atom in the queue has a flag that is set when it is used. When an Atom shall be replaced (according the FIFO policy) but the flag is set, it gets a second chance, i.e. its flag is cleared and it is moved to the beginning of the FIFO queue (as if it were new). 'Clock' is a different implementation of the same policy.	

High-level H.264 video encoder flow, showing replacement decisions for LRU & MRU



SIs:

- **SAD**: Sum of Absolute Differences
- **SATD**: Sum of Absolute (Hadamard-) Transformed Differences

- **DCT**: Discrete Cosine Transformation
- **HT:** Hadamard Transformation
- Intra-Frame Prediction, Motion Compensation, …

Typical Time Budget
(33 ms A 30 fps)

Computational Kernels

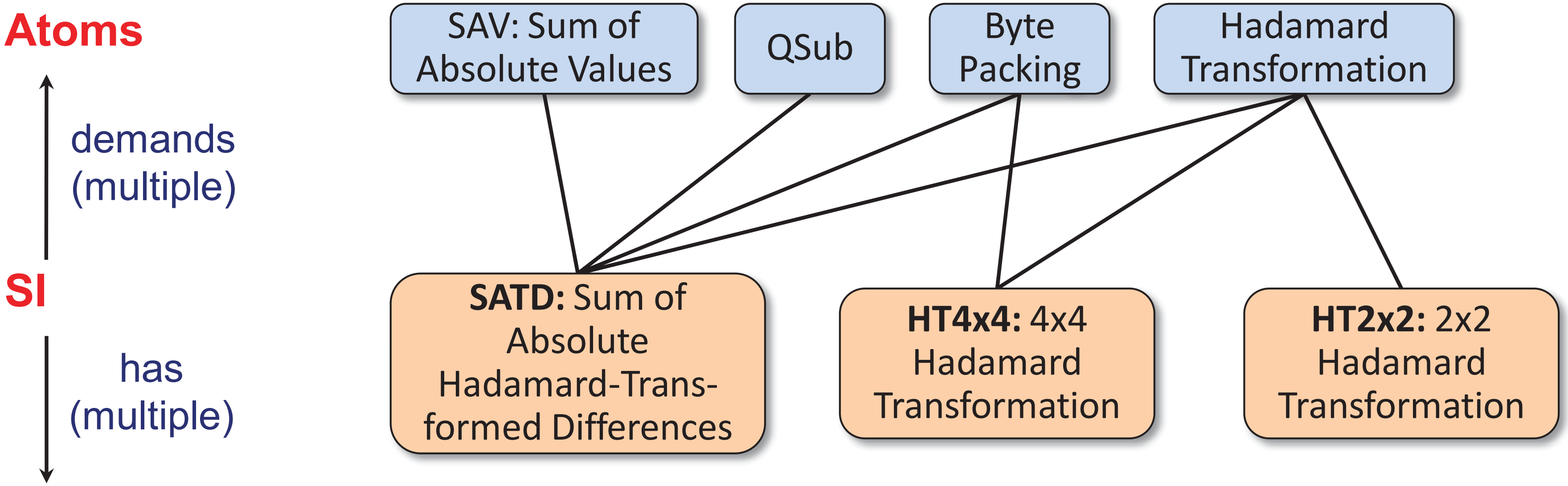
Critical replacement decision point

Note:

- Execution time of LF is rather short → not all Atoms replaced
- ME and EE share Atoms (e.g. Hadamard Transformation for SATD and HT)
- **It is crucial** to avoid replacing the Atoms demanded by ME when prefetching for LF

Policy	Replaced Atoms when prefetching for LF	Demanded for SIs
LRU	Parallel Difference Computation and Accumulation	SAD, SATD
MRU	Transformation	SATD, DCT, HT

Example for performance-wise impact of replacement decision



Atoms

demands (multiple)

SI

has (multiple)

SAV: Sum of Absolute Values

QSub

Byte Packing

Hadamard Transformation

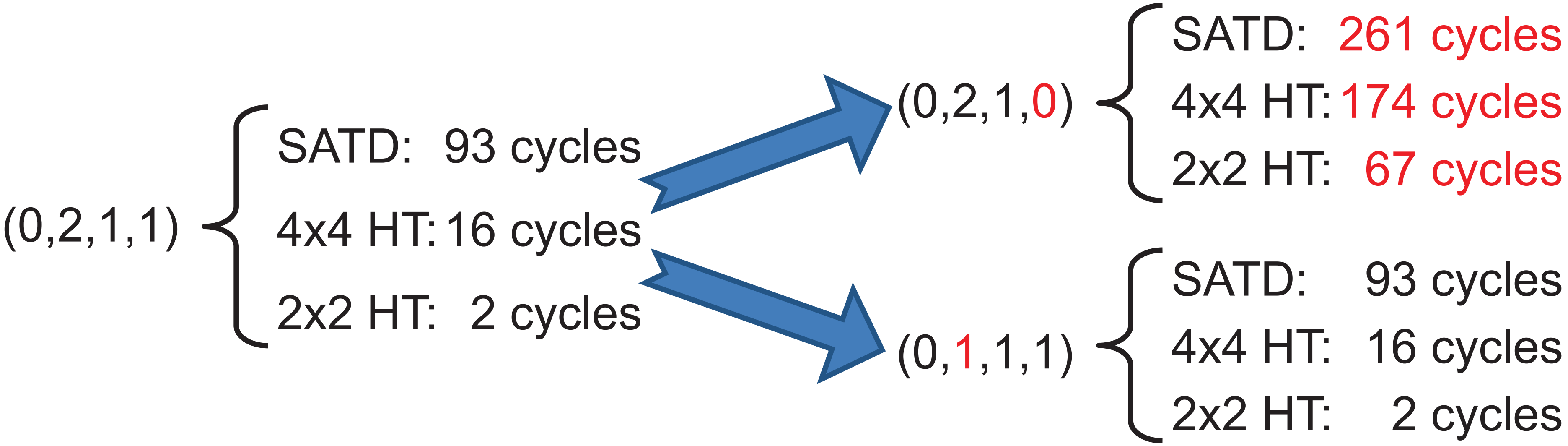
SATD: Sum of Absolute Hadamard-Transformed Differences

HT4x4: 4x4 Hadamard Transformation

HT2x2: 2x2 Hadamard Transformation

Molecules	(0,0,0,0)→319 cycles	(0,0,0,0)→201 cycles	(0,0,0,0)→67 cycles
	(0,0,1,0)→261 cycles	(0,0,1,0)→174 cycles	(0,0,0,1)→2 cycles
	(0,0,1,1)→173 cycles	(0,0,1,1)→16 cycles	
	(0,1,1,1)→93 cycles	(0,0,2,2)→11 cycles	
	(1,1,1,1)→31 cycles	…	
	(1,2,2,2)→27 cycles		
	…		

Example for performance-wise impact of replacement decision



(0,2,1,1)

SATD: 93 cycles

4x4 HT: 16 cycles

2x2 HT: 2 cycles

(0,2,1,0)

SATD: 261 cycles

4x4 HT: 174 cycles

2x2 HT: 67 cycles

(0,1,1,1)

SATD: 93 cycles

4x4 HT: 16 cycles

2x2 HT: 2 cycles

- Depending on the replaced Atoms, all SIs might be affected
    - Some Atoms are critical for the performance and thus should not be replaced
- This is independent of history-based matters, e.g. ‘when’ they were reconfigured, ‘how often’ they were used etc.

Determining Replacement Candidates

- Some Atoms are selected to implement SIs

$$\vec{s} := (s_0, \ldots, s_{n-1})$$

- Some Atoms are currently available

$$\vec{a} := (a_0, \ldots, a_{n-1})$$

- Some Atoms need to be reconfigured (prefetching selected them but they are currently not available)

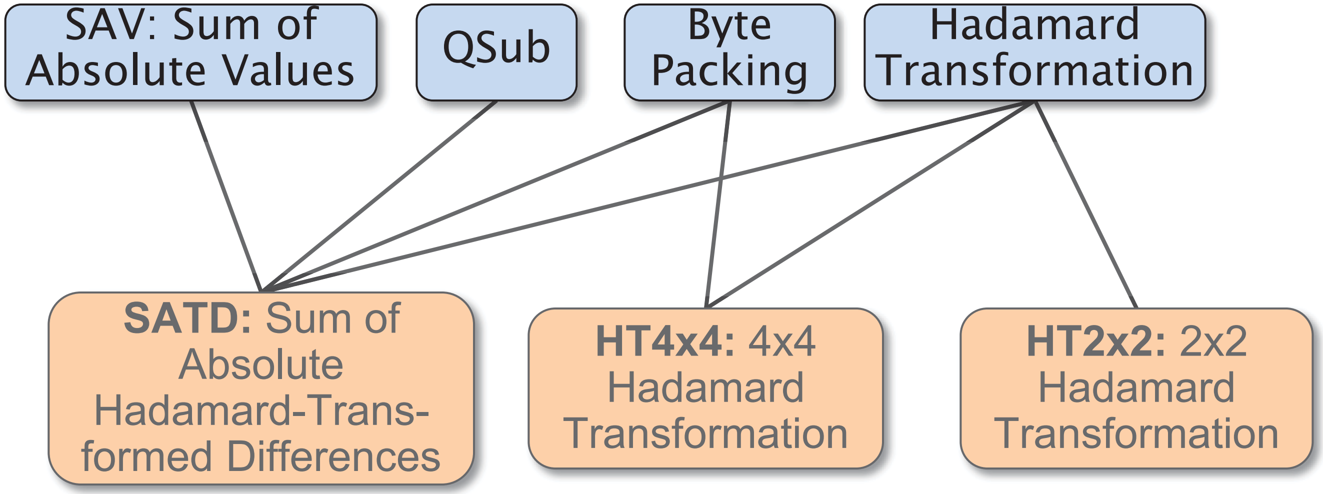
$$\vec{s} / \vec{a}$$

- Some Atoms are replacement candidates (they are available but prefetching did not select them)

$$\vec{c} := \vec{a} / \vec{s}$$

- Next: determine the Atom that leads to the minimum performance degradation, accumulated over all SIs: MinDeg

MinDeg Algorithm: Example



SAV: Sum of Absolute Values

QSub

Byte Packing

Hadamard Transformation

SATD: Sum of Absolute Hadamard-Transformed Differences

HT4x4: 4x4 Hadamard Transformation

HT2x2: 2x2 Hadamard Transformation

SATD	HT4x4	HT2x2
(0,0,0,0)→319 cycles	(0,0,0,0)→201 cycles	(0,0,0,0)→67 cycles
(0,0,1,0)→261 cycles	(0,0,1,0)→174 cycles	(0,0,0,1)→2 cycles
(0,0,1,1)→173 cycles	(0,0,1,1)→16 cycles	
(0,1,1,1)→93 cycles	(0,0,2,2)→11 cycles	
(1,1,1,1)→31 cycles	…	
(1,2,2,2)→27 cycles		
…		

- Available Atoms

  $\vec{a} := (1, 2, 1, 1)$
- Replacement Candidates

  $\vec{c} := (0, 2, 1, 1)$
- Candidate:

  $(0, 1, 0, 0)$
- Afterwards available Atoms

  $(1, 1, 1, 1)$

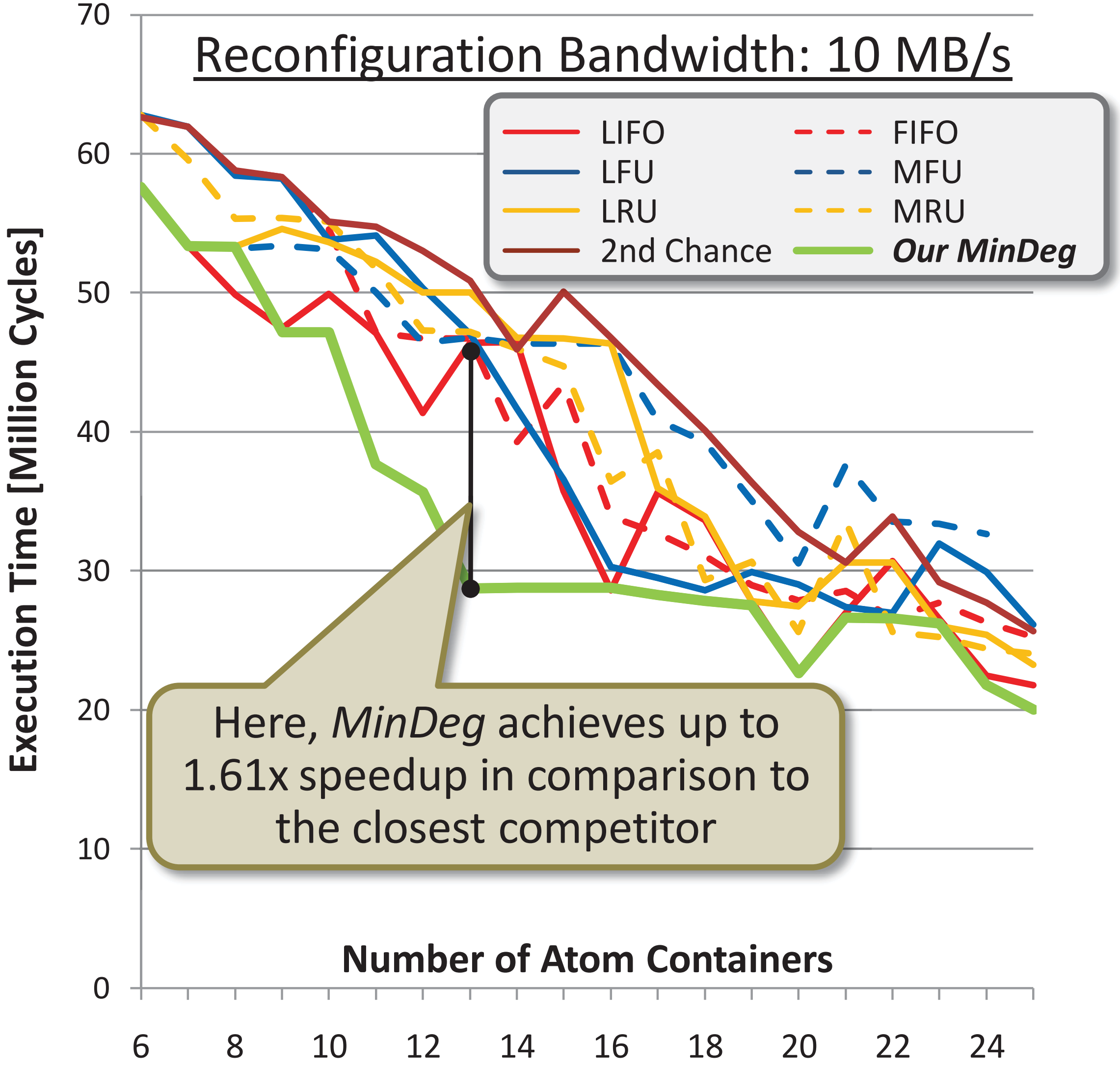
(0,0,0,1)→261+174+67=502 cycles

(0,0,1,0)→319+201+67=587 cycles

(0,1,0,0)→ 31+ 16+ 2= 49 cycles

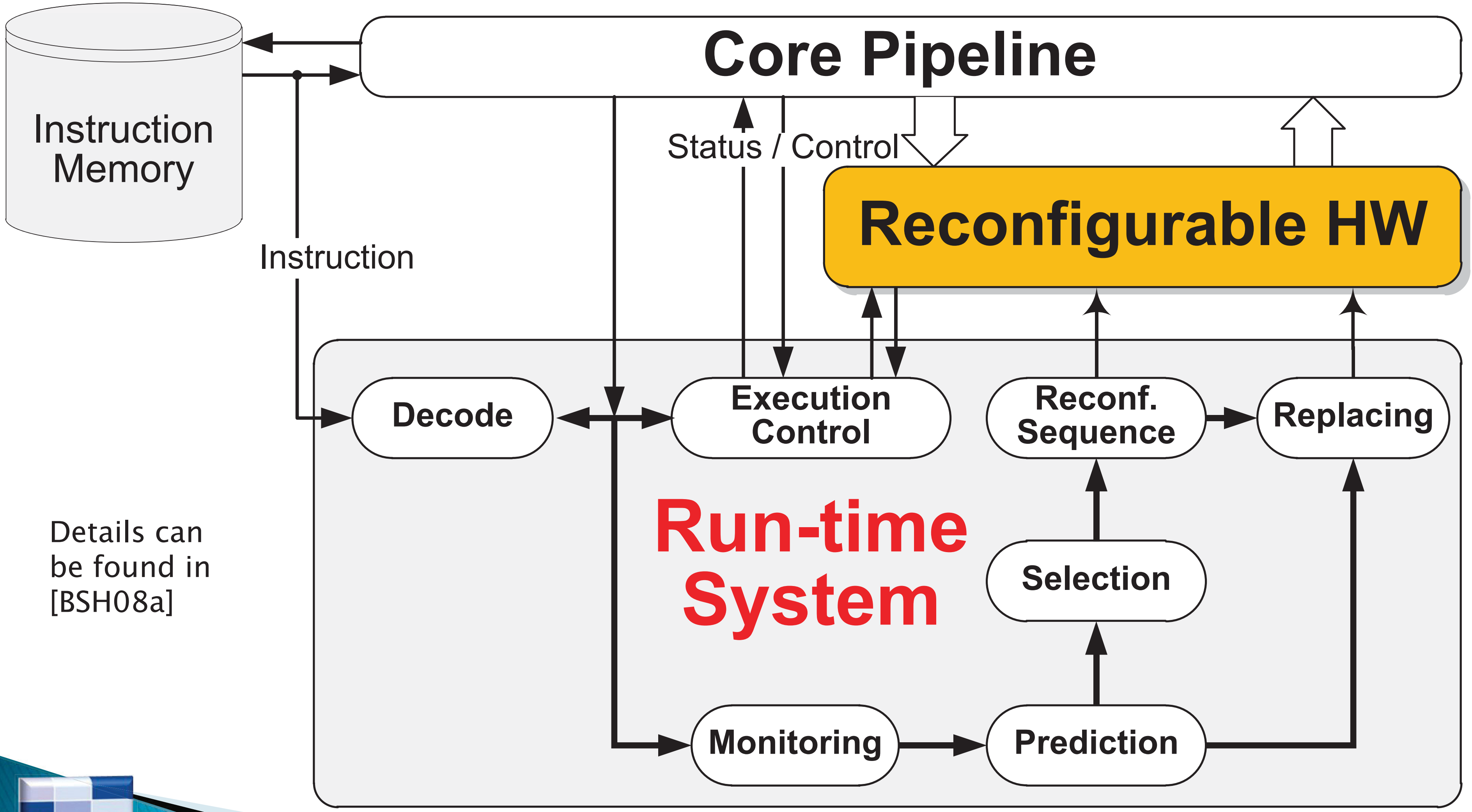
Application Execution Speed

- When a rather small reconfigurable fabric is available, then often all Atoms need to be replaced (minor impact of replacement policy)
- When a rather large fabric is available, then all ever-demanded Atoms might fit to the fabric at the same time (minor impact of replacement function)
- In between, MinDeg provides the best performance



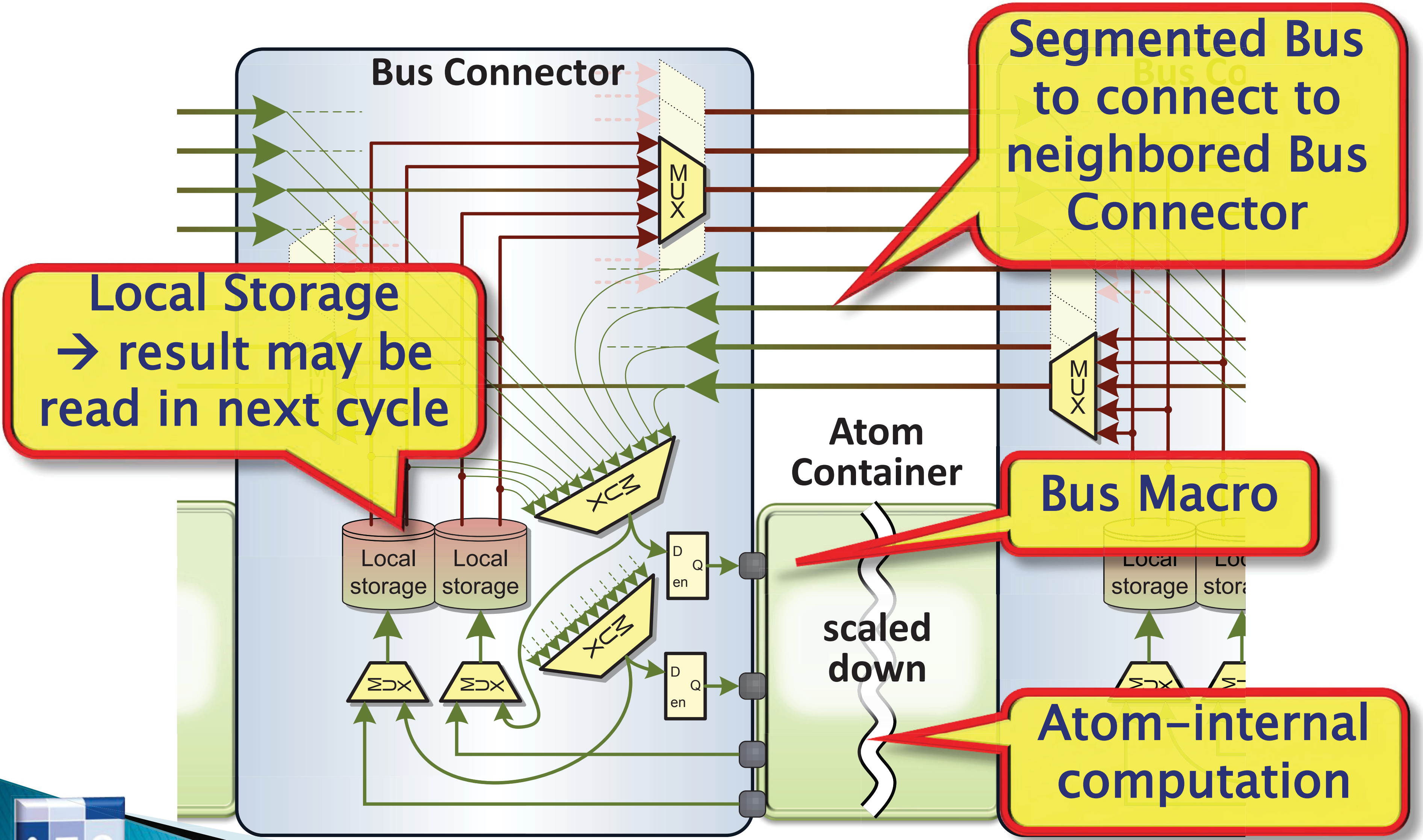
Here, *MinDeg* achieves up to 1.61x speedup in comparison to the closest competitor

Runtime System: Simplified Overview



Details can be found in [BSH08a]

Infrastructure for Modular SIs



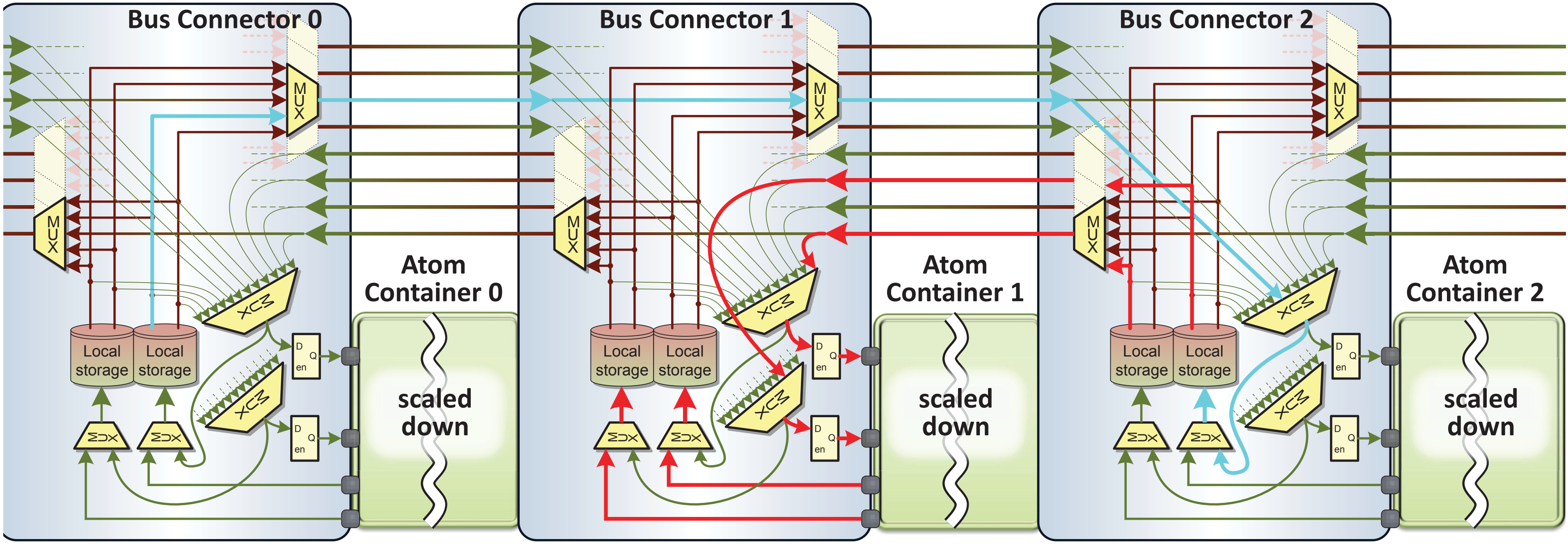
Segmented Bus to connect to neighbored Bus Connector

Local Storage → result may be read in next cycle

Bus Macro

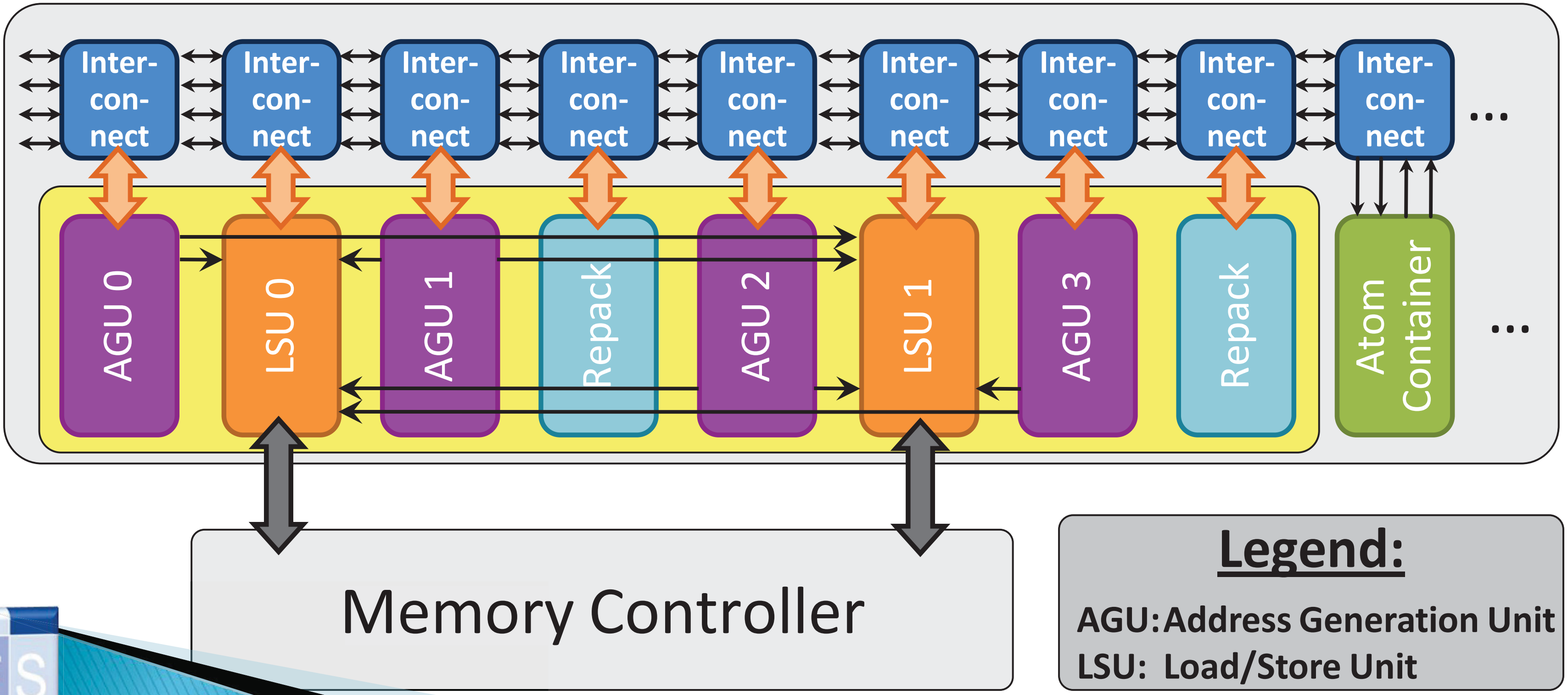
Atom-internal computation

Infrastructure for Modular SIs (cont'd)



Details of non-reconfigurable parts

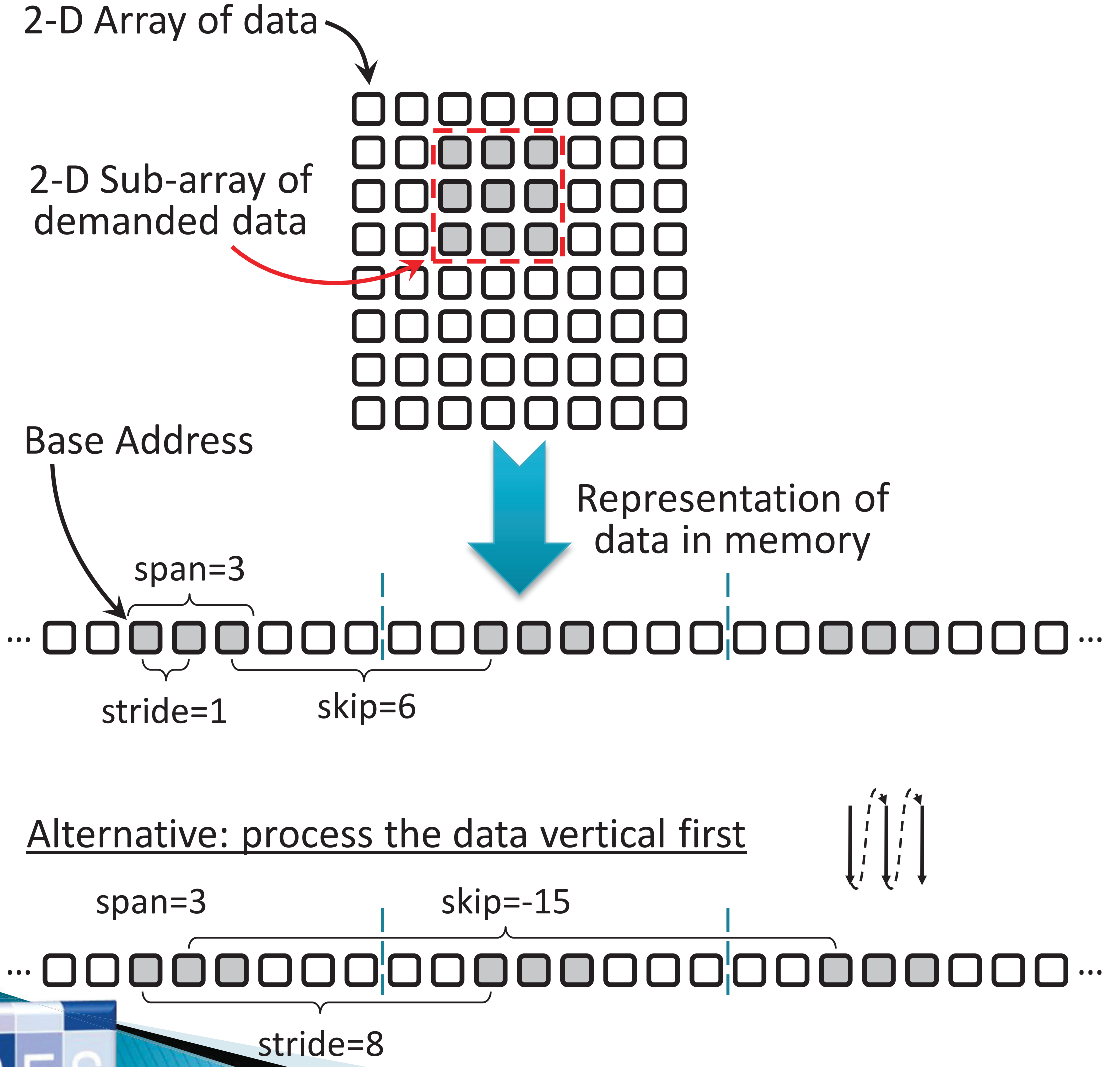
- In addition to the reconfigurable Atom Containers, there are several non-reconfigurable components connected to the bus
  - Load/Store Units (LSU), Address Generation Units (AGU), and Repack (Byte-wise rearrangement of data)



Legend:

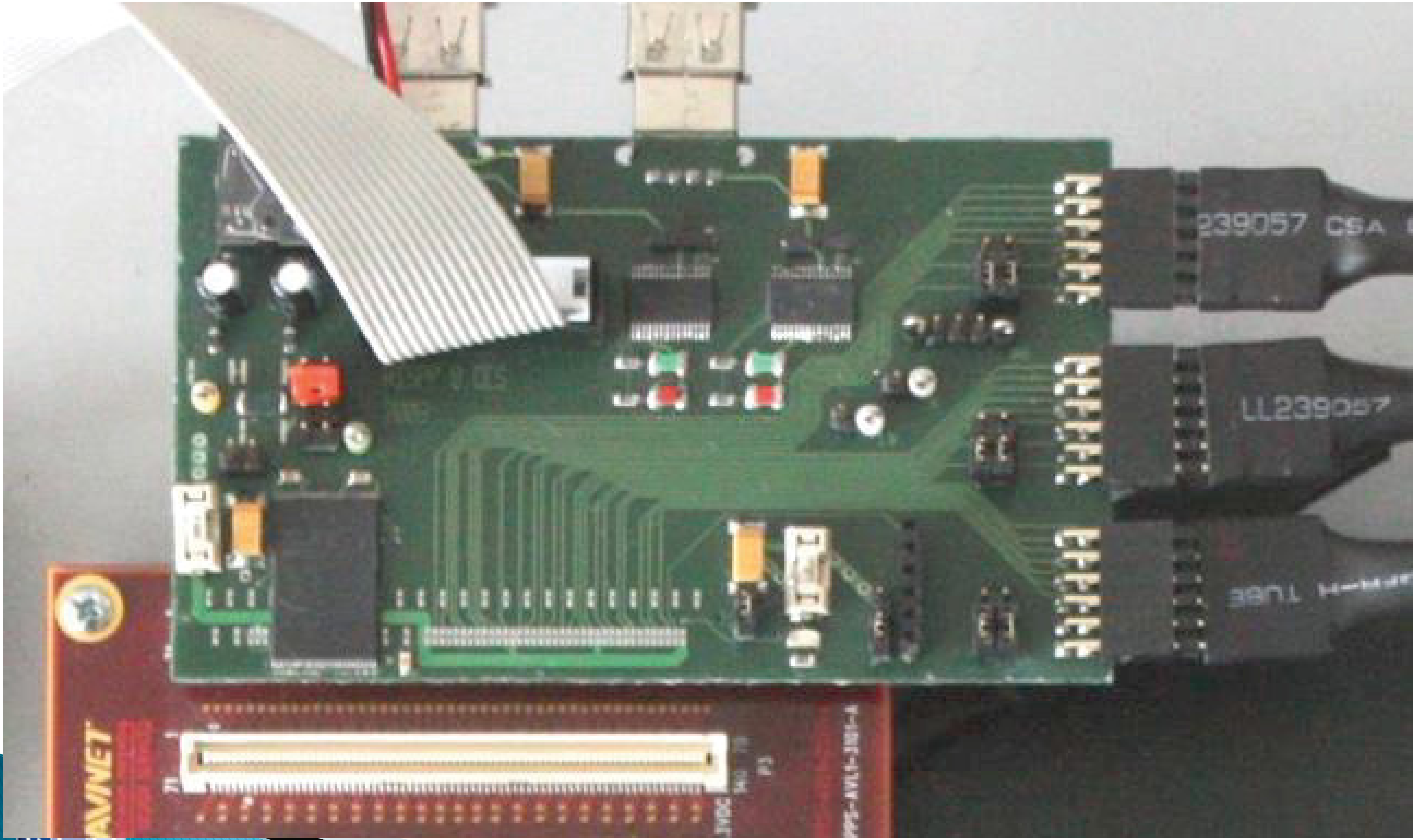
AGU: Address Generation Unit
LSU: Load/Store Unit

Details of AGU

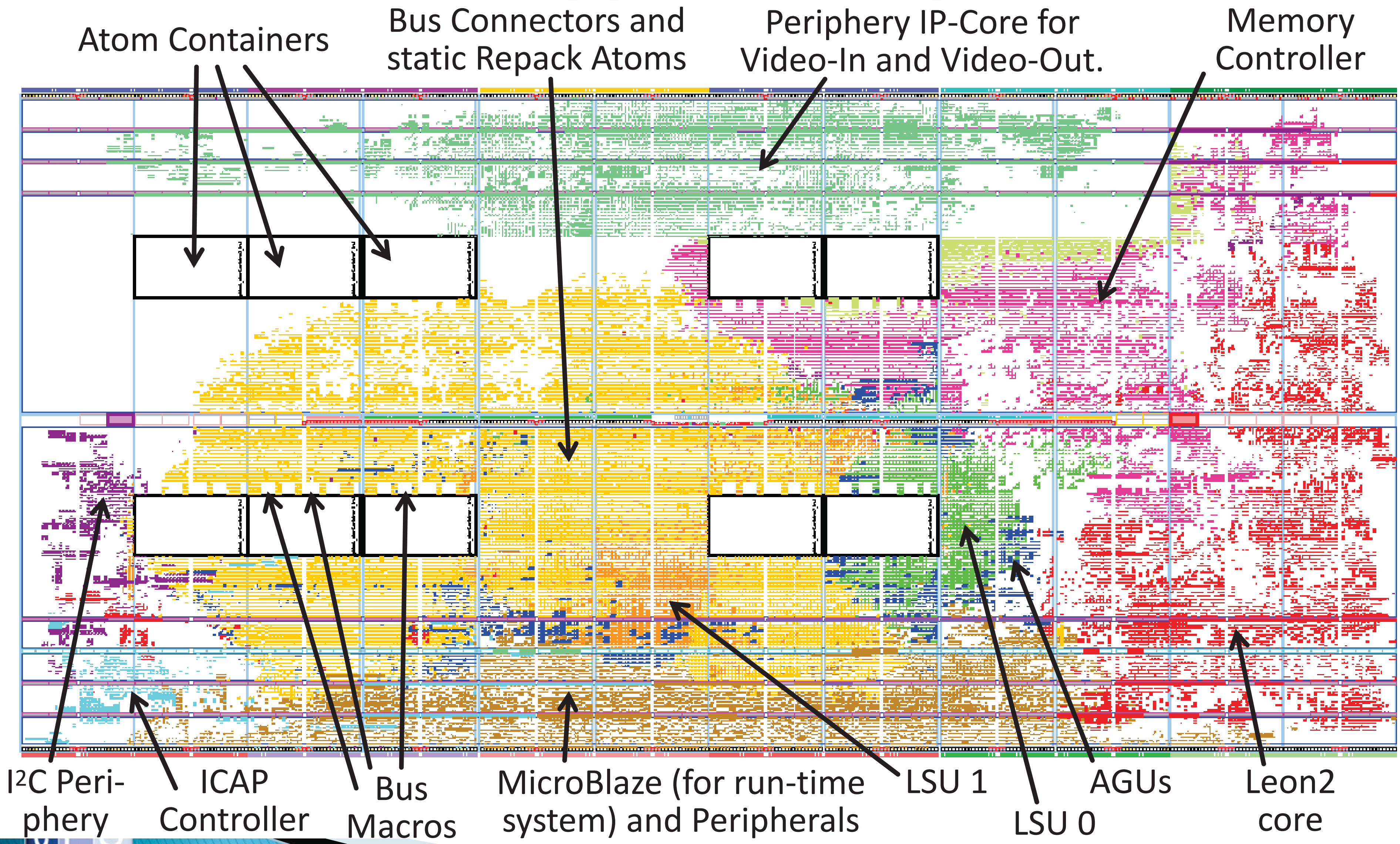

2-D Array of data
2-D Sub-array of demanded data
Base Address
Representation of data in memory
span=3
stride=1
skip=6
...
...
Alternative: process the data vertical first
span=3
skip=-15
stride=8

- AGU initialization
  - Baseaddress, Stride, Span, Skip
  - Based on parameters of SI (constants or from register file)
- 4 AGUs can be used to describe 4 different memory streams
  - e.g. reading from two different arrays and writing to two different arrays
- Each AGU pre-computes the 'next' and the 'next next' address
  - required to feed both LSUs at the same time (e.g. using both LSUs to read only one memory stream)

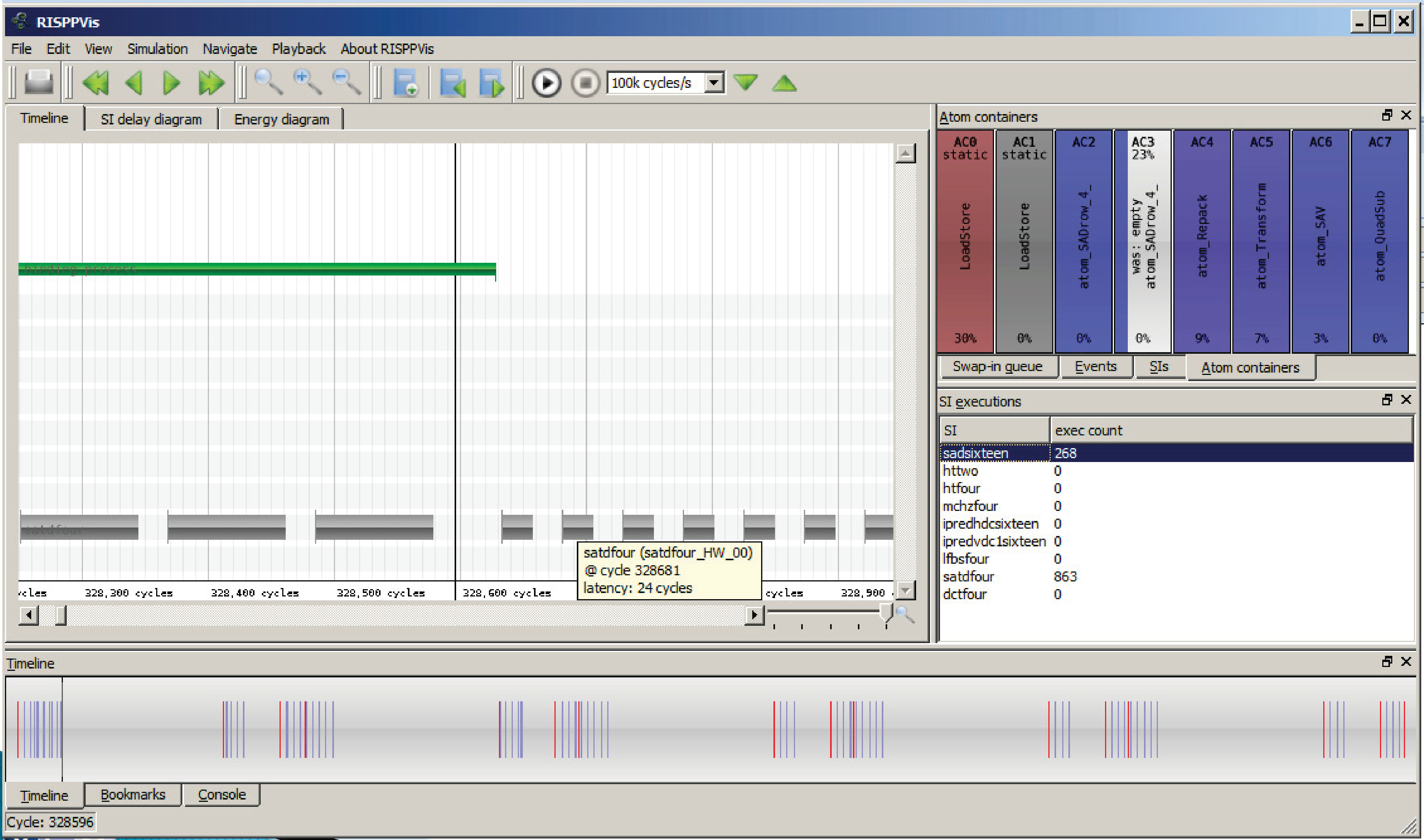
FPGA-based Prototype



RISPP Prototype Floorplan


Atom Containers
Bus Connectors and static Repack Atoms
Periphery IP-Core for Video-In and Video-Out.
Memory Controller
I²C Periphery
ICAP Controller
Bus Macros
MicroBlaze (for run-time system) and Peripherals
LSU 1
LSU 0
AGUs
Leon2 core

RISPP Simulator GUI

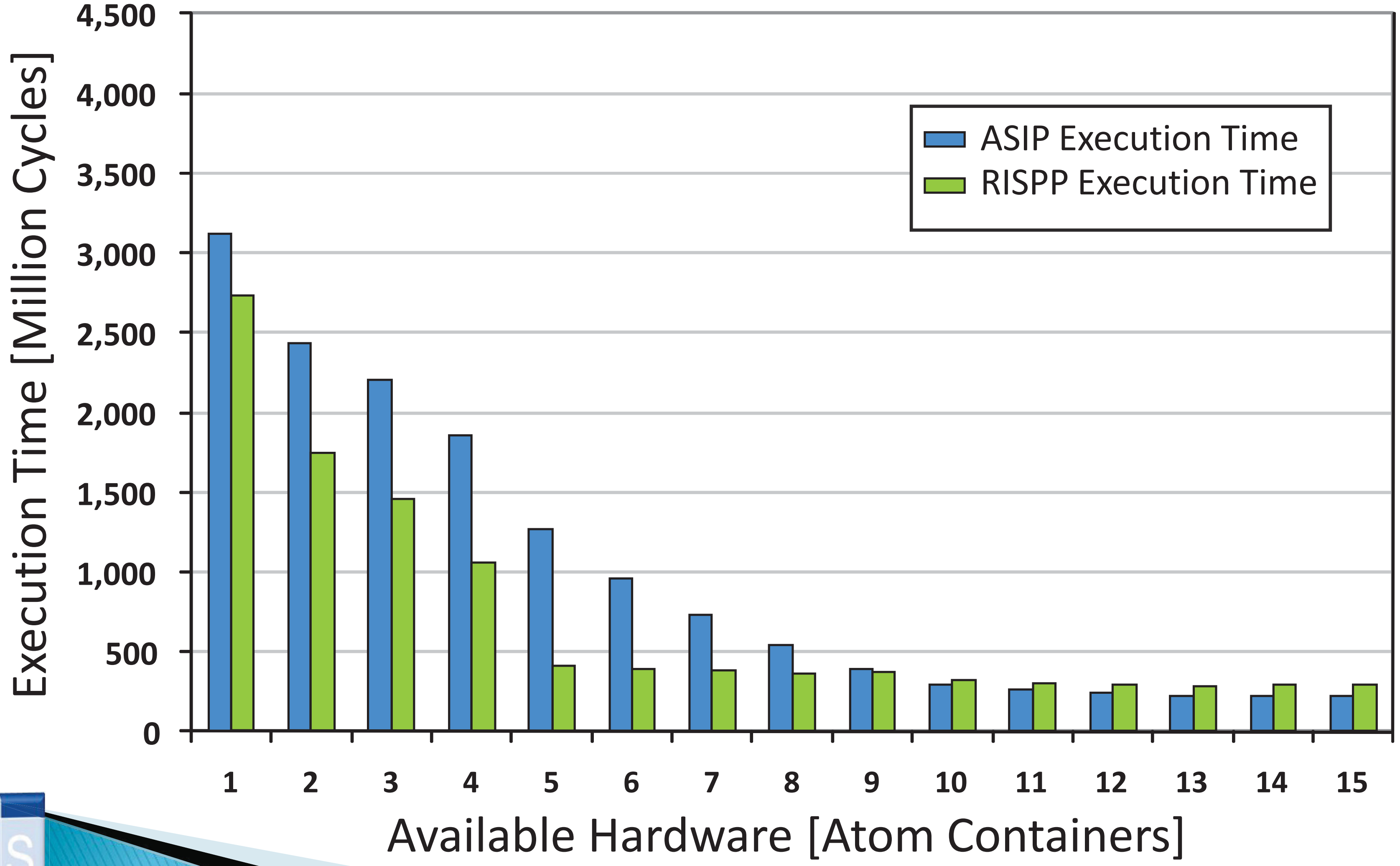


satdfour (satdfour_HW_00)
@ cycle 328681
latency: 24 cycles

SI	exec count
sadsixteen	268
httwo	0
htfour	0
mchzfour	0
ipredhdcsixteen	0
ipredvdc1sixteen	0
lfbsfour	0
satdfour	863
dctfour	0

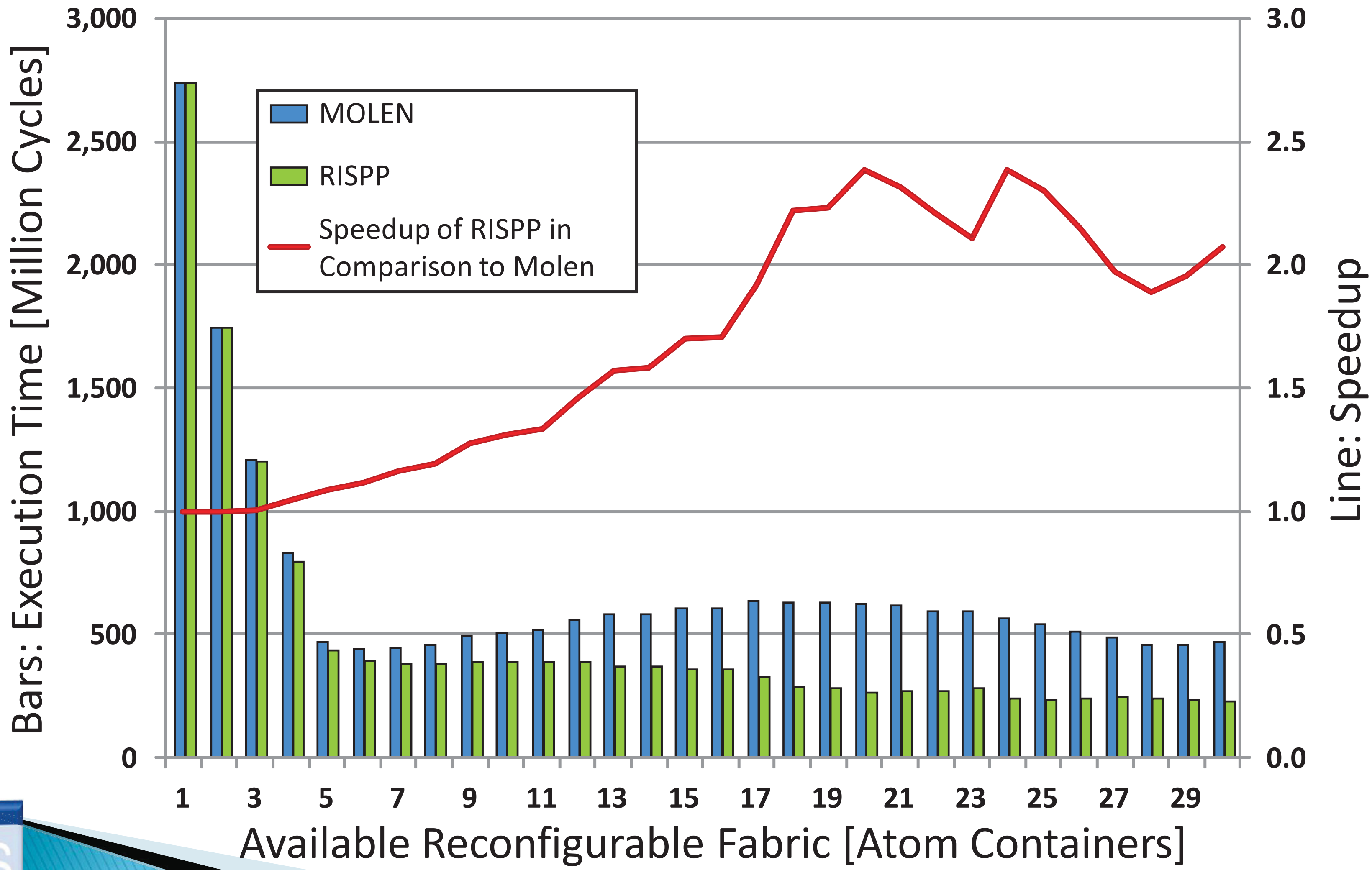
Cycle: 328596

H.264 comparison with State-of-the-art ASIPs



src: [BSH08c]

H.264 comparison with State-of-the-art Reconfigurable Processors: Molen



src: [BSKH08]

Overall System Evaluation

	Min	Avg	Max
H.264 Video Encoder	1.11x	15.80x	22.21x
SUSAN Image Processing	1.22x	14.48x	15.99x
SHA	6.10x	6.44x	6.45x
ADPCM Encoder	1.17x	5.00x	5.16x
JPEG Decoder	1.23x	3.31x	3.79x

- Application Speedup compared to Leon-only
  - Depending on number of available Atom Containers (in simulation up to 20)

RISPP Summary

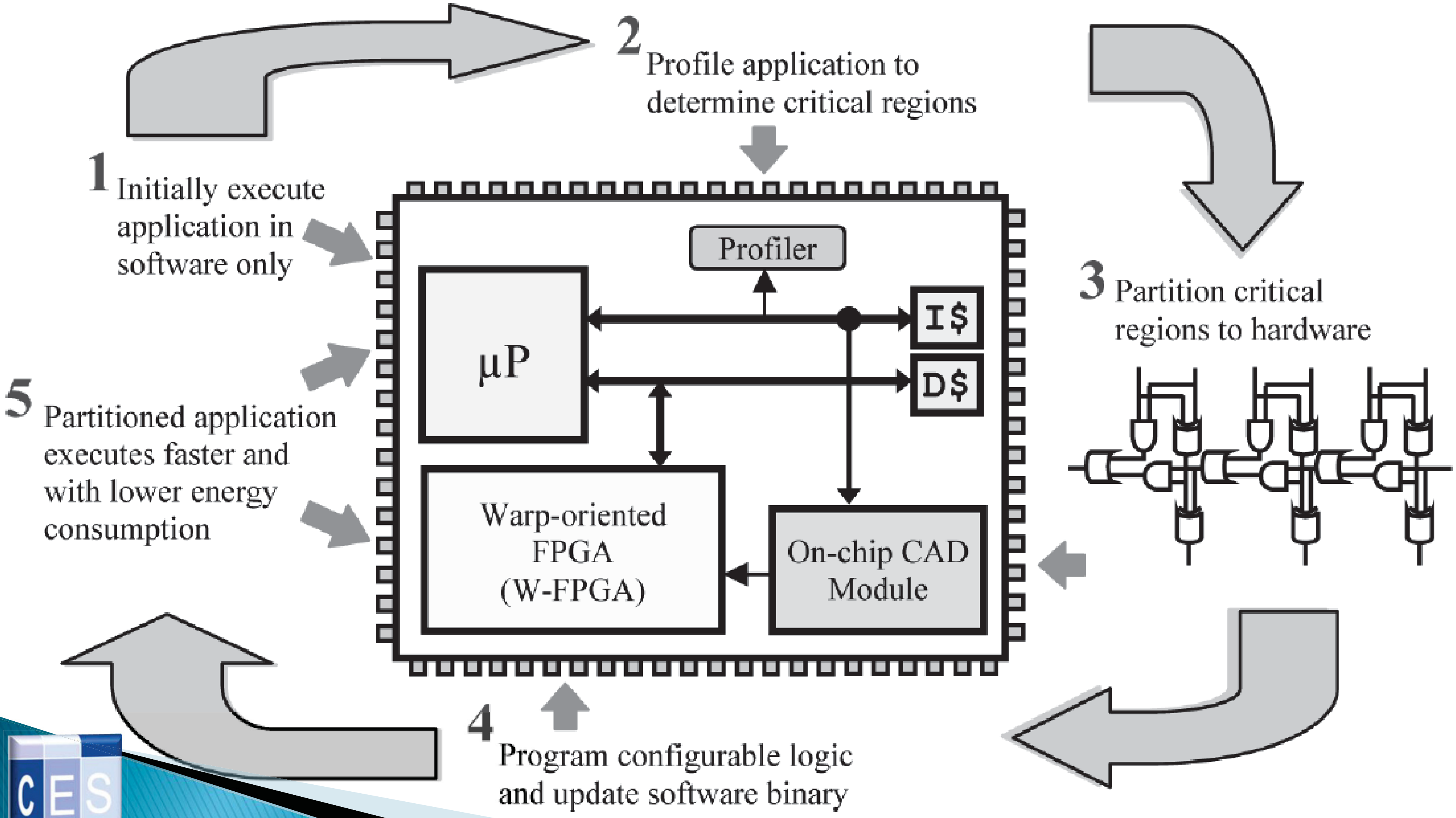
- Novel hierarchical Special Instruction composition, enabling different performance-area trade-offs
- RISPP provides very high adaptivity that is demanded for changing control flow (e.g. depending on input data)
- Solved the reconfiguration overhead problem by upgrading the SIs
- Evaluated using simulations and FPGA-based prototype
- Conservative Comparison with state-of-the-art
  - Comparison with ASIP: up to 3.06x faster
  - Comparison with Molen: up to 2.38x faster
  - Comparison with Proteus: up to 7.19x faster
  - Compared to Leon 2 GPP: up to 26.6x faster

7.2 WARP Processor

Overview

- Fine-grained loosely-coupled Coprocessor
- No compiler required; works on standard binaries
- Detects application hot spots during runtime
- Re-implements hot spots as Special Instructions
  - → Online Synthesis
- Developed special FPGA fabric and special place & route tools for online synthesis

WARP Architecture and Runtime Flow



1 Initially execute application in software only

2 Profile application to determine critical regions

3 Partition critical regions to hardware

4 Program configurable logic and update software binary

5 Partitioned application executes faster and with lower energy consumption

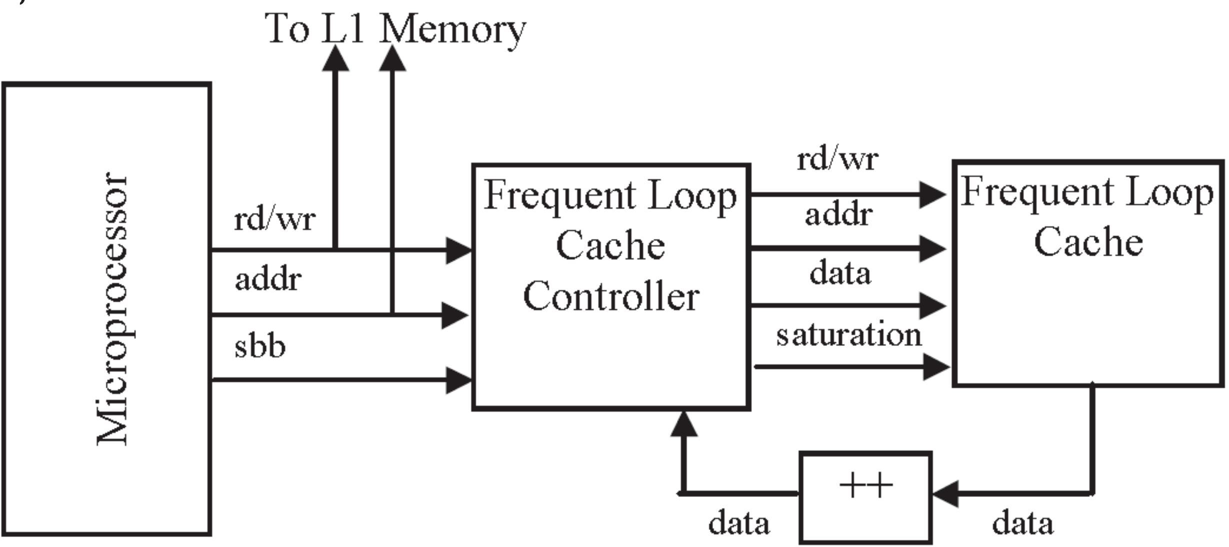
src: [LSV06]

Determining critical kernels by online profiling

- Typically, the critical kernels correspond to frequently executed (inner) loops
- Characteristic of inner loops: ends with a short backward branch (sbb) targeting the beginning of the loop
  - 'short' means: small offset compared to current instruction memory address
- Generally unknown how many different inner loops exist
  - → use a Cache architecture to track the most important ones (i.e. those with the highest execution frequency)

Determining critical kernels by online profiling (cont'd)

- On a miss in that cache (currently unknown sbb needs to be stored) replace the least frequently used sbb (loss of accuracy)
- On overflow in any counter halve all values (shift)
  - Emphasizes on recent sbb activities
  - Loss of accuracy; but critical kernels still can be detected
  - Halving is done as a feature of the cache either parallel(area) or sequential (latency overhead)



src: [GV03]

Determining critical kernels by online profiling (cont'd)

- The Cache Controller can detect sbb instructions automatically by partially decoding the executed instruction
- Non-intrusive System (µP not modified)
  - Important for real-time systems where changes in execution behavior could significantly affect the guarantees
  - Additionally minimizes the impact on current tool chains, e.g. avoids special compilers or binary modification tools
- Extension: Coalescing
  - When the inner loop executes several times, the cache controller in the online monitoring is very active in reading, incrementing, writing the cache → high power consumption
  - Instead: count all executions of one inner loop separately and whenever another loop executes, then update the cache once

Online Synthesis

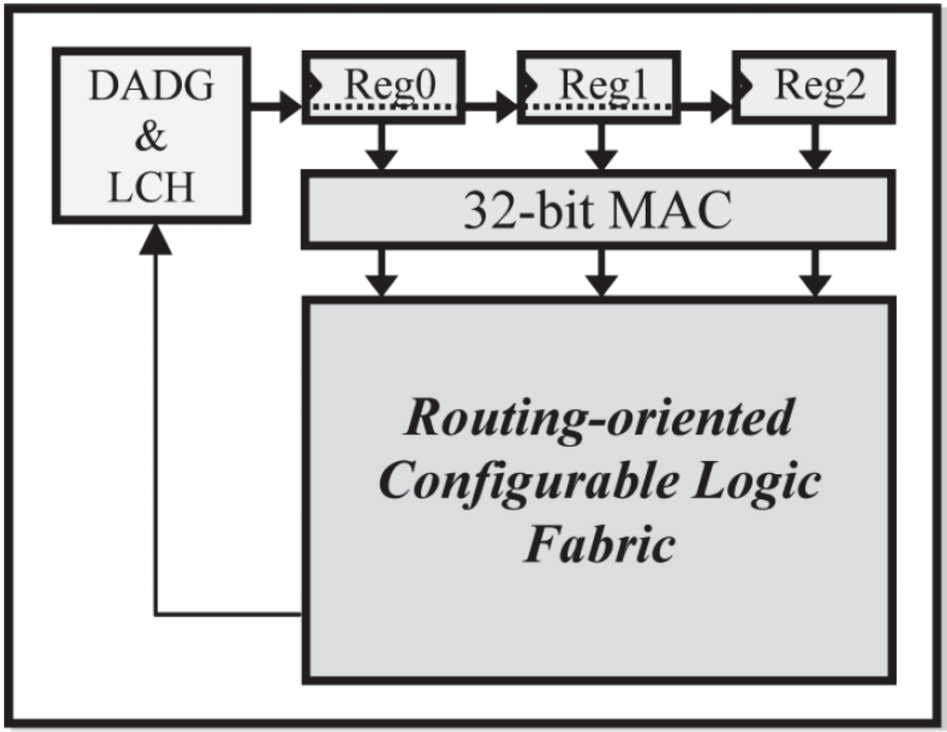
- **Challenges**: The online synthesis (CAD tool) needs to execute online while the user application is running
  - Typically CAD tools executes offline on a powerful workstation
  - Demanding high memory (GB) and computational resources (minutes to hours to implement accelerators)
- **Simplification**: Warp targets seldom-changing, long-running applications
  - It may be acceptable to spend seconds to minutes for online synthesis after the application started (once!), if it runs faster afterwards
  - Limits the adaptivity during application execution while maintaining a high flexibility to accelerate any type of application
- **But:** memory problem remains (time is available if you are willing to wait; gigabytes of memory are not)

Reducing Memory- and Computational requirements for online synthesis

- Simplified FPGA
  - Smaller LUTs (3-input LUTs; state-of-the-art FPGAs have 4-6 input LUTs) → simplified Mapping and Placement
  - Less LUTs per CLB → simplified Mapping and Placement
  - Fixed routing inside a CLB → simplified Placement and Routing
  - Simplified Switching Matrices (less connections per Switching Matrix and no connection to distant Switching Matrices) → simplified Placement and Routing
- Simplified algorithms
  - Nearly all algorithms (Mapping, Placement, and Routing) are greedy heuristics that do not achieve the quality (e.g. area and latency) of state-of-the-art algorithms
- Together: Trading-off quality vs. runtime overhead

WARP-oriented FPGA

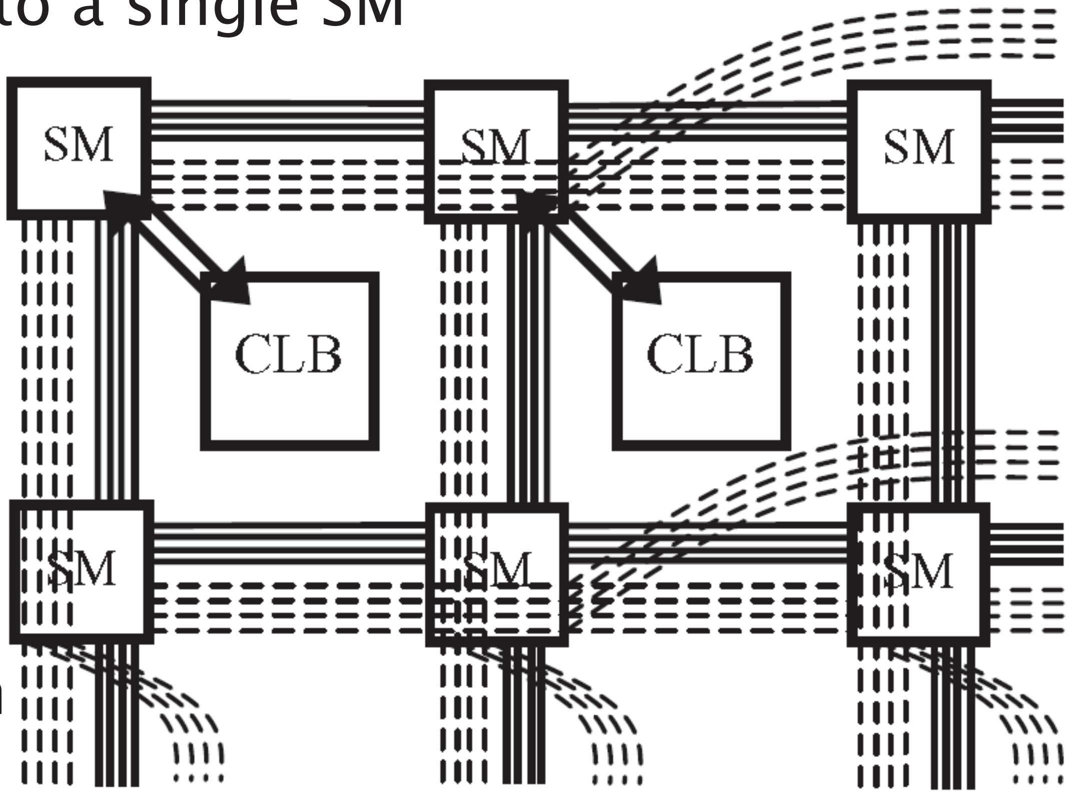
- Contains several hard-wired elements in addition to the actual FPGA
  - Access to memory via Data Address Generator (DADG)
  - Loop Control Hardware (LCH)
  - Input/Output registers
  - Dedicated Multiply Accumulate unit (MAC)
- The core pipeline is stalled during SI execution
  - No cache coherency/ consistency issues
  - Not really co-processor



src: [LSV06]

WARP-oriented FPGA (cont'd)

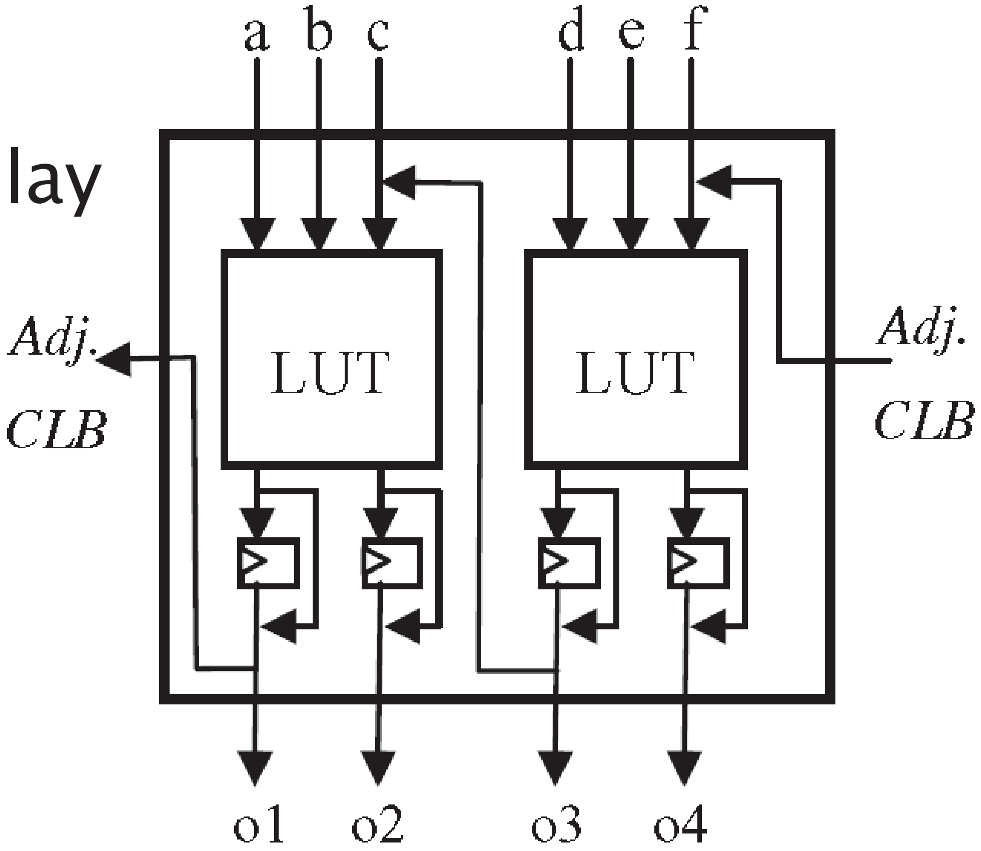
- Simple Configurable Logic Fabric
- CLBs are surrounded by Switching Matrices (SMs)
- Each CLB connected to a single SM
- SMs are interconnected to nearest neighbors (short channels) and to second nearest neighbors (long channels; dashed lines) in horizontal and vertical direction



src: [LVT05]

WARP-oriented FPGA (cont'd)

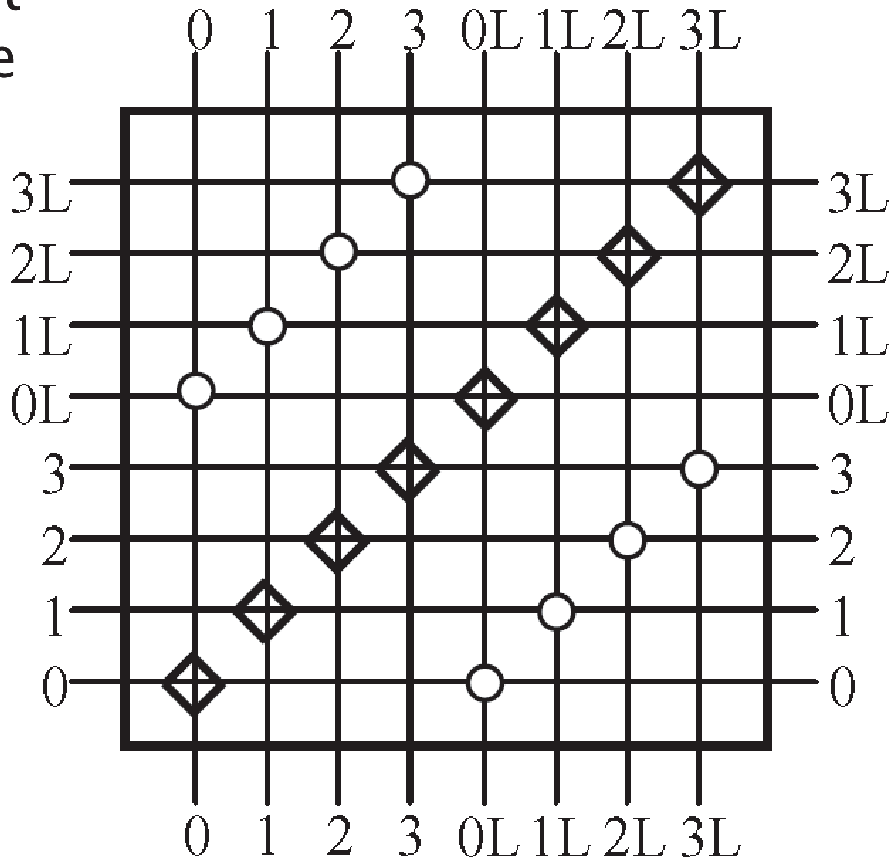
- CLB contains two 3-input/2-output LUTs with optional registers at the outputs
- Provides a trade-off between area and delay
- Simple and regular structure simplifies mapping and placement



src: [LVT05]

WARP-oriented FPGA (cont'd)

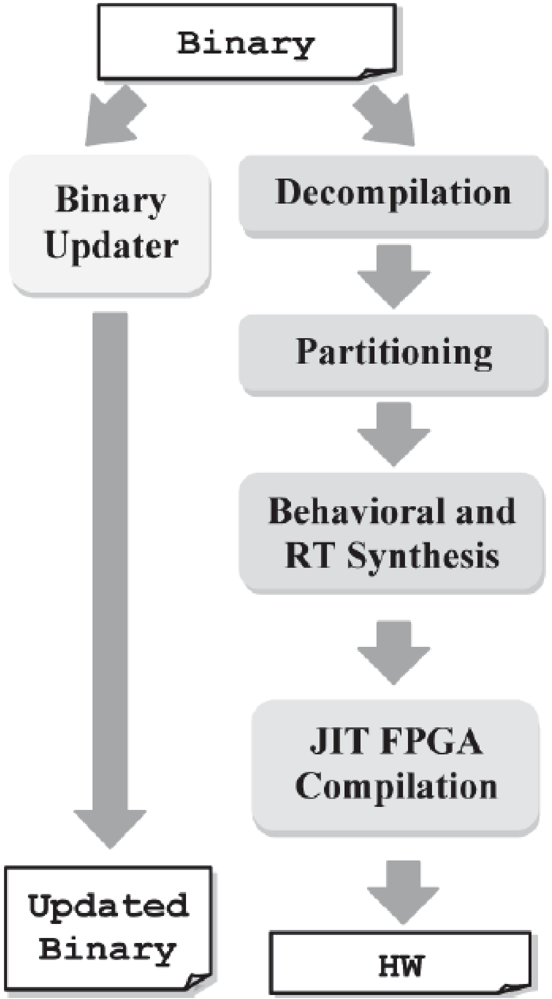
- 4 short channels and 4 long channels (L) per direction
- A channel *i* can only connect to the same channel *i* at one of the 3 other directions (using the diamonds as connectors)
- Additionally the short and the long channels of the same channel number *i* can be connected (using the circles)
- Simplifies the routing



src: [LVT05]

Online Synthesis

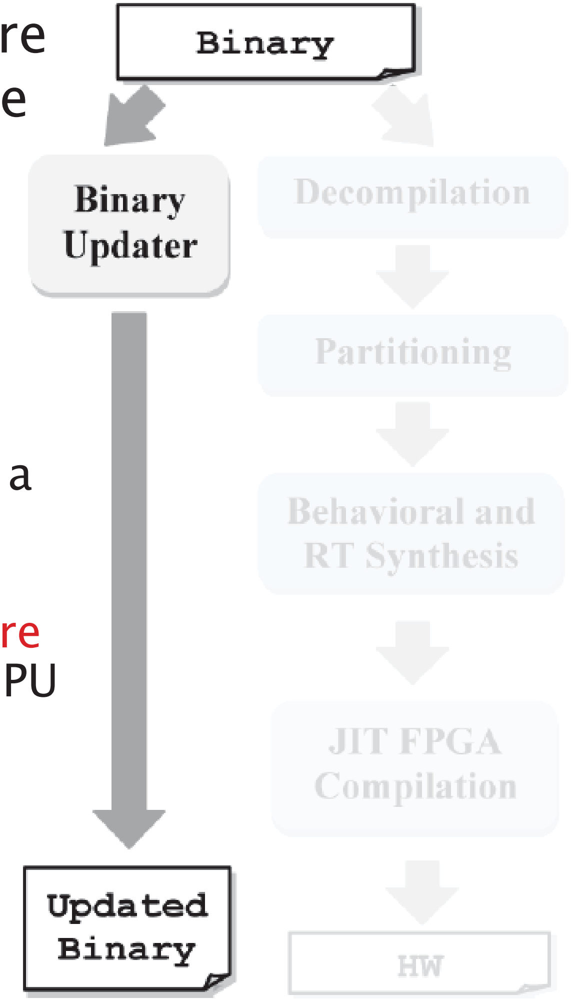
- **Decompilation:** converts binary into a high-level representation (e.g. control/data-flow graph)
- **Partitioning:** selecting critical kernels
- **High-level synthesis:** create netlist (Boolean expressions)
- **Low-level synthesis (FPGA compilation):** FPGA specific place and route
- **Binary updater:** Actually use the new hardware



src: [LSV06]

Calling Special Instructions

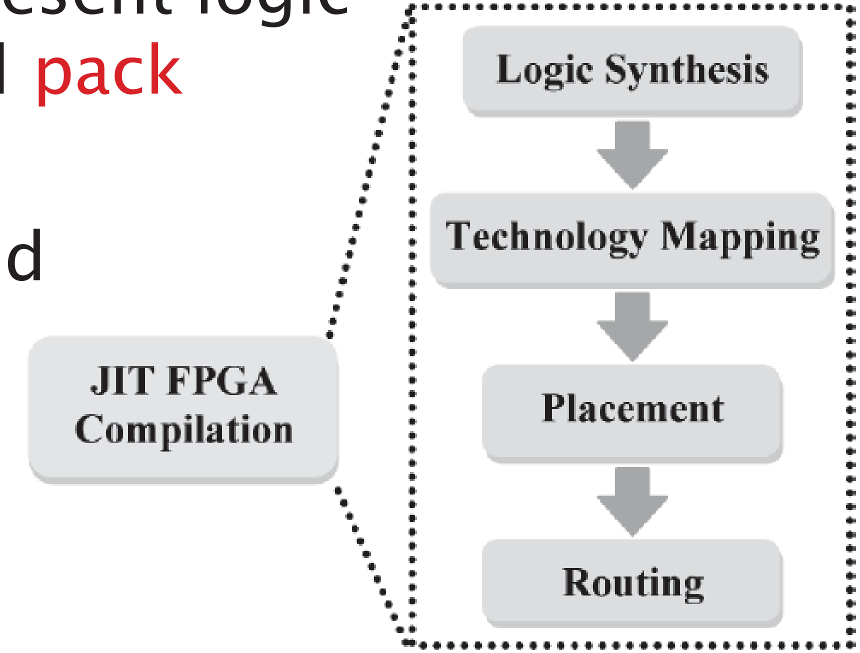
- Problem: application binary is not aware of the Special Instruction (due to online synthesis)
- But: old code is no longer required → may be overwritten
- Solution:
  1. Replace first instruction of old code with a jump to a new hardware initialization handler
  2. This handler prepares & calls the hardware of the Special Instruction and stalls the CPU pipeline
  3. When the Special Instruction completes, the handler jumps to the instruction that follows the last instruction of the old code



src: [LSV06]

Low-Level Synthesis

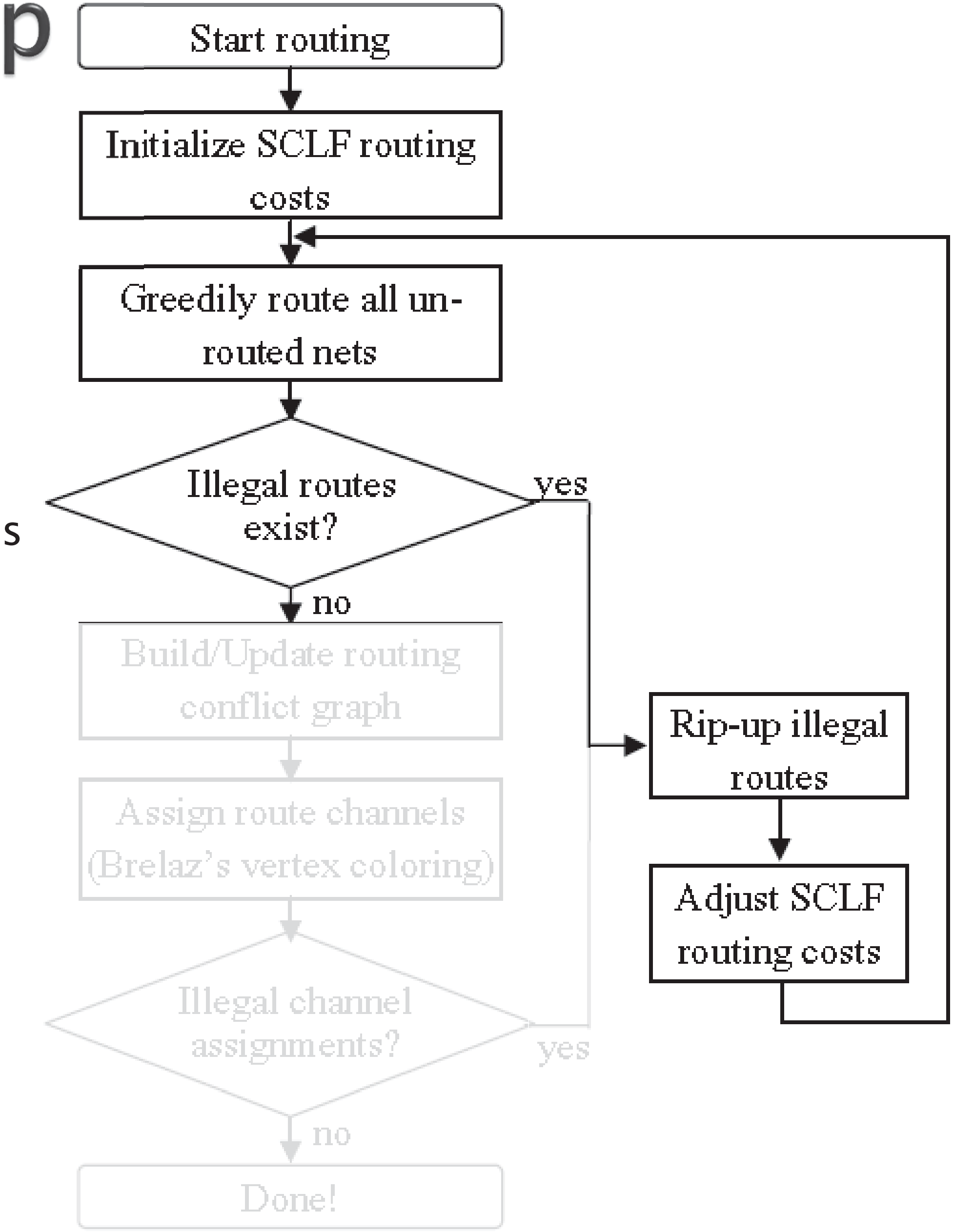
- Logic Synthesis: simplified logic minimizer
- Technology Mapping: represent logic as FPGA-specific LUTs and pack multiple LUTs into CLBs
- Placement: Bind the created CLB-nodes (of the graph/ netlist) to actual CLBs on the FPGA such that communication partners are placed near to each other
- Routing: Connect communication partners



src: [LSV06]

Riverside On-chip Router (ROCR)

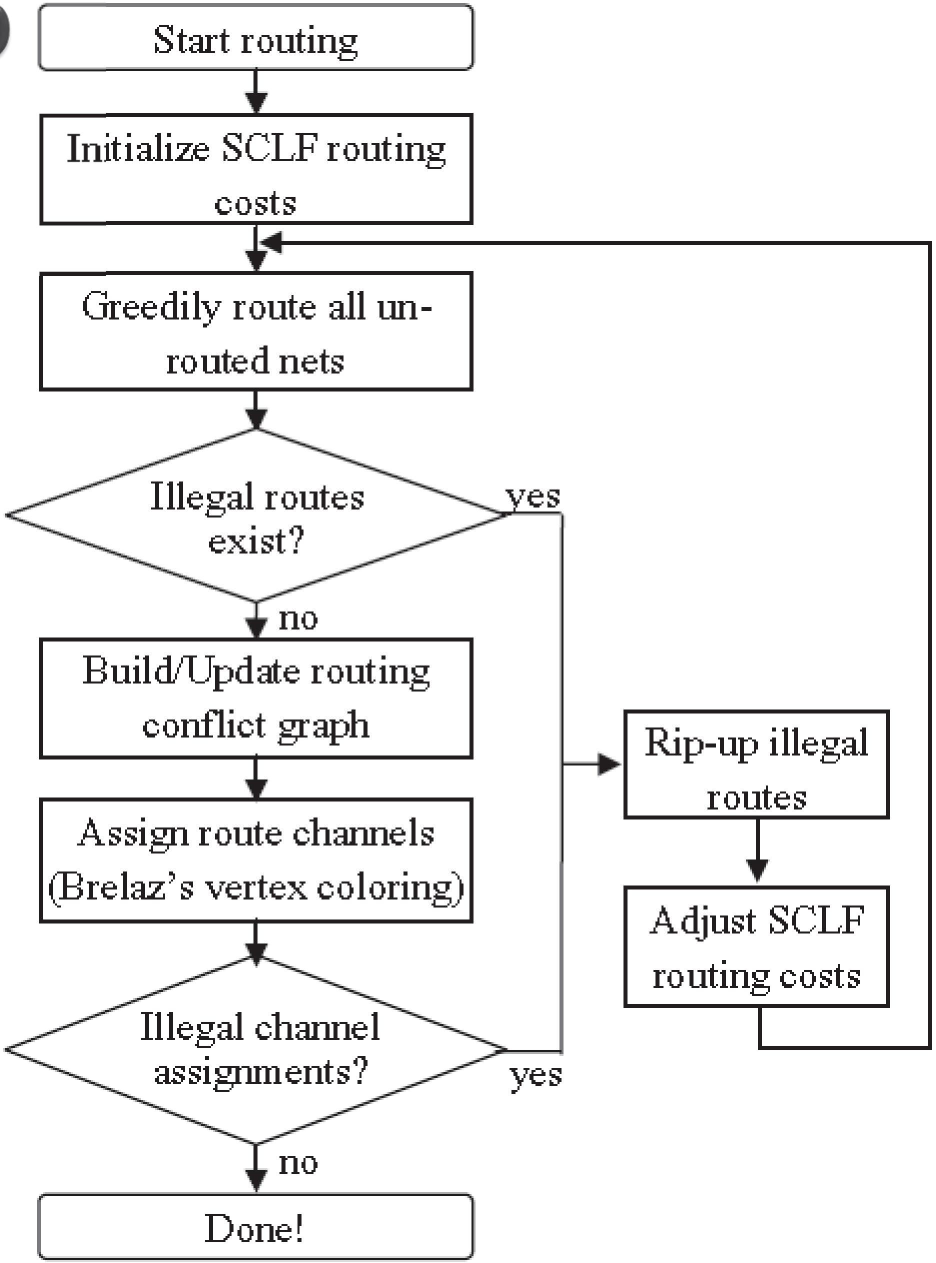
- Simplified routing resource graph
  - Goal: saving memory
  - Two connection types for long and short routing channels
  - Connections annotated with costs
- Top-down approach: greedy assignment of edges to connections
  - Connections contain the actual routing channels
  - The first step does not assign edges to channels but only counts whether sufficient channels would be available
  - Adjust the routing cost for overutilized connections


Start routing
Initialize SCLF routing costs
Greedily route all un-routed nets
Illegal routes exist?
yes
no
Build/Update routing conflict graph
Assign route channels (Brelaz's vertex coloring)
Illegal channel assignments?
yes
no
Done!
Rip-up illegal routes
Adjust SCLF routing costs

src: [LVT05]

Riverside On-chip Router (ROCR)

- Second step: detailed routing, i.e. assigning edges to channels
- Based on a conflict graph
  - Two edges of the routing graph conflict when both routes pass through the same switching matrix
  - The routes (edges) in the routing graph become nodes in the conflict graph that are connected if they have a conflict
- Solved by graph coloring
  - Ensuring that two connected nodes have different colors (corresponds to different channel assignments)

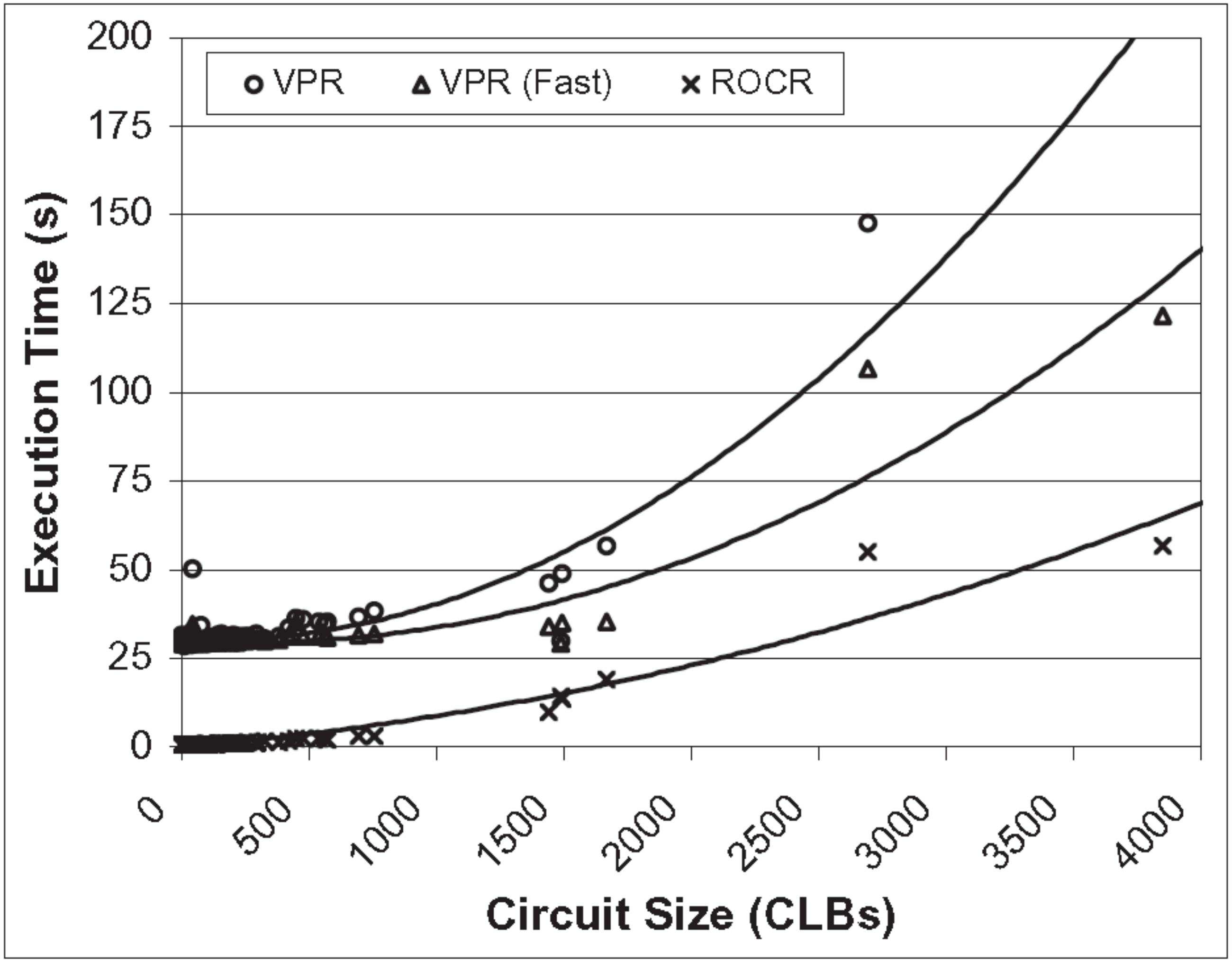


Start routing
Initialize SCLF routing costs
Greedily route all un-routed nets
Illegal routes exist?
yes
no
Build/Update routing conflict graph
Assign route channels (Brelaz's vertex coloring)
Illegal channel assignments?
yes
no
Done!
Rip-up illegal routes
Adjust SCLF routing costs

src: [LVT05]

Results

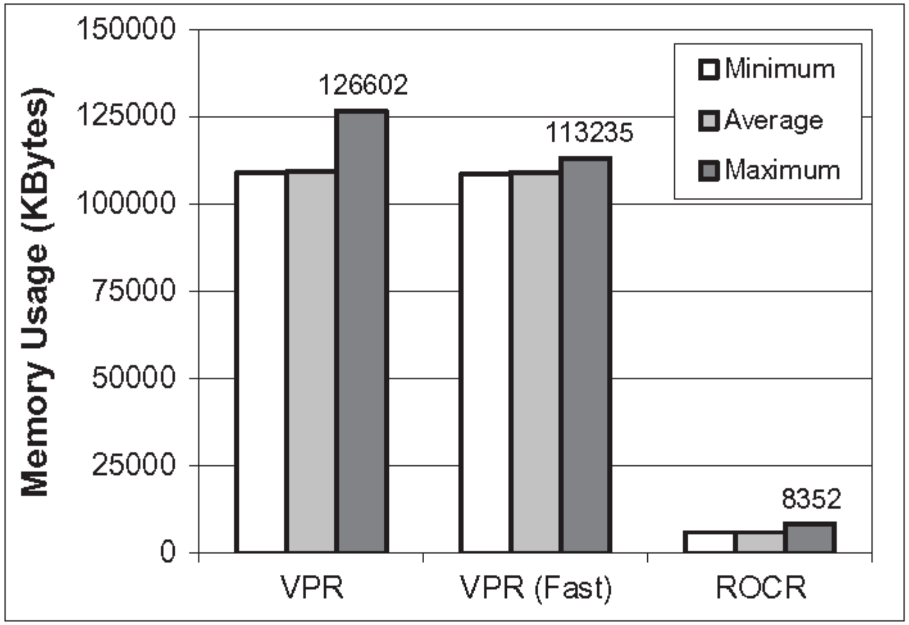
- Comparing scalability with a standard router (VPR) in normal mode and in fast mode
  - Executed on a 1.6 GHz Pentium
- Routing different algorithms for a 100x100 CLB array
  - Note: low array utilization!

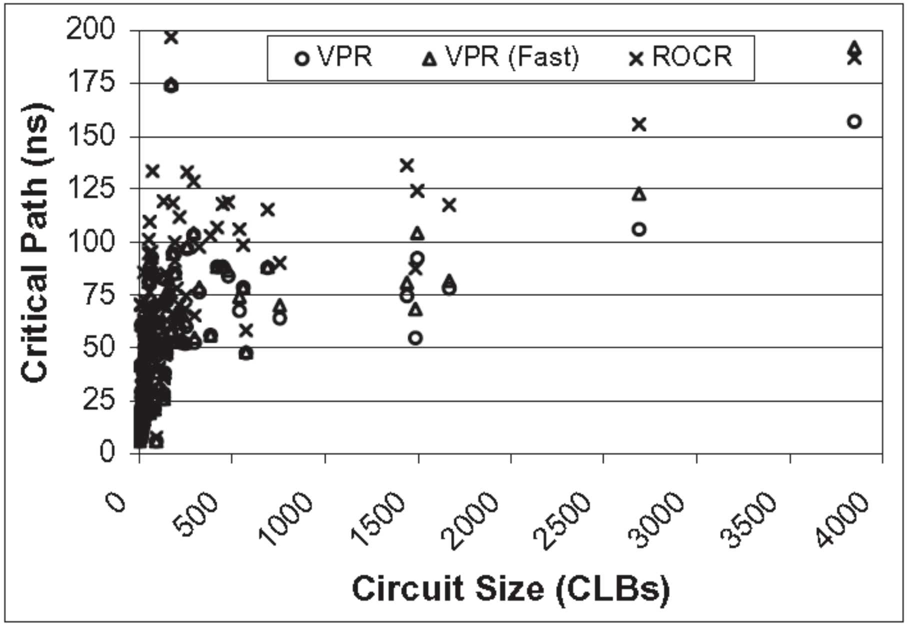


src: [LVT05]

Results (cont'd)

- Significantly reduced memory requirements (at most 8 MB; allows for execution on embedded CPUs)
- Slower critical path (30%)
  - Not clear, how it would perform for higher FPGA utilization





src: [LVT05]

Warp Summary

- No effort for Application developers
  - Works on existing application binaries
- High speedup possible for small kernels (after online synthesis is completed)
- But: some applications are hard to optimize
  - Code is not restructured by Warp tools to separate between HW-accelerated parts and software parts
  - Interface must be derived automatically
- Optimization takes rather long due to online synthesis
  - From seconds to minutes for the router running on a 1.6 GHz Pentium and correspondingly longer on an embedded ARM (i.e. the actual target on which they wanted to execute their online synthesis)
- Altogether: interesting approach that demonstrates high flexibility (targeting different applications but not within an application or across multiple applications) and that provides a new trade-off between flexibility, programmer/compiler effort, and efficiency

7.3 Dynamic Instruction Merging (DIM)

DIM Overview

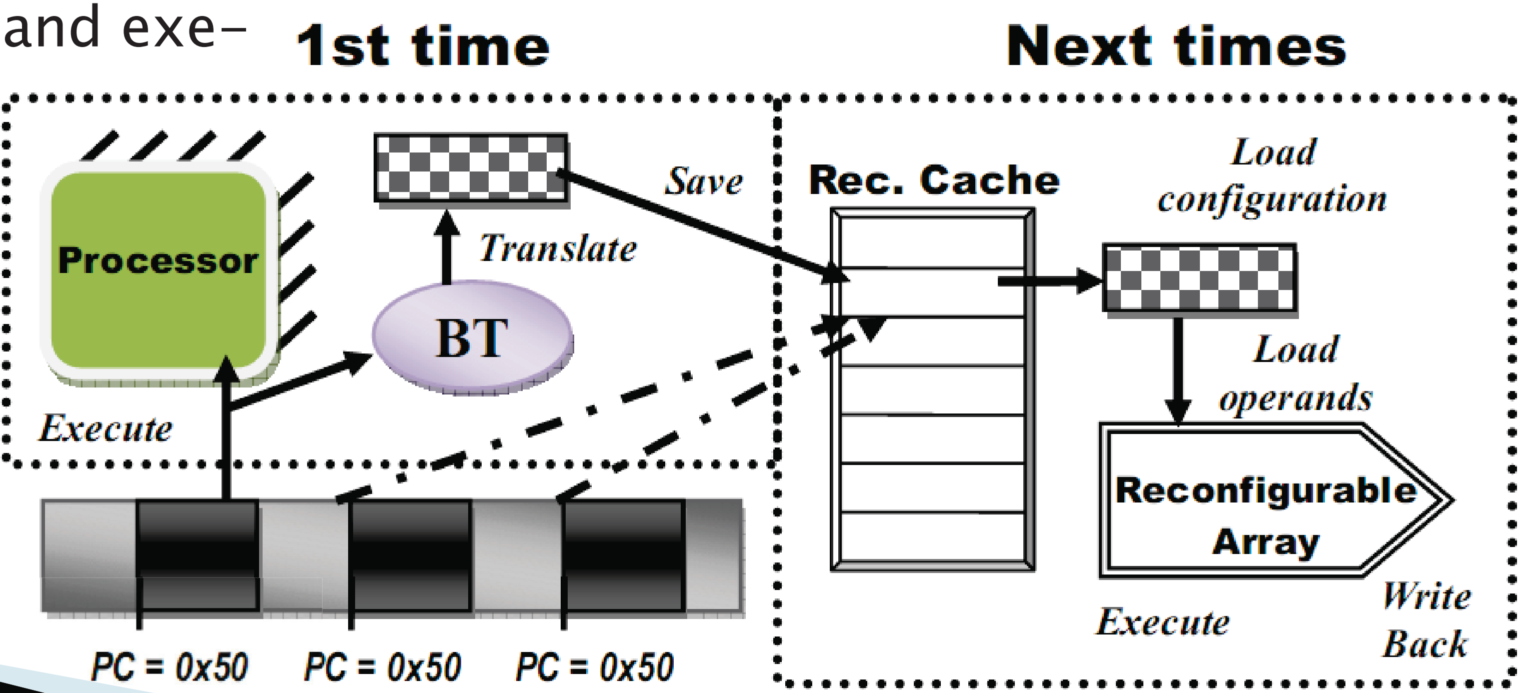
- Tightly-coupled coarse-grained architecture
- No compiler required; works on standard binaries
- On-the-fly online-synthesis
    - i.e. no lengthy synthesis algorithms
    - creation of the Special Instructions during execution of the original instructions
- Caching of the created SIs

Binary Translation (BT)

- Starts on the first instruction after a branch
- Stops when it detects an unsupported instruction or another branch (unless ‘speculative execution’ is supported, i.e. speculating on the branch)
- In between: each executed instruction is placed on the reconfigurable array
  - Creating a configuration on-the-fly and extending it by each executed assembler instruction
  - Using several temporary tables to manage utilized resources, data dependencies etc.
- If more than three instructions were found, the created configuration is cached

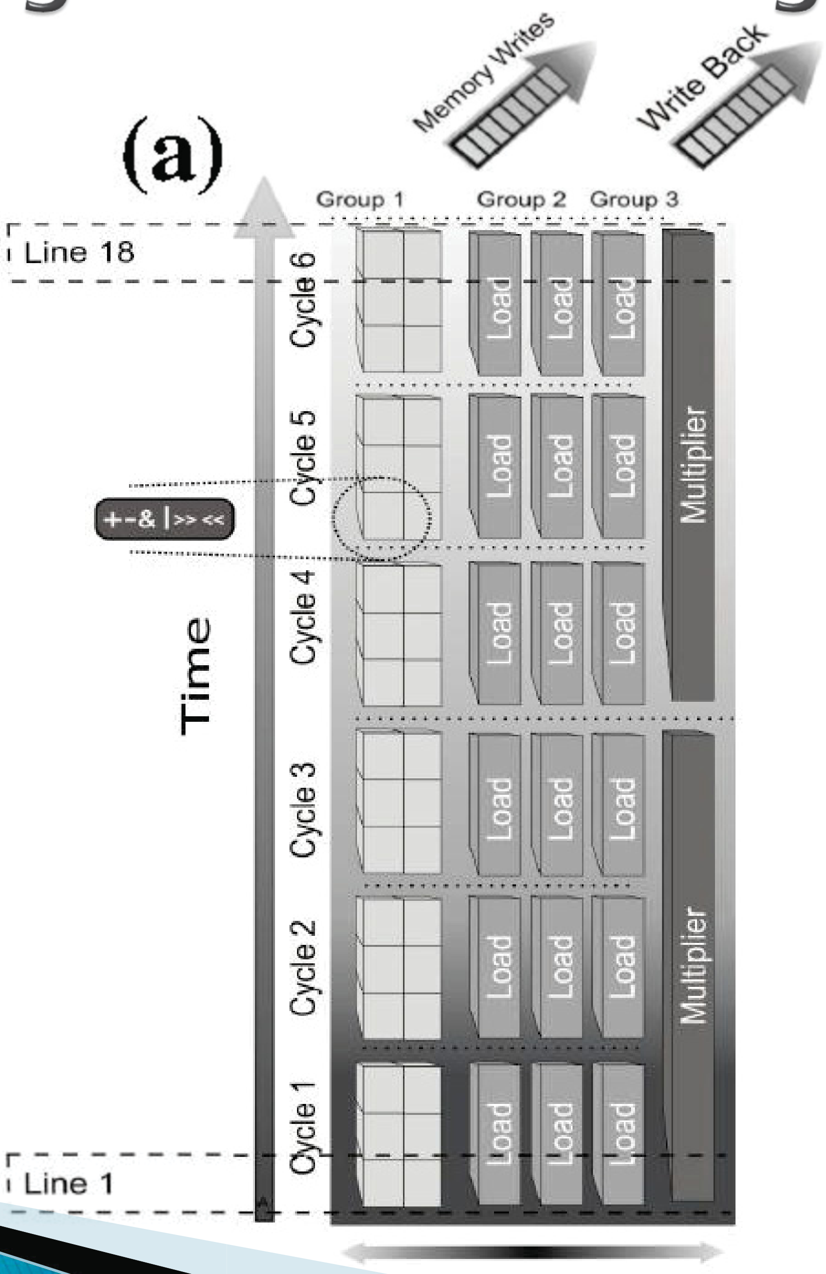
BT Overview

- First time a hot spot (dark grey) is executed, it is translated into a configuration, i.e. SI
  - It is not necessarily known, that it is a hot spot; but ‘hotter’ spots have a higher chance to remain in the cache
- For subsequent executions, the cached configuration is loaded and exe-cuted



src: [BRGC08]

Coarse-grained Reconfigurable Array



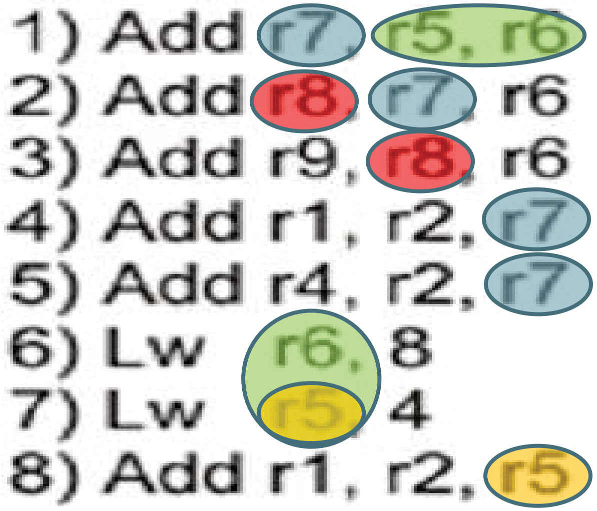
src: [RBC08]

Coarse-grained Reconfigurable Array (cont'd)

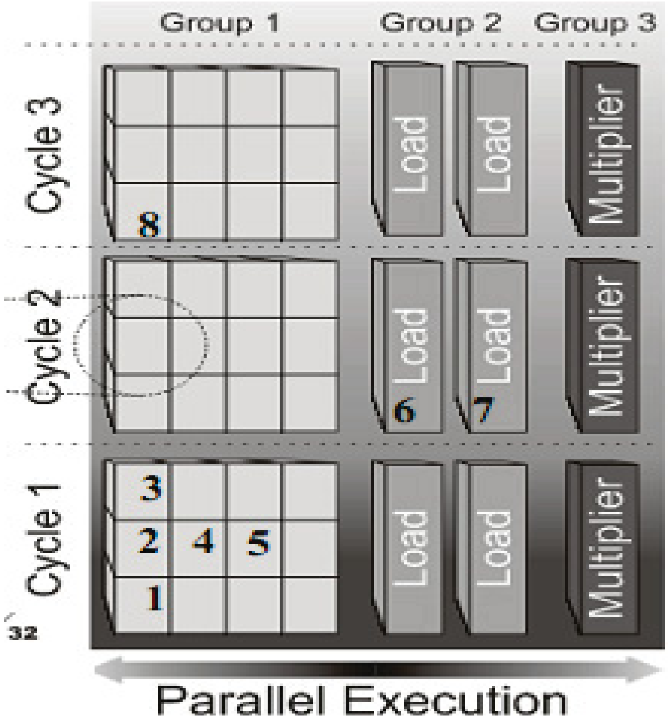
- The array is composed of different building blocks
  - ALUs, Load/Store Units, Multipliers
- Lines of these building blocks are connected to subsequent lines, using multiplexers
  - Note: the previous example does not necessarily have 18 physical lines; it rather has 3 physical lines; Line 4 reuses the hardware of Line 1
  - But: configuration memory for all lines is needed to switch the configuration while the Special Instruction executes
- At design time, different (application specific) reconfigurable fabrics can be composed

Example

dst-reg



```
1) Add r7, r5, r6
2) Add r8, r7, r6
3) Add r9, r8, r6
4) Add r1, r2, r7
5) Add r4, r2, r7
6) Lw  r6, 8
7) Lw  r5, 4
8) Add r1, r2, r5
```

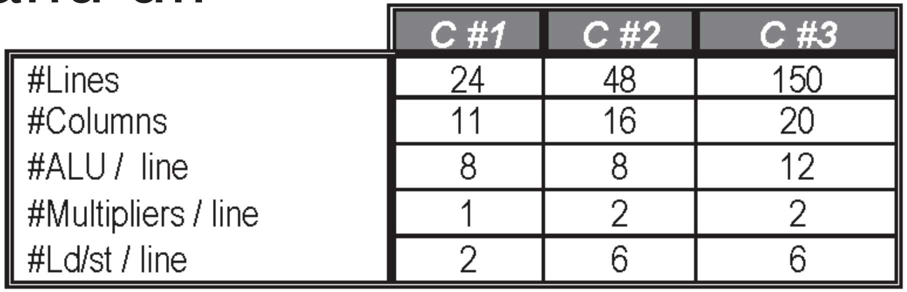


- Creating the configuration step-by-step
- Considering dependencies

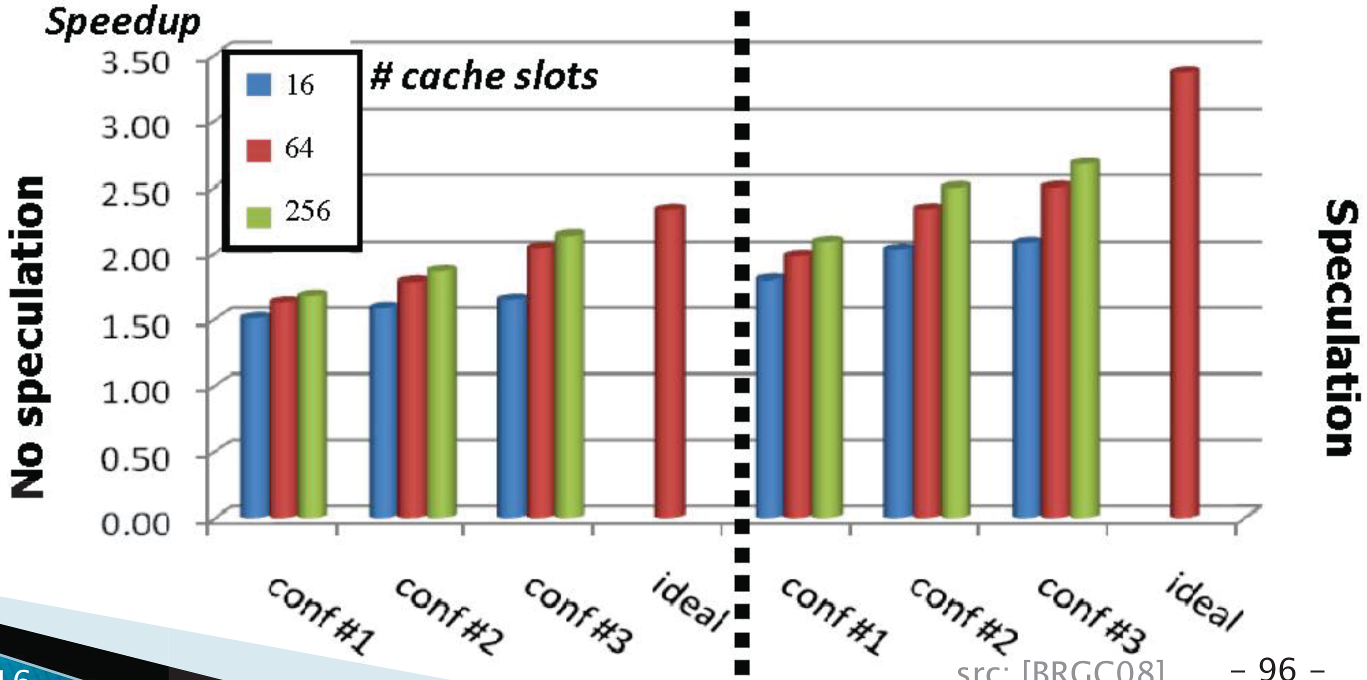
src: [RBC08]

Results

- Average Speedup for different Configurations of the reconfigurable array and different Cache sizes for the configuration data
- "Ideal" assumes infinite hardware
- "Speculation" allows speculative execution



	C #1	C #2	C #3
#Lines	24	48	150
#Columns	11	16	20
#ALU / line	8	8	12
#Multipliers / line	1	2	2
#Ld/st / line	2	6	6



src: [BRGC08]

DIM summary

- Efficient way to support online synthesis on-the-fly
- Moderate speedups
  - Also depends on how the compiler schedules the code
  - Limited room for optimizations when creating a configuration on-the-fly
- Application-specific reconfigurable fabrics provide higher speedup for the targeted application at the cost of reduced generality

References and Sources

[BSH08a] L. Bauer, M. Shafique, J. Henkel: "A Computation- and Communication-Infrastructure for Modular Special Instructions in a Dynamically Reconfigurable Processor", International Conference on Field Programmable Logic and Applications (FPL), pp. 203-208, 2008.

[BSKH08] L. Bauer, M. Shafique, S. Kreutz, J. Henkel: "Run-time System for an Extensible Embedded Processor with Dynamic Instruction Set", Design Automation and Test in Europe Conference (DATE), pp. 752-757, 2008.

[BSH08b] L. Bauer, M. Shafique, J. Henkel: "Run-time Instruction Set Selection in a Transmutable Embedded Processor", Design Automation Conference (DAC), pp. 56-61, 2008.

[BSH09] L. Bauer, M. Shafique, J. Henkel: "MinDeg: A Performance-guided Replacement Policy for Run-time Reconfigurable Accelerators", Int'l Conference on Hardware-Software Codesign and System Synthesis (CODES+ISSS), pp. 335-342, 2009.

[BSH08c] L. Bauer, M. Shafique, J. Henkel: "Efficient Resource Utilization for an Extensible Processor through Dynamic Instruction Set Adaptation", IEEE Transaction on Very Large Scale Integration (TVLSI) , vol. 16, no. 10, pp. 1295-1308, 2008.

References and Sources

[LSV06] R. Lysecky, G. Stitt, F. Vahid: “Warp Processors”, ACM Transactions on Design Automation of Electronic Systems (TODAES), vol. 11, no. 3, pp. 659-681, 2006.

[GV03] A. Gordon-Ross, F. Vahid: “Frequent Loop Detection Using Efficient Non-Intrusive On-Chip Hardware”, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), pp. 117-124, 2003.

[LVT05] R. Lysecky, F. Vahid, S. X.-D. Tan: “A Study of the Scalability of On-Chip Routing for Just-in-Time FPGA compilation”, IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 57-62, 2005.

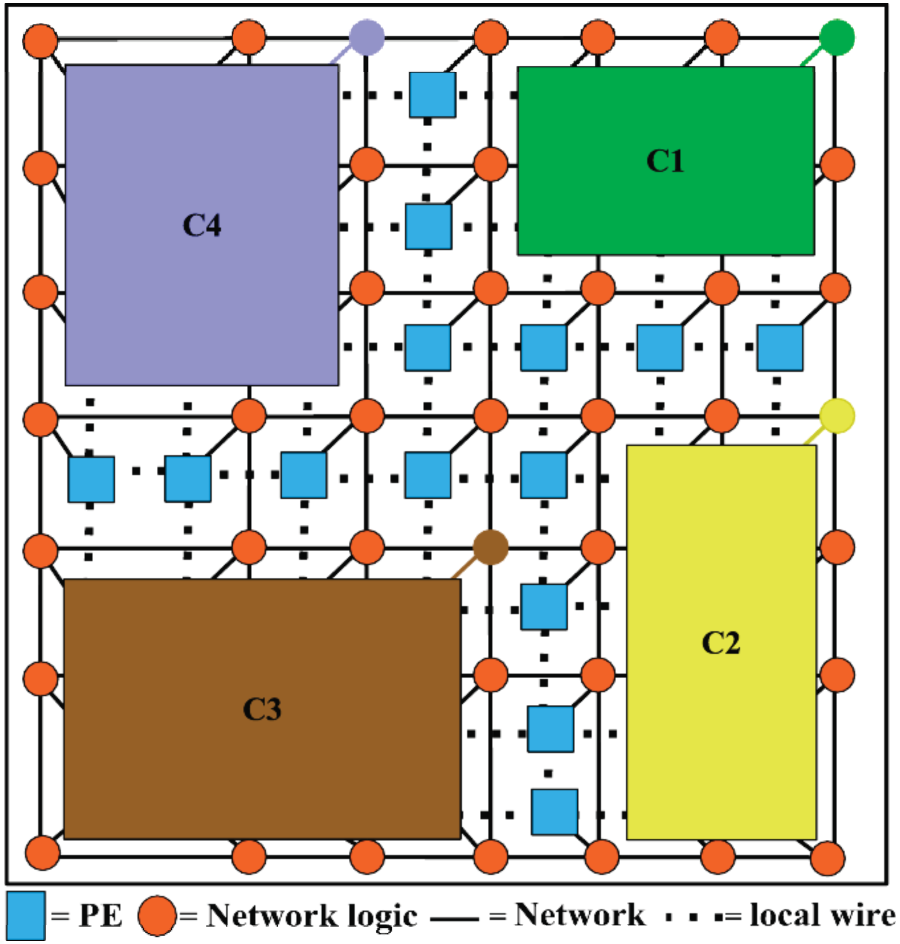
[BRGC08] A.C.S. Beck, M.B. Rutzig, G. Gaydadjiev, L. Carro: “Transparent reconfigurable acceleration for heterogeneous embedded applications”, Design Automation and Test in Europe Conference (DATE), pp. 1208-1213, 2008.

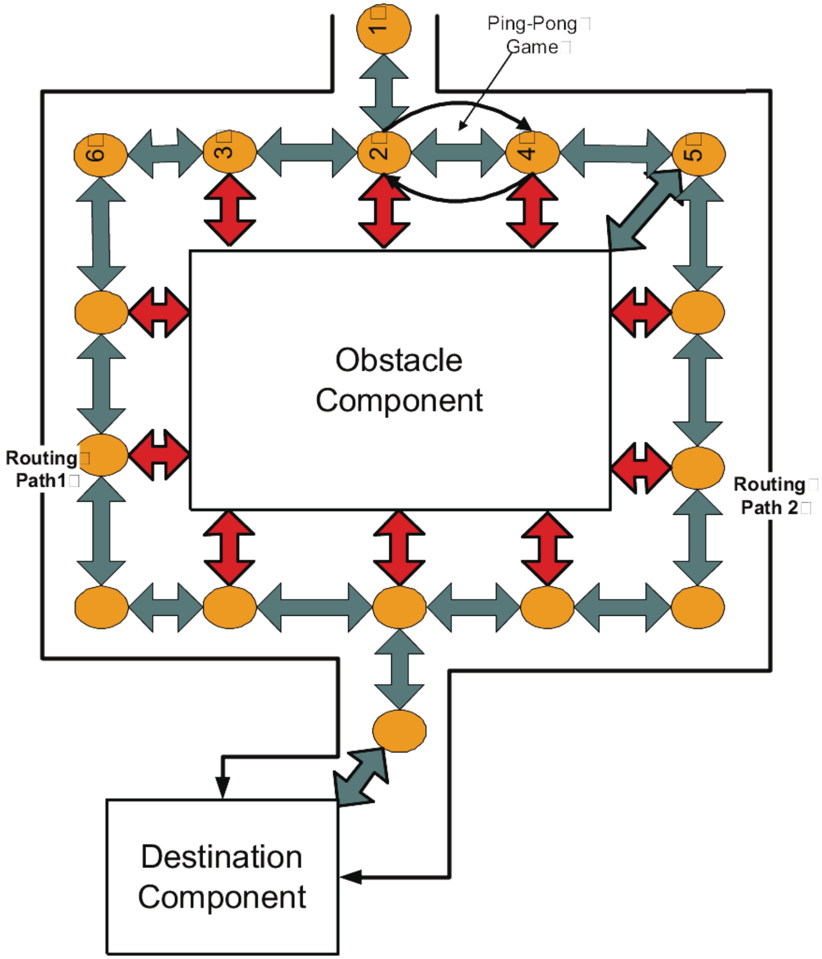
[RBC08] M.B. Rutzig, A.C.S. Beck, L. Carro: “Balancing reconfigurable data path resources according to application requirements”, International Parallel and Distributed Processing Symposium, pp. 1-8, 2008.

7.4 Further relevant architectures / domains

(not relevant for exam)

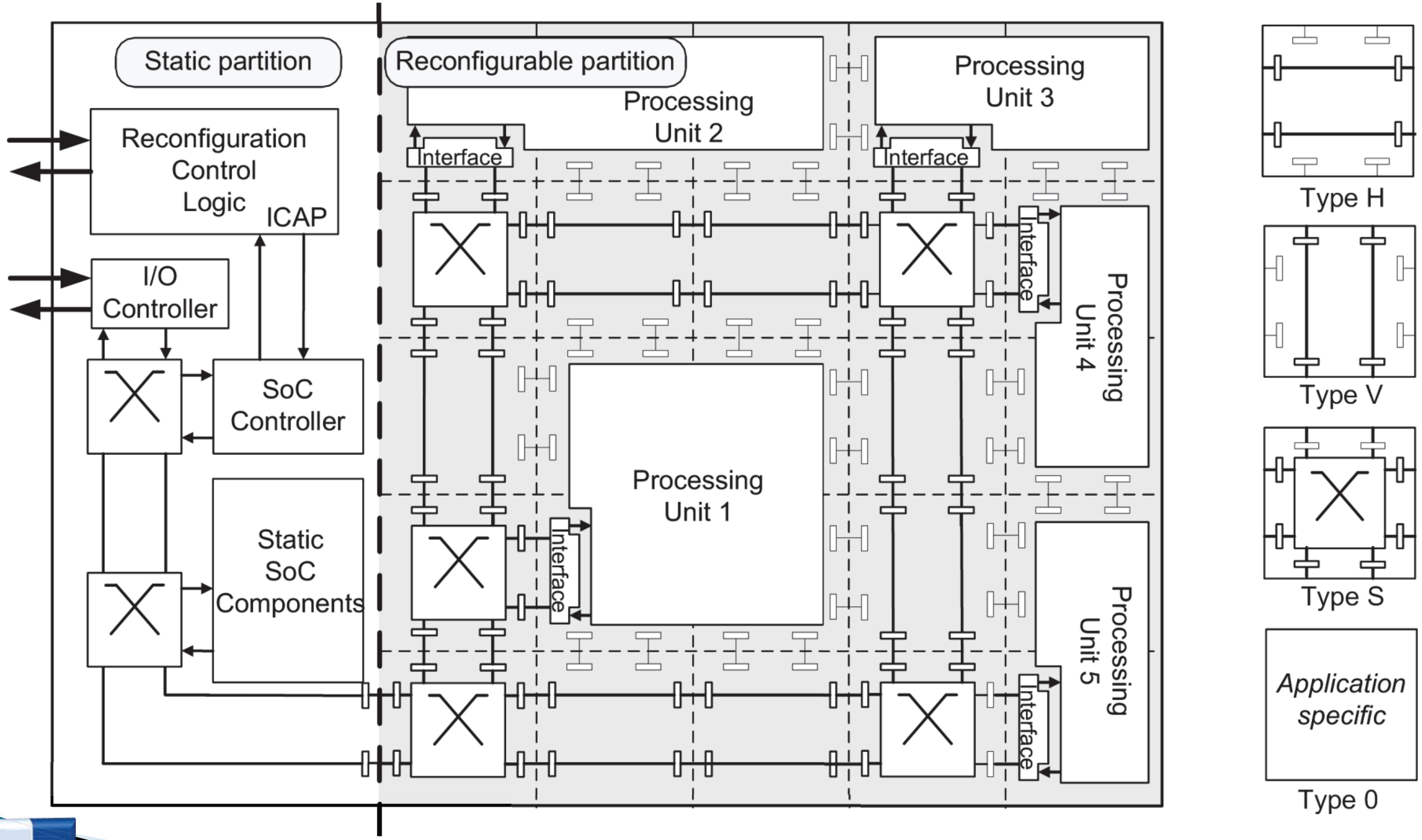
7.4.1 Dynamic Network-on-Chip (DyNoC)





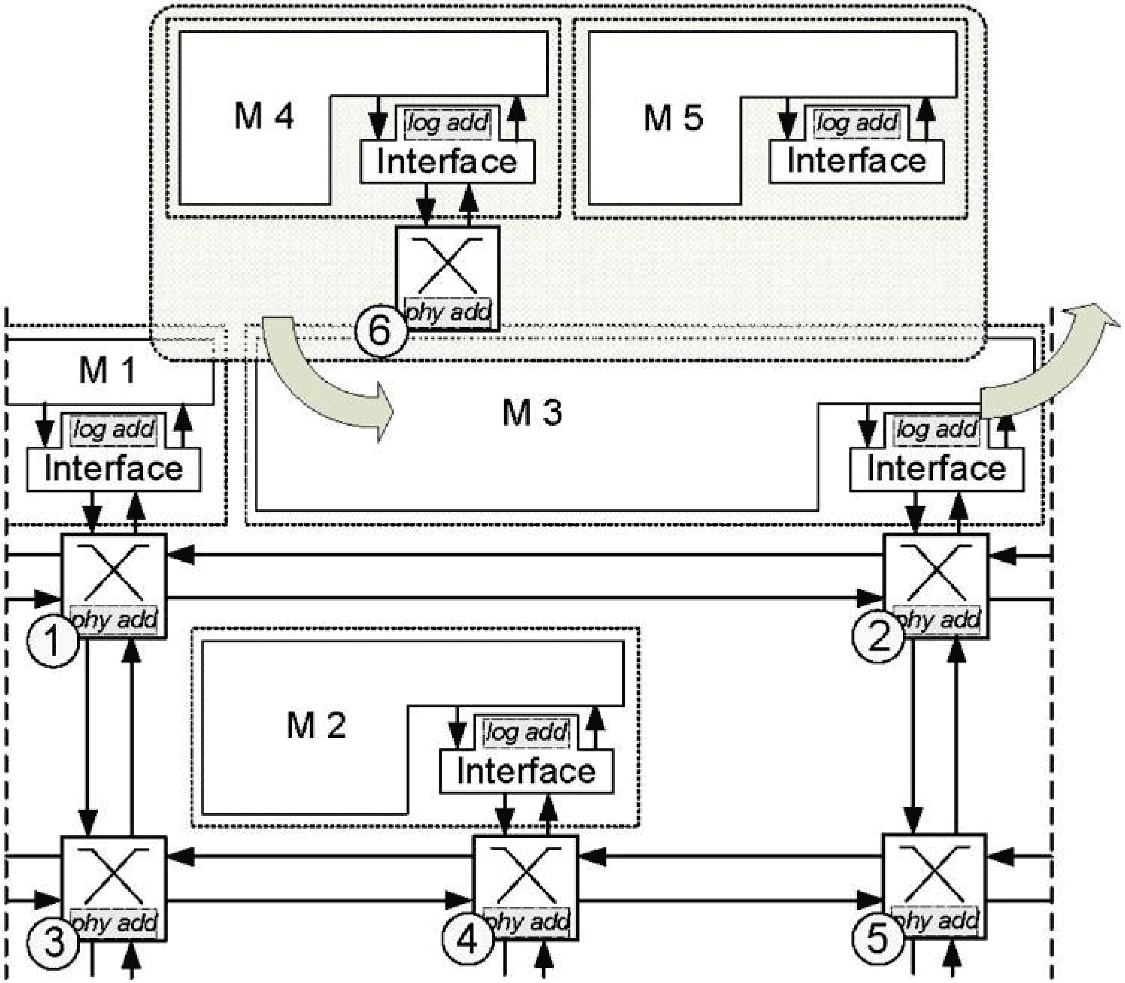
src: C. Bobda et al. "DyNoC: A Dynamic Infrastructure for Communication in Dynamically Reconfigurable Devices", IEEE Design & Test of Computers, 22(5), pp. 443-451, 2005.

7.4.2 Configurable NoC: CoNoChi



src: T. Pionteck et al. “A Design Technique for Adapting Number and Boundaries of Reconfigurable Modules at Runtime”, Int’l Journal of Reconfigurable Computing, 2009.

Configurable NoC: CoNoChi

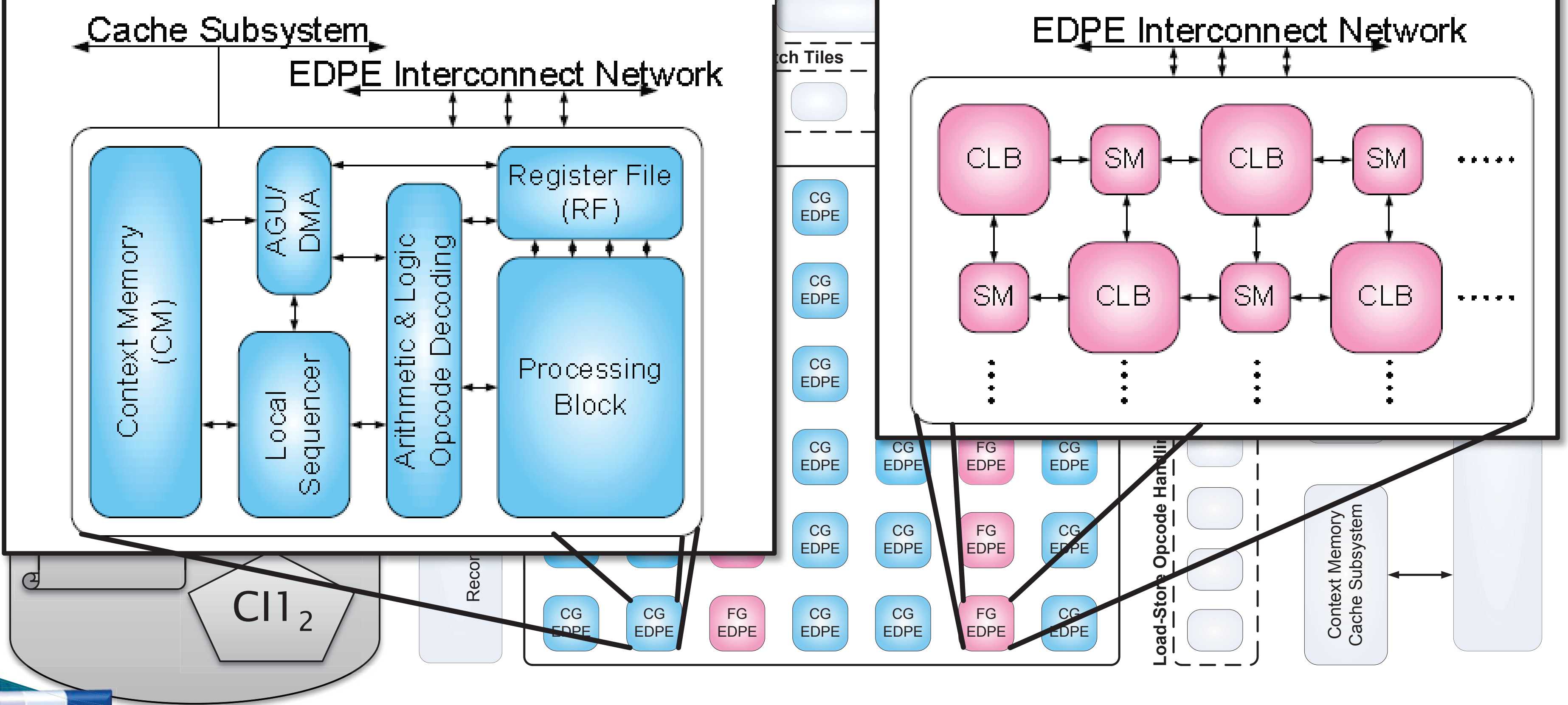


src: T. Pionteck et al. “A Design Technique for Adapting Number and Boundaries of Reconfigurable Modules at Runtime”, Int’l Journal of Reconfigurable Computing, 2009.

7.4.3 KAHRISMA

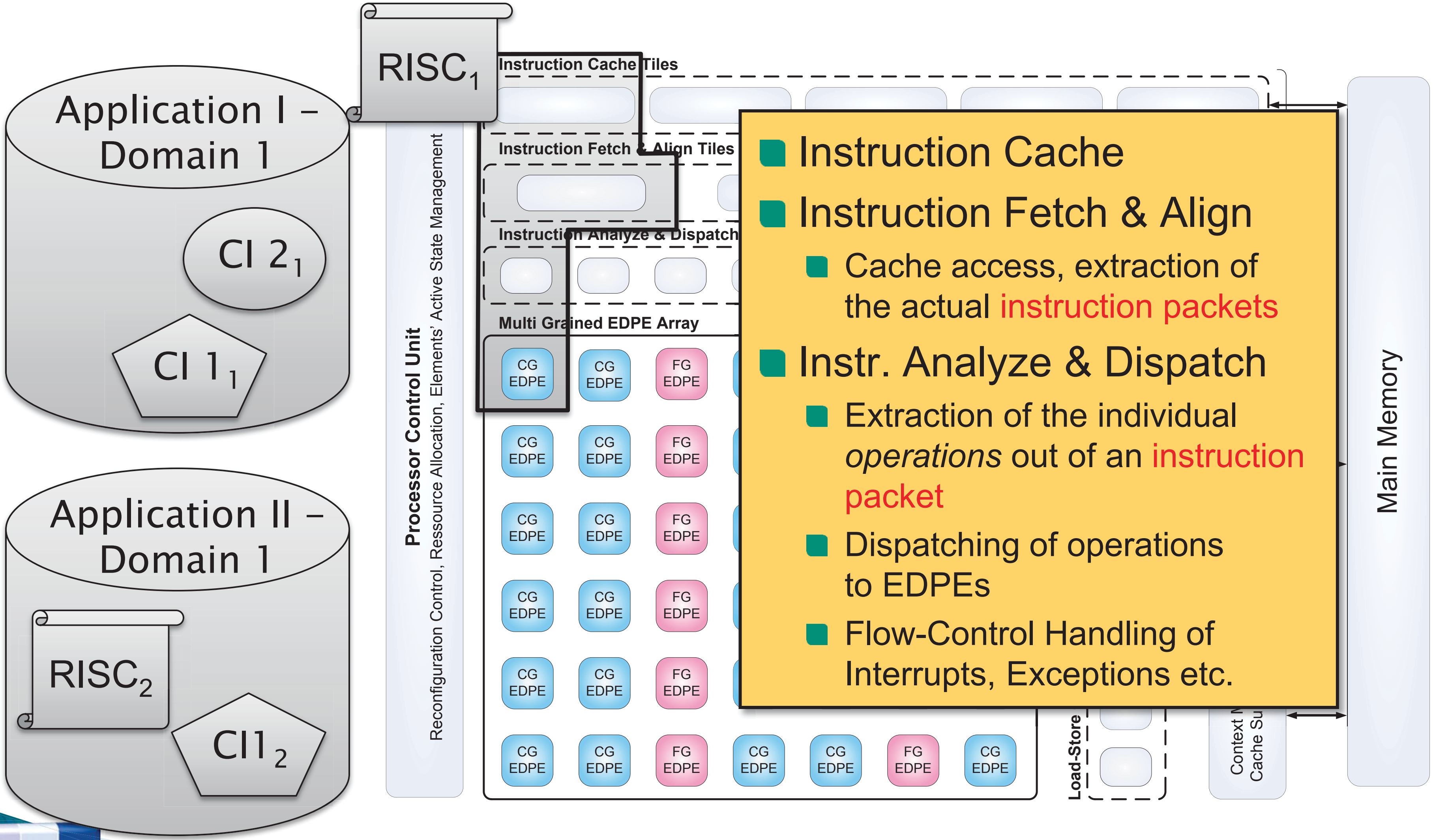
CG-EDPEs are ALU-like reconfigurable fabrics, optimized for word/sub-word level operations

FG-EDPEs are FPGA-like reconfigurable fabrics, optimized for bit/byte level operations, state machines etc.



src: R. Koenig et al. “KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture”, Design Automation and Test in Europe Conference (DATE), pp. 819-824, 2009.

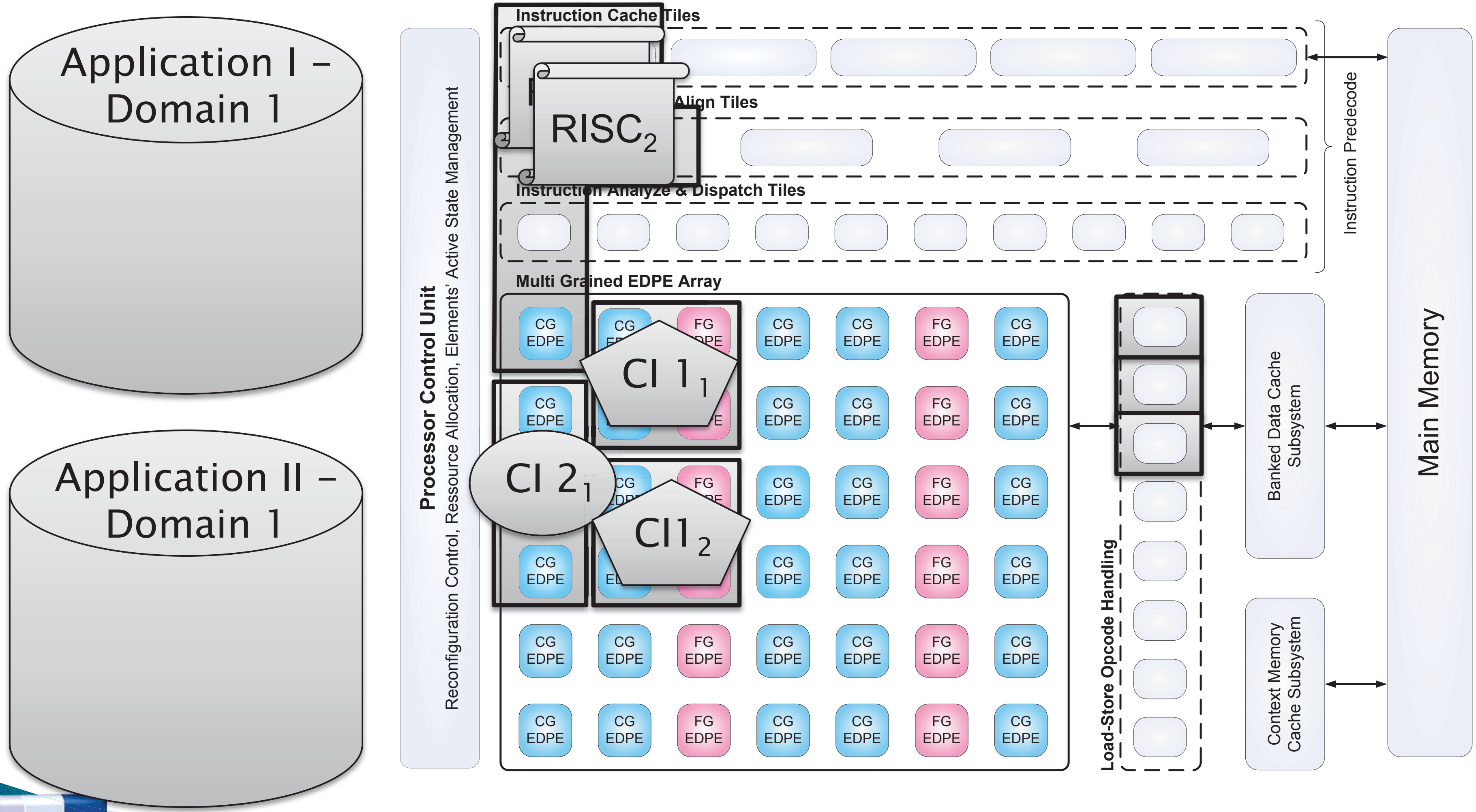
KAHRISMA



- Instruction Cache
- Instruction Fetch & Align
  - Cache access, extraction of the actual instruction packets
- Instr. Analyze & Dispatch
  - Extraction of the individual *operations* out of an instruction packet
  - Dispatching of operations to EDPEs
  - Flow-Control Handling of Interrupts, Exceptions etc.

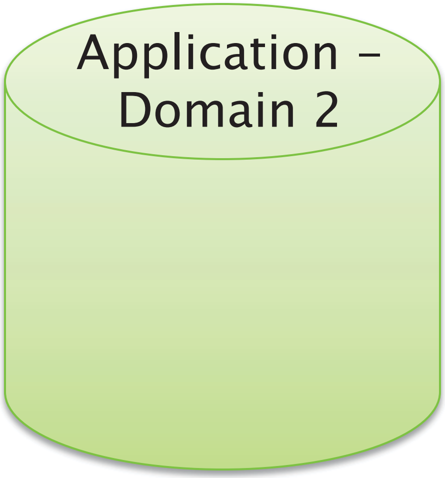
src: R. Koenig et al. "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture", Design Automation and Test in Europe Conference (DATE), pp. 819-824, 2009.

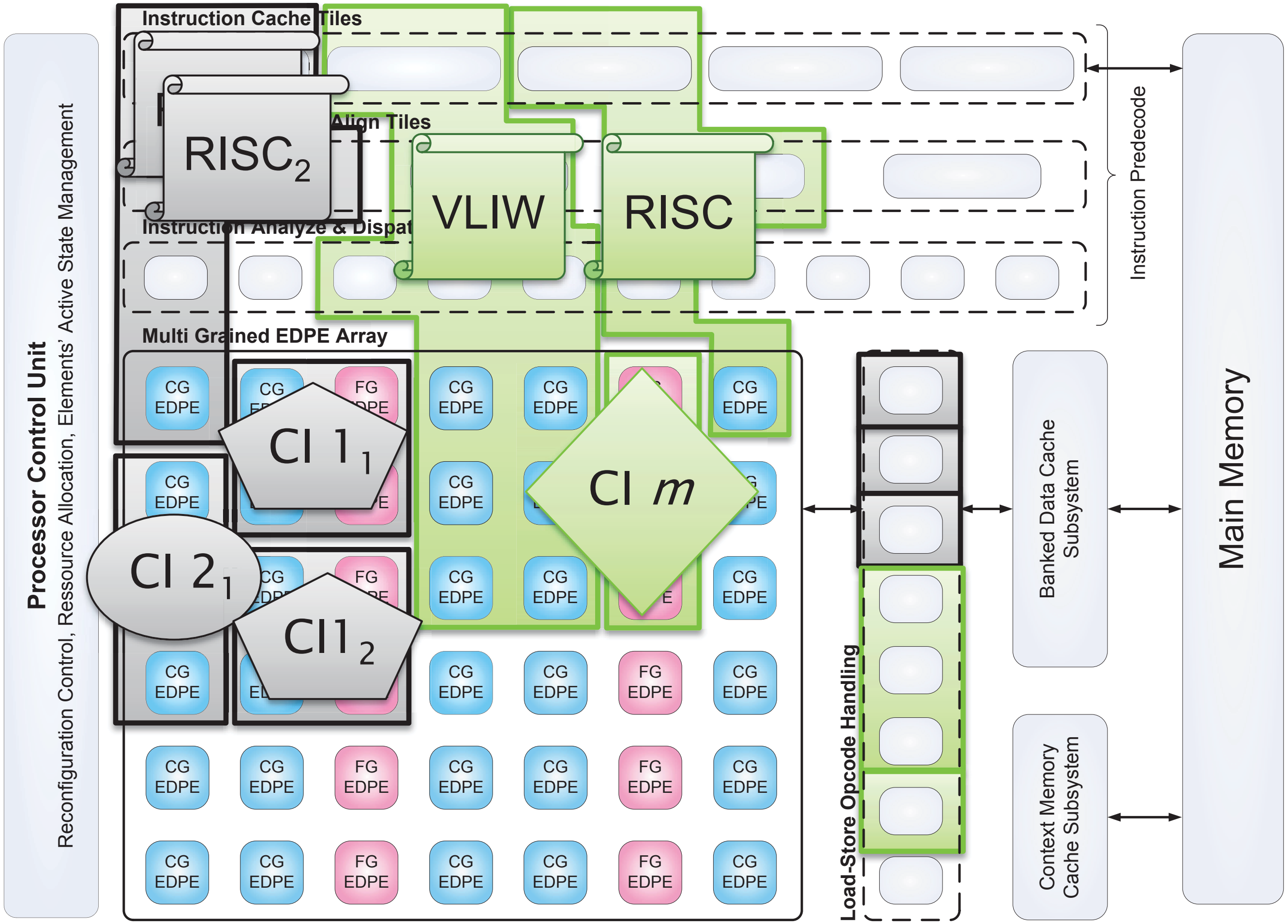
KAHRISMA


Application I – Domain 1
Application II – Domain 1
Processor Control Unit
Reconfiguration Control, Ressource Allocation, Elements' Active State Management
Instruction Cache Tiles
RISC$_2$
Align Tiles
Instruction Analyze & Dispatch Tiles
Instruction Predecode
Multi Grained EDPE Array
CG EDPE
FG EDPE
CI 1$_1$
CI 2$_1$
CI1$_2$
Load-Store Opcode Handling
Banked Data Cache Subsystem
Context Memory Cache Subsystem
Main Memory

src: R. Koenig et al. "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture", Design Automation and Test in Europe Conference (DATE), pp. 819-824, 2009.

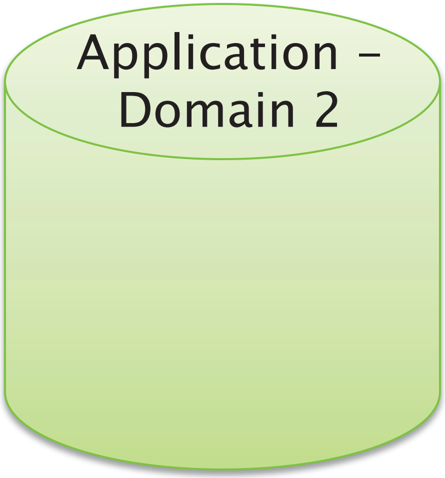
KAHRISMA

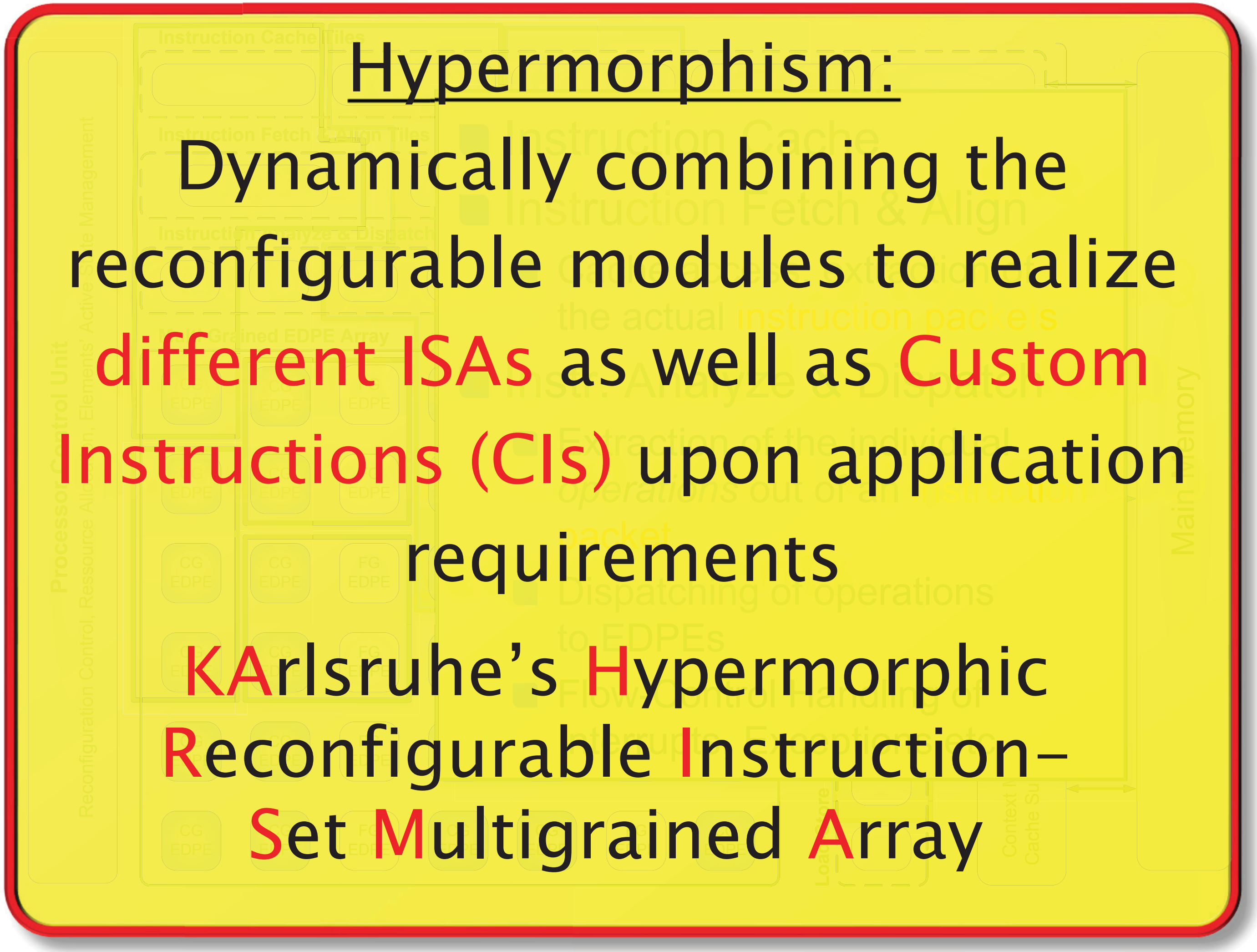



Instruction Cache Tiles
Align Tiles
Instruction Analyze & Dispatch
Multi Grained EDPE Array
RISC₂
VLIW
RISC
CI 1₁
CI 2₁
CI1₂
CI m
CG EDPE
FG EDPE
Processor Control Unit
Reconfiguration Control, Ressource Allocation, Elements' Active State Management
Load-Store Opcode Handling
Banked Data Cache Subsystem
Context Memory Cache Subsystem
Instruction Predecode
Main Memory

src: R. Koenig et al. "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture", Design Automation and Test in Europe Conference (DATE), pp. 819-824, 2009.

KAHRISMA




Instruction Cache Tiles
Instruction Fetch & Align Tiles
Instruction Analyze & Dispatch
Grained EDPE Array
CG EDPE
FG EDPE
Processor Control Unit
Reconfiguration Control, Ressource Allocation, Elements' Active Management
Main Memory
Context Cache

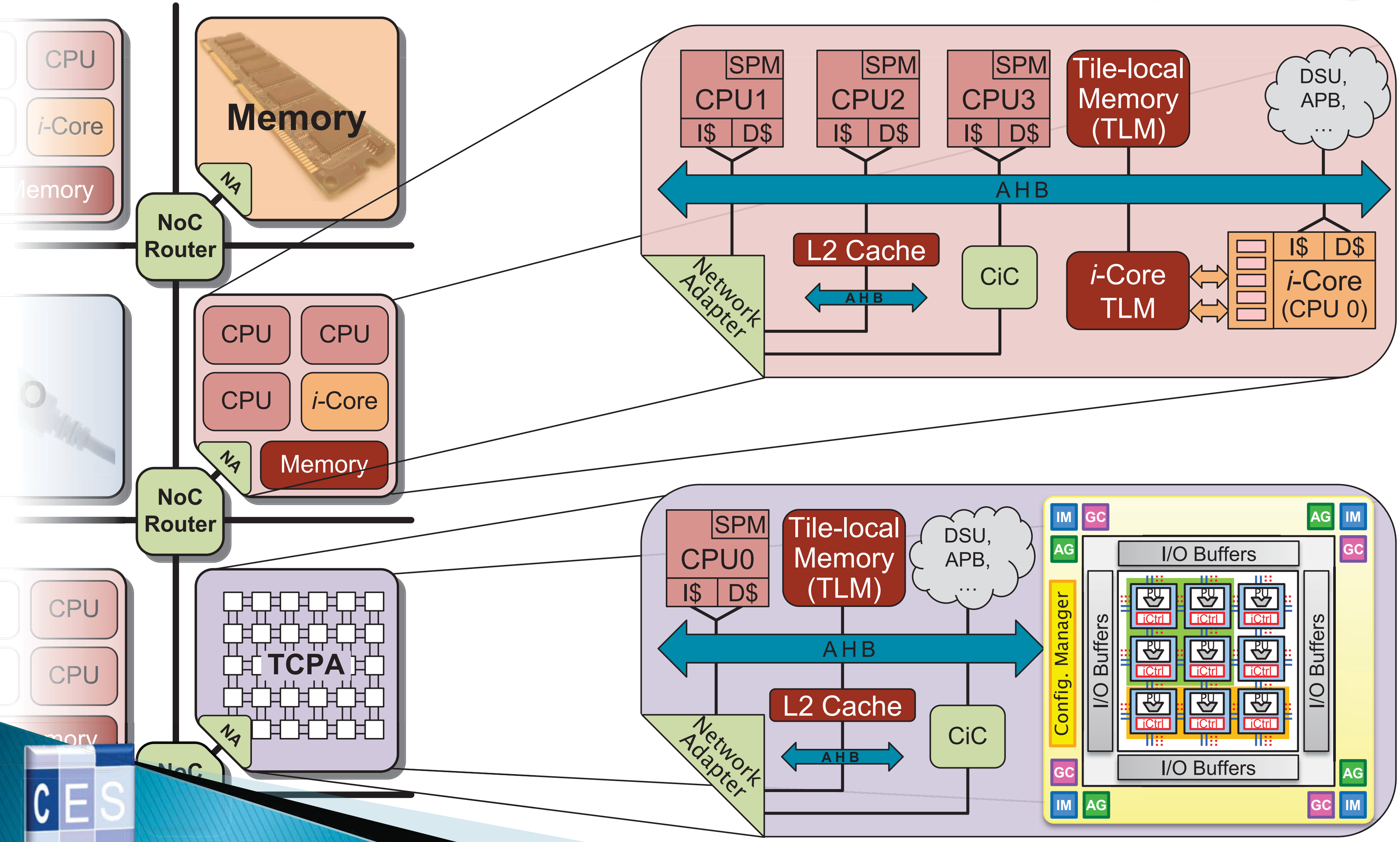
Hypermorphism:

Dynamically combining the reconfigurable modules to realize different ISAs as well as Custom Instructions (CIs) upon application requirements

KArlsruhe's Hypermorphic Reconfigurable Instruction-Set Multigrained Array

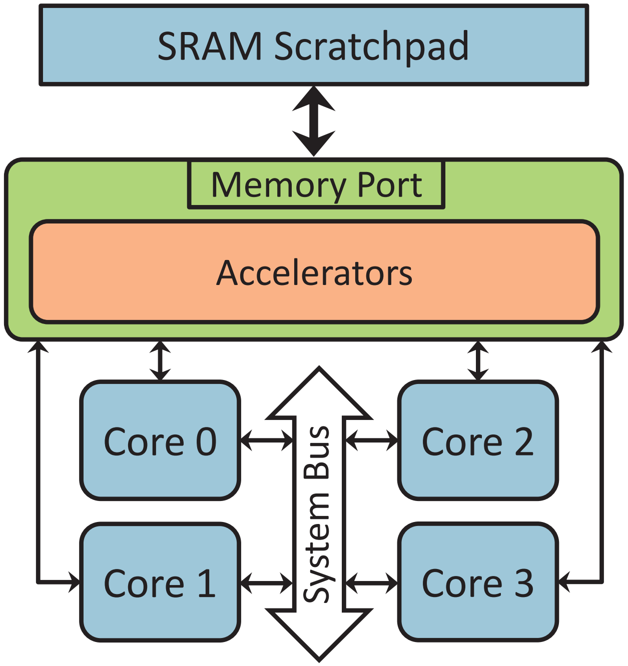
src: R. Koenig et al. "KAHRISMA: A Novel Hypermorphic Reconfigurable-Instruction-Set Multi-grained-Array Architecture", Design Automation and Test in Europe Conference (DATE), pp. 819-824, 2009.

7.4.4 Reconfigurable Multi-Core Architecture in "Invasive Computing"



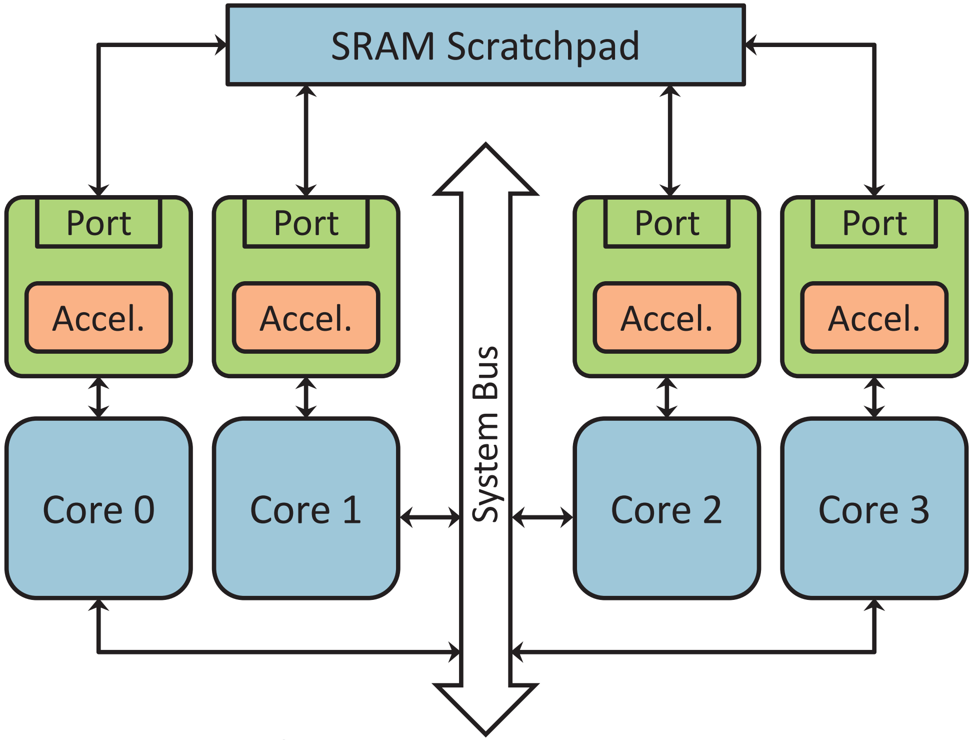
Previous Reconfigurable Multi-Core Architectures

Shared fabric:



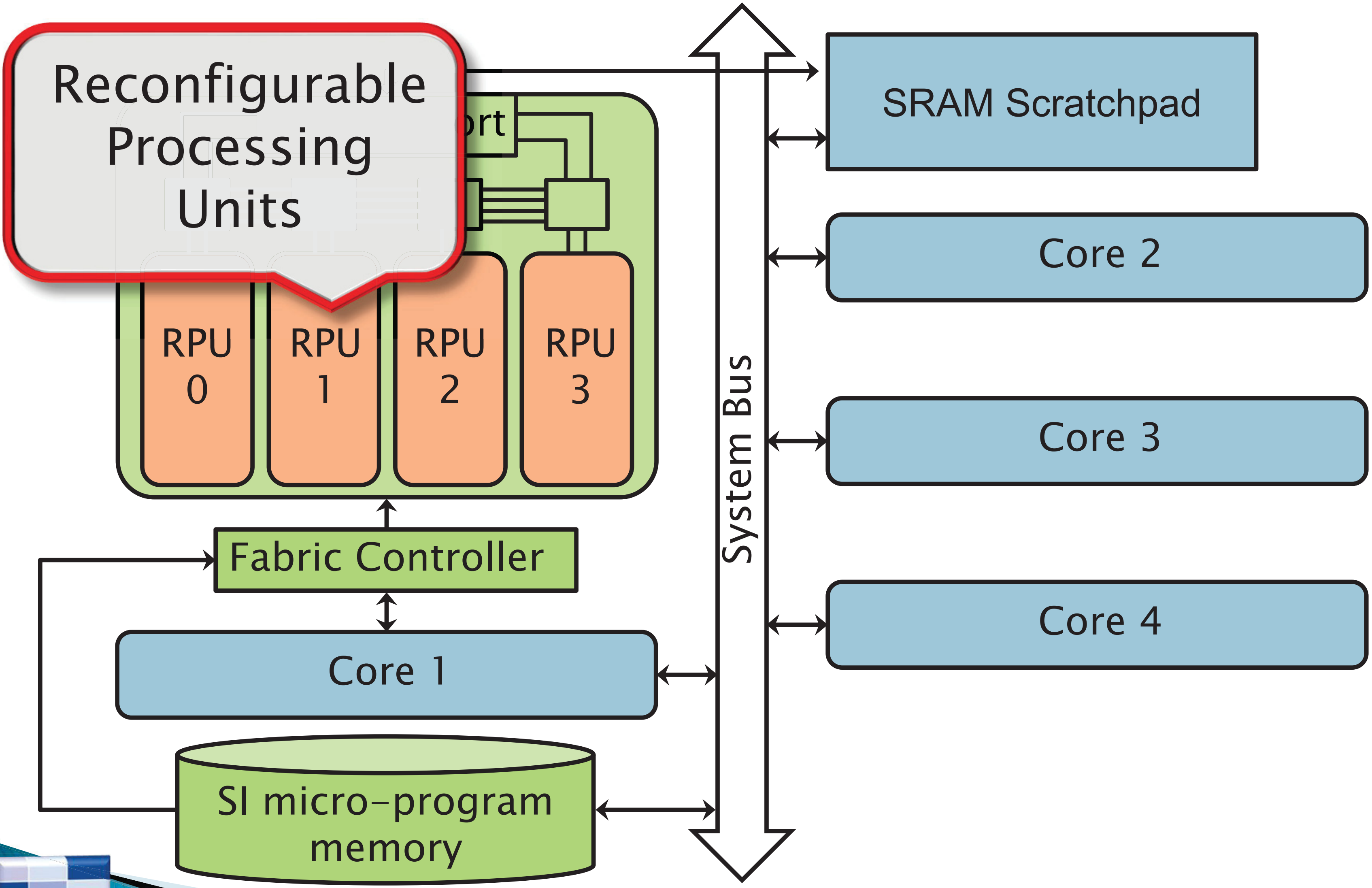
- Shared reconfigurable fabric, e.g. [Chen@DAC11]
- **Problem**: Only 1 kernel can be run on the fabric at any time

Spatial Partitioning:

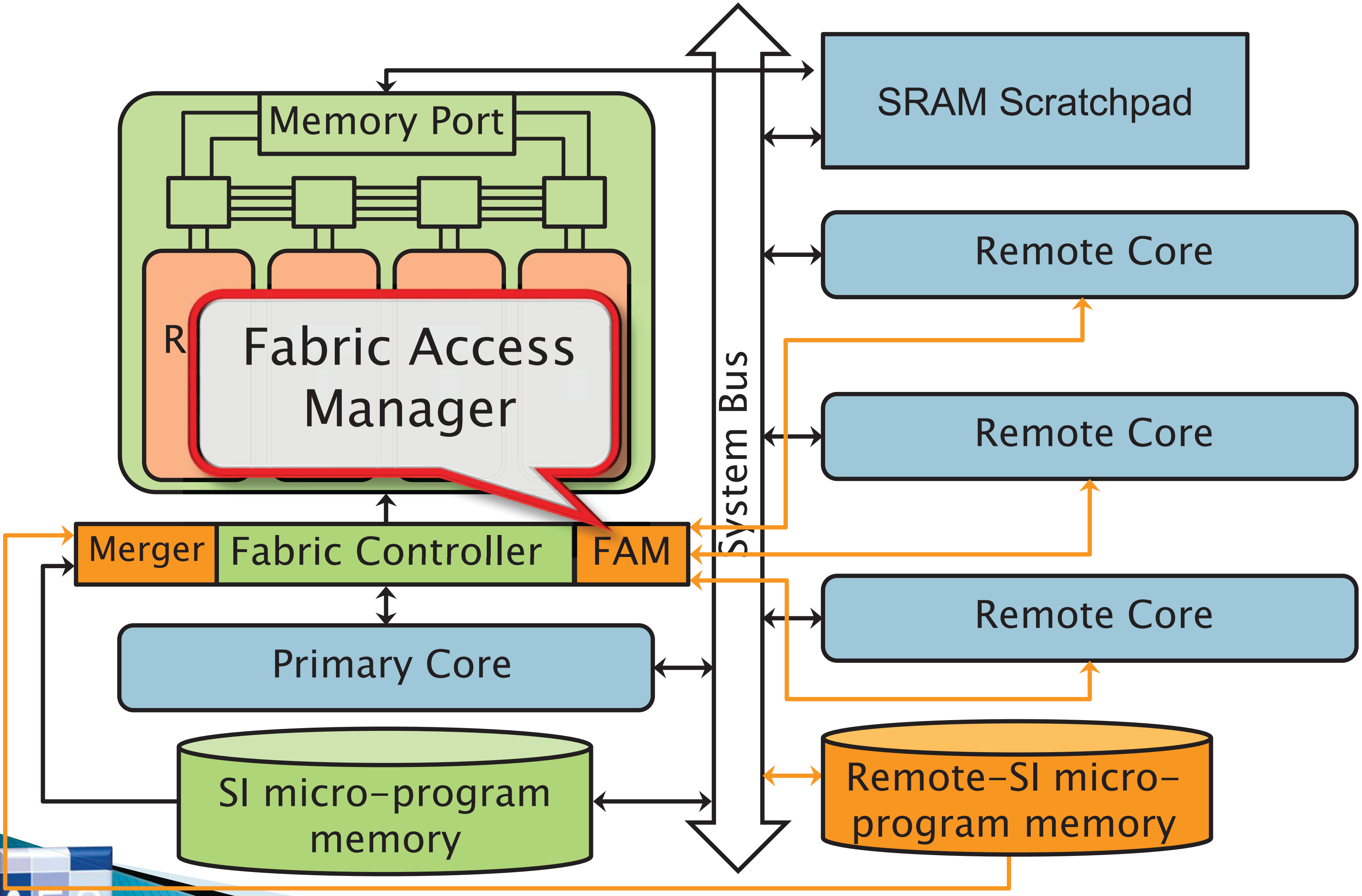


- Dedicated fabric share per core, e.g. [Watkins@MICRO10]
- Reduced reconfigurable area and memory bandwidth per core
- **Problem:** No adaption to dynamic workloads

COREFAB Architectural Extensions



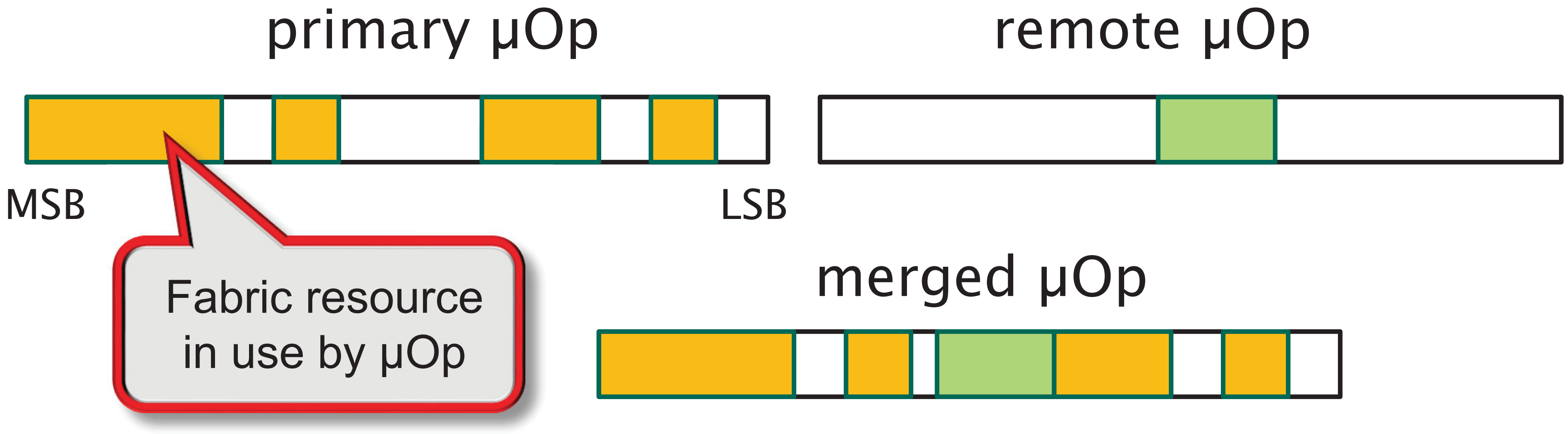
COREFAB Architectural Extensions



Merging Fabric Accesses

Prerequisite:

- Set of fabric **resources** used by current primary and remote µOp must be **disjoint**

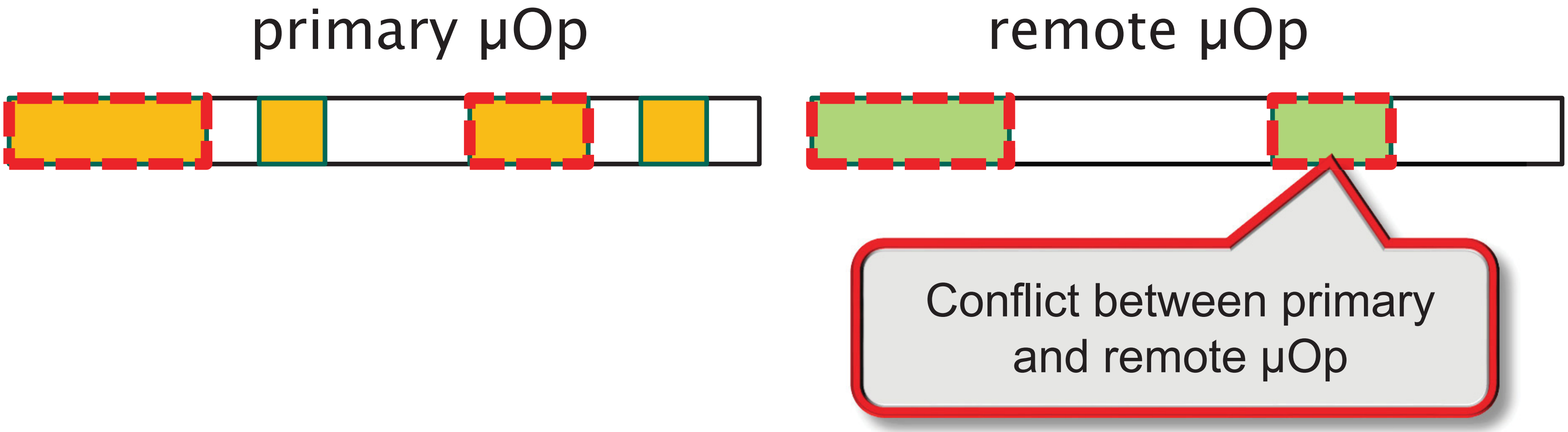


-

Merging Fabric Accesses

Prerequisite:

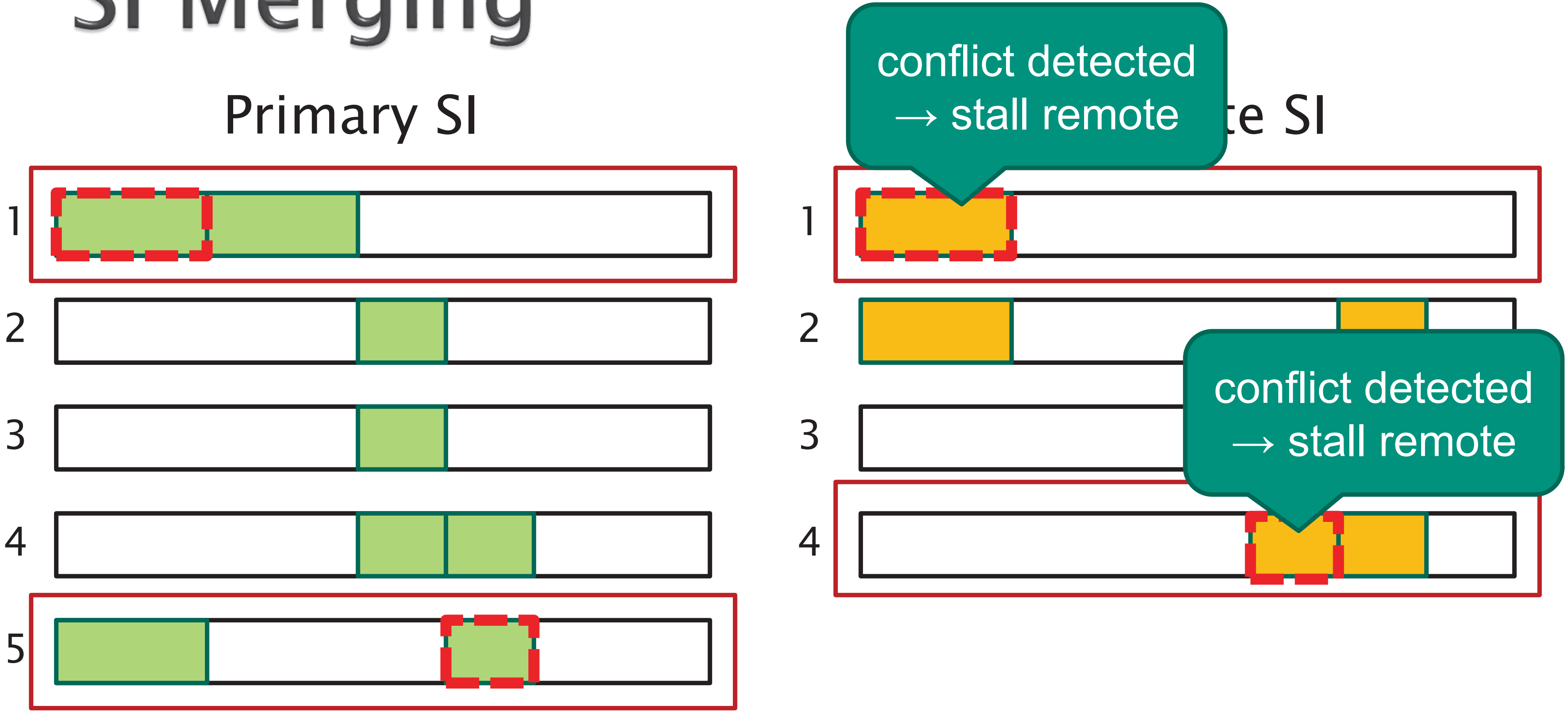
- Set of fabric **resources** used by current primary and remote µOp must be **disjoint**



Conflict between primary and remote µOp

- Conflict between µOps → merging not possible

SI Merging



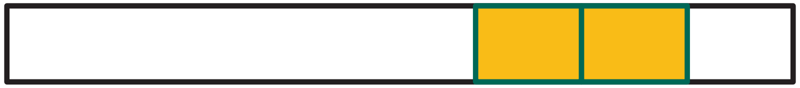
Primary SI

te SI

conflict detected
→ stall remote

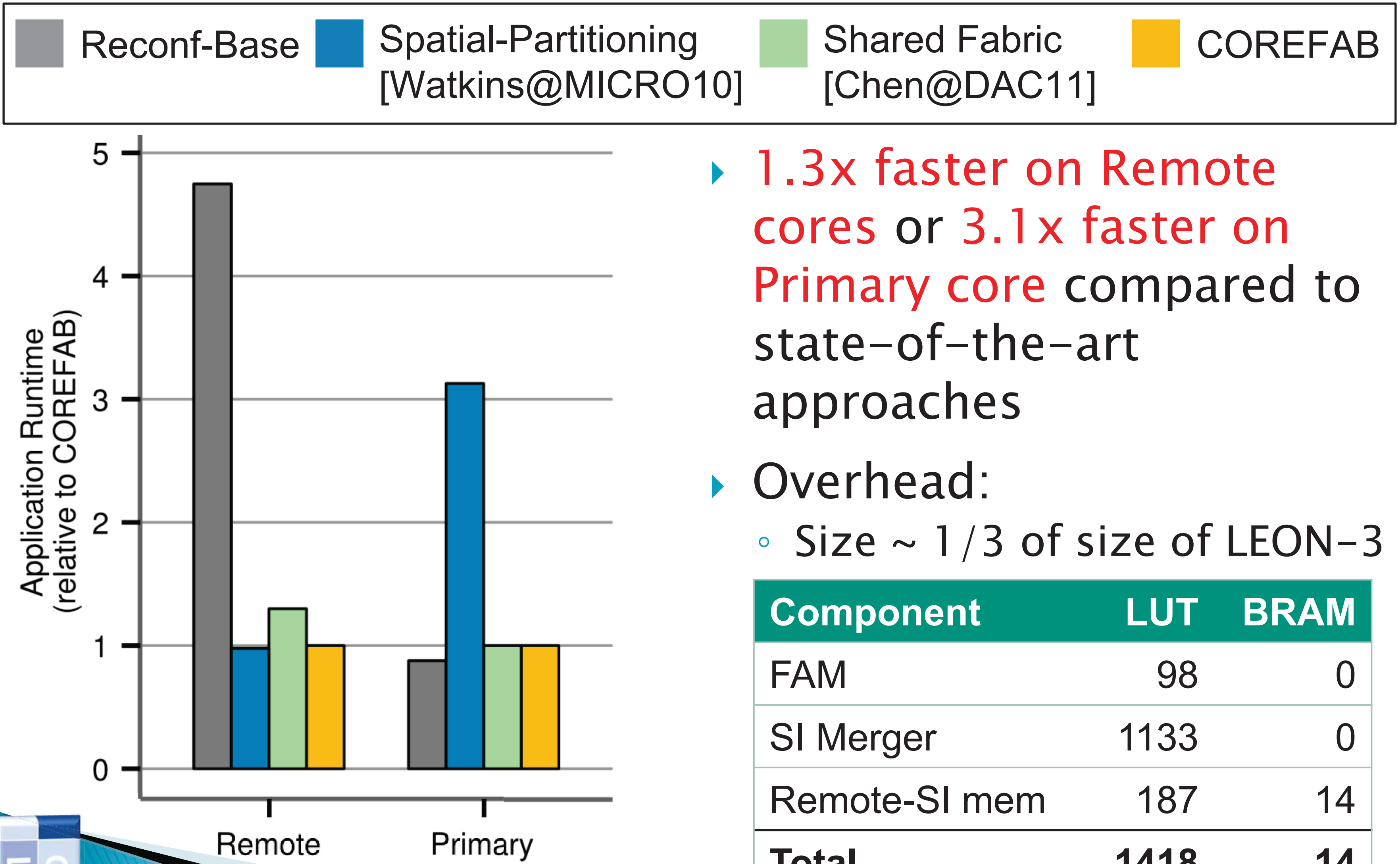
conflict detected
→ stall remote

µOp for fabric configuration



Execution time for both SIs: 6 cycles

COREFAB Results

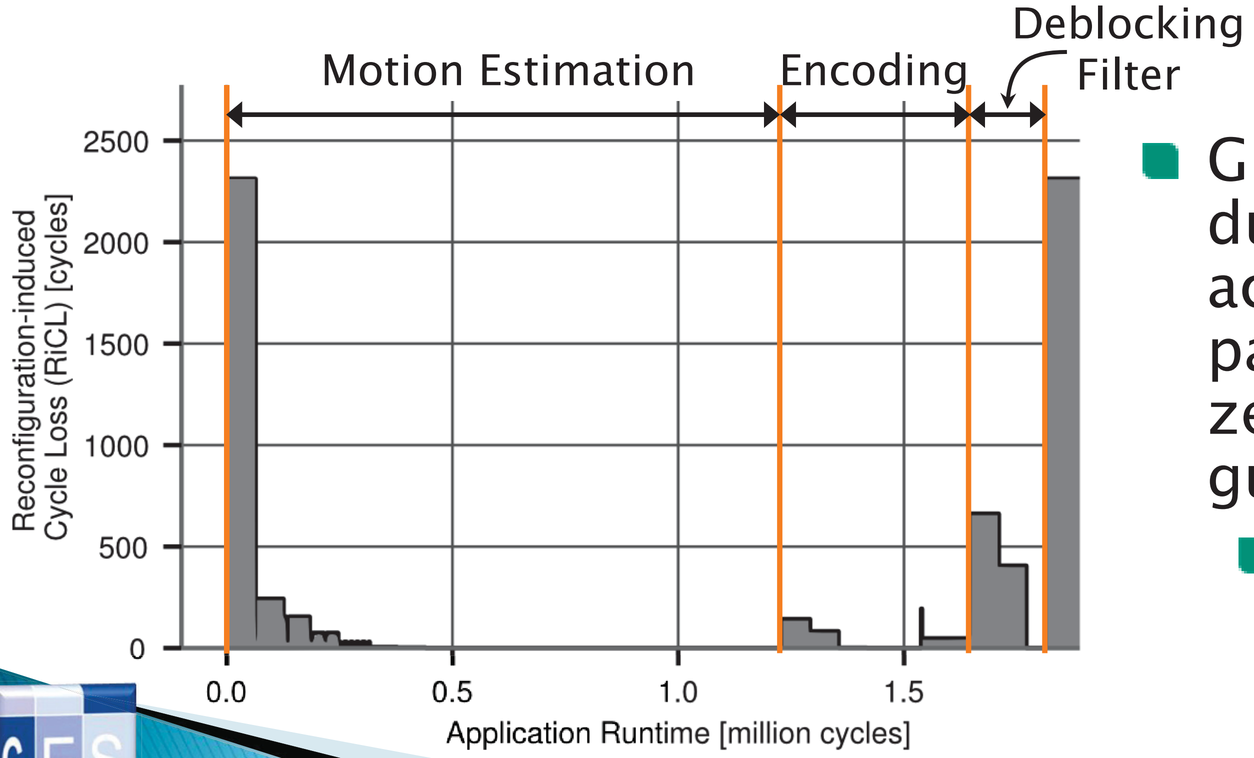


- 1.3x faster on Remote cores or 3.1x faster on Primary core compared to state-of-the-art approaches
- Overhead:
  - Size ~ 1/3 of size of LEON-3

Component	LUT	BRAM
FAM	98	0
SI Merger	1133	0
Remote-SI mem	187	14
Total	**1418**	**14**

7.4.5 Multi-tasking for reconf. proc.

- Demand: Many systems are multi-tasking systems anyway
- Optimization: Performance loss until reconfiguration of accelerators finished (range of milliseconds)
- Example: H.264 video encoder processes 1 frame



- Gray Bars: Cycle loss due to unavailable accelerators (compared to optimistic zero-cycle reconfiguration latency)
  - If it were zero, the frame would have been processed 1.35x faster

Analyzing the EDF scheduling policy for reconf. processors

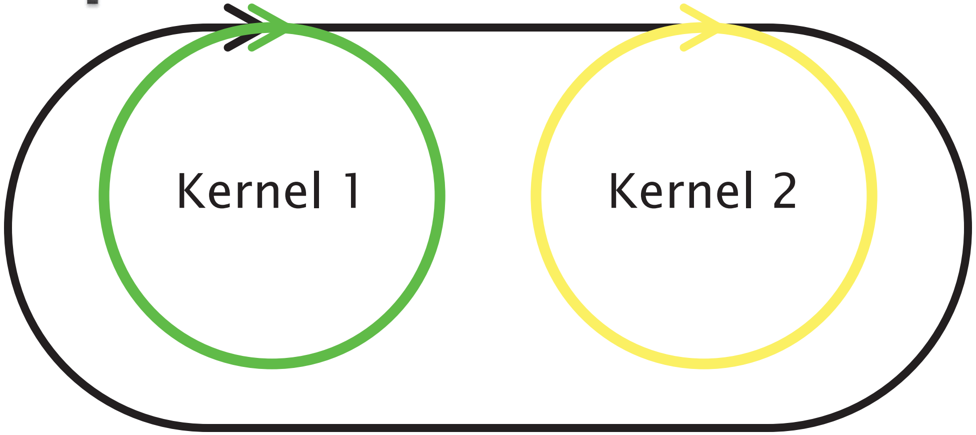
Task **T1**: Deadline: 10ms

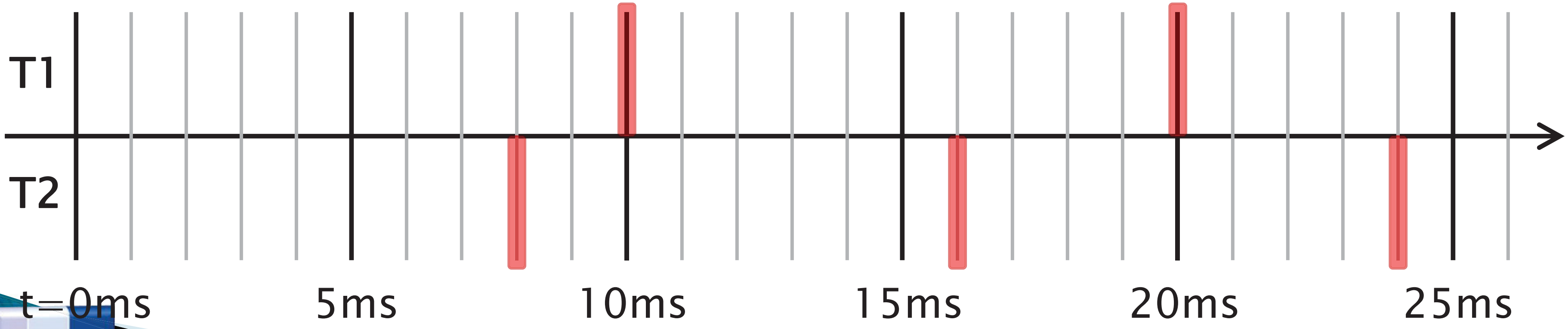
Kernel 1: • Software: 10ms

Kernel 2: • Software: 6ms

Task **T2**: Deadline: 8ms

Kernel 1: • Software: 5ms





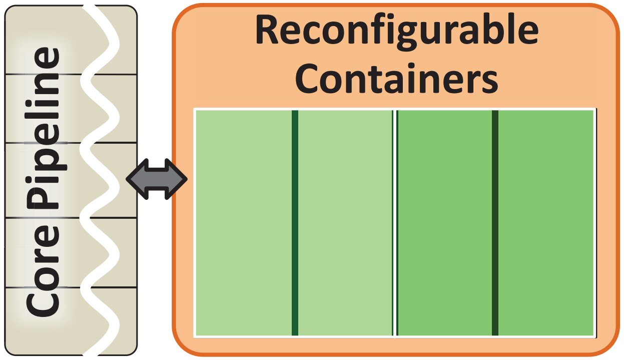
M. Damschen, KIT, 2016

Analyzing the EDF scheduling policy for reconf. processors

Task **T1**: Deadline: 10ms

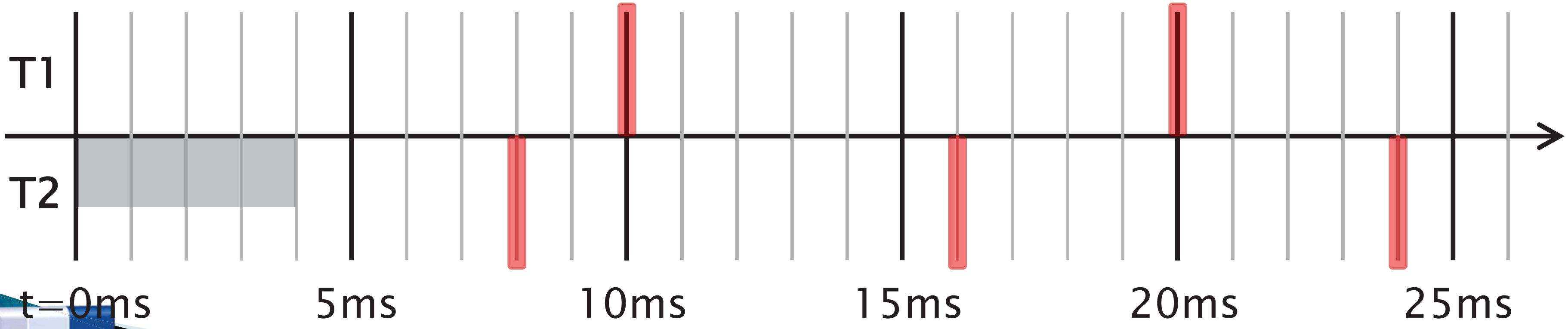
Kernel 1:
- Software: 10ms
- After 2ms reconf: 5ms (2x faster)
- After 4ms reconf: 2.5ms (4x faster)

Kernel 2:
- Software: 6ms
- After 3ms reconf: 1ms (6x faster)



Task **T2**: Deadline: 8ms

Kernel 1:
- Software: 5ms

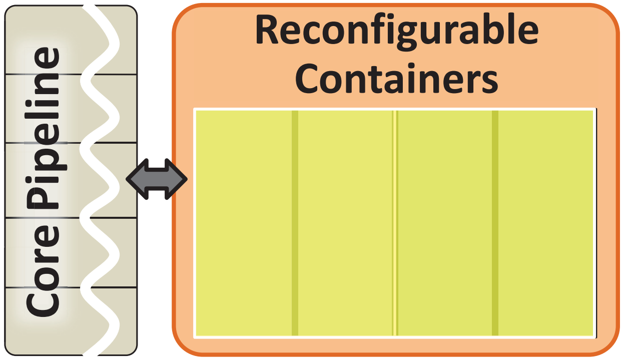


Analyzing the EDF scheduling policy for reconf. processors

Task **T1**: Deadline: 10ms

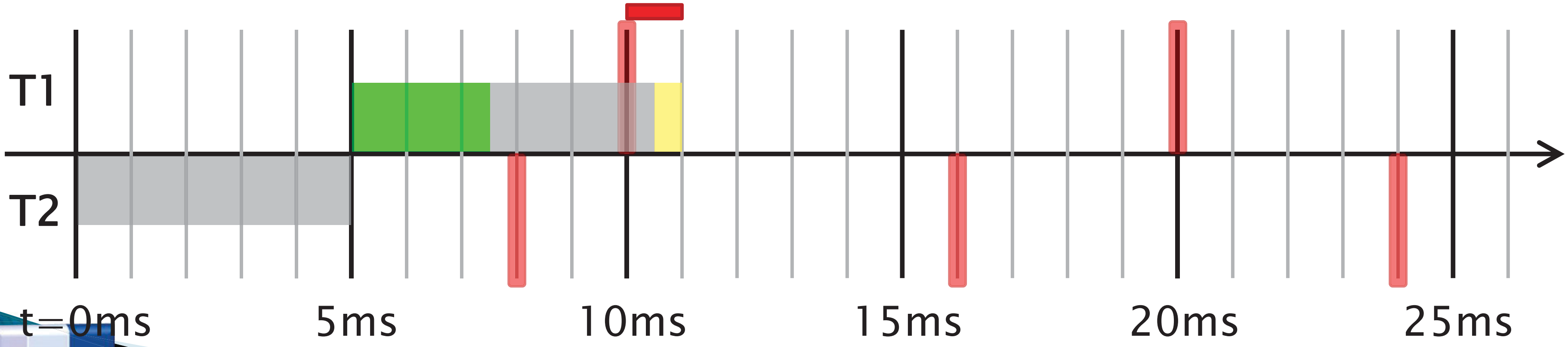
Kernel 1:
- Software: 10ms
- After 2ms reconf: 5ms (2x faster)
- After 4ms reconf: 2.5ms (4x faster)

Kernel 2:
- Software: 6ms
- After 3ms reconf: 1ms (6x faster)



Task **T2**: Deadline: 8ms

Kernel 1:
- Software: 5ms

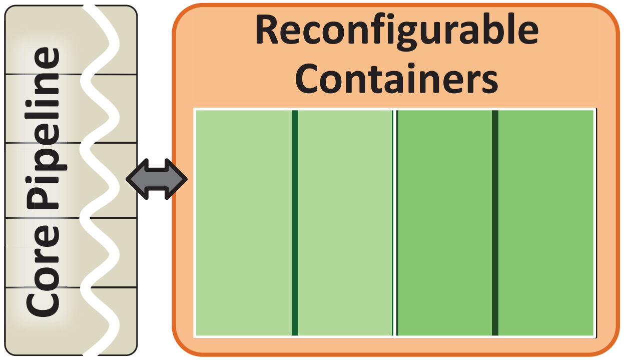


Analyzing the EDF scheduling policy for reconf. processors

Task **T1**: Deadline: 10ms

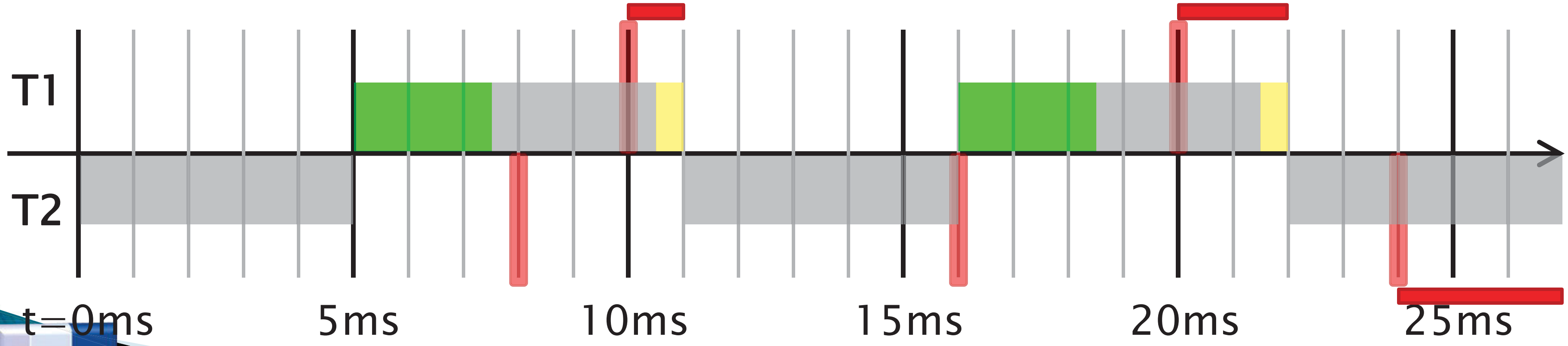
Kernel 1:
- Software: 10ms
- After 2ms reconf: 5ms **(2x faster)**
- After 4ms reconf: 2.5ms **(4x faster)**

Kernel 2:
- Software: 6ms
- After 3ms reconf: 1ms **(6x faster)**



Task **T2**: Deadline: 8ms

Kernel 1:
- Software: 5ms

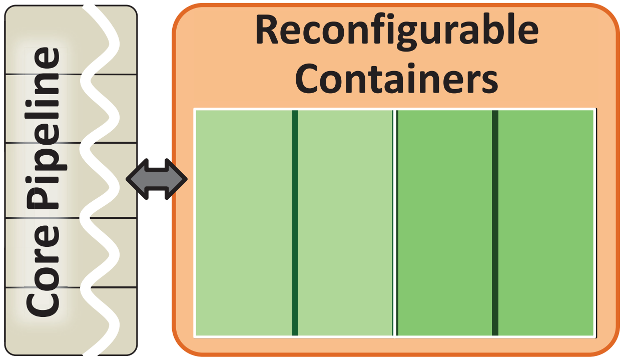


Analyzing the EDF scheduling policy for reconf. processors

Task **T1**: Deadline: 10ms

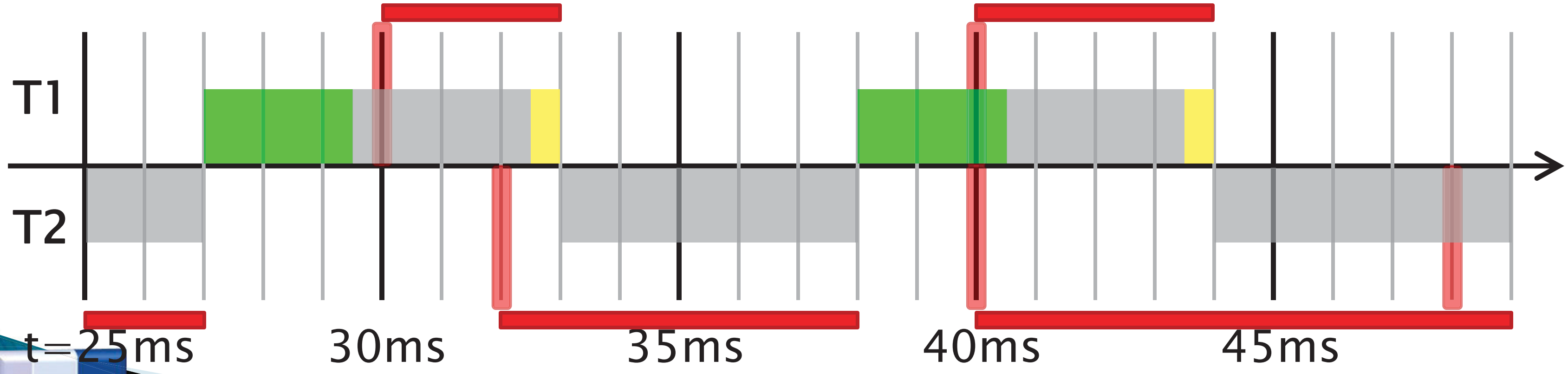
<u>Kernel 1:</u>
- Software: 10ms
- After 2ms reconf: 5ms **(2x faster)**
- After 4ms reconf: 2.5ms **(4x faster)**

<u>Kernel 2:</u>
- Software: 6ms
- After 3ms reconf: 1ms **(6x faster)**



Task **T2**: Deadline: 8ms

<u>Kernel 1:</u>
- Software: 5ms



Lessons learned

- Scheduler needs to consider that tasks have different Performance Levels that change over time
  - Try to exploit high performance levels, i.e. schedule those tasks
  - Try to avoid low performance levels, i.e. do not schedule those tasks
- Keep the reconfiguration port busy
  - If a task that is known to use Special Instructions did not issue a reconfiguration request (for the next kernel) yet, then schedule it
  - Reason: it will not increase its performance level until it at least issues a reconfiguration request
- Additionally: consider the soft deadlines of tasks
  - Even if a task has a low performance level, it might need to be scheduled to meet its deadline

A better schedule

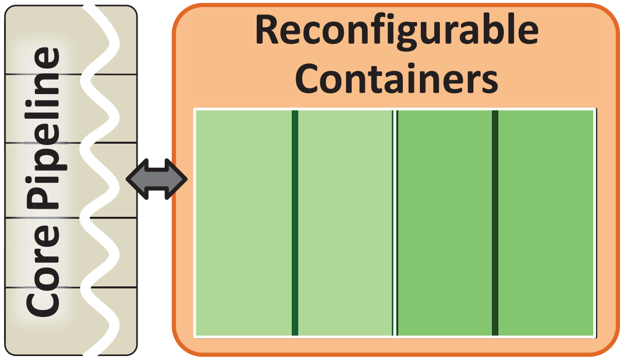
Task **T1**: Deadline: 10ms

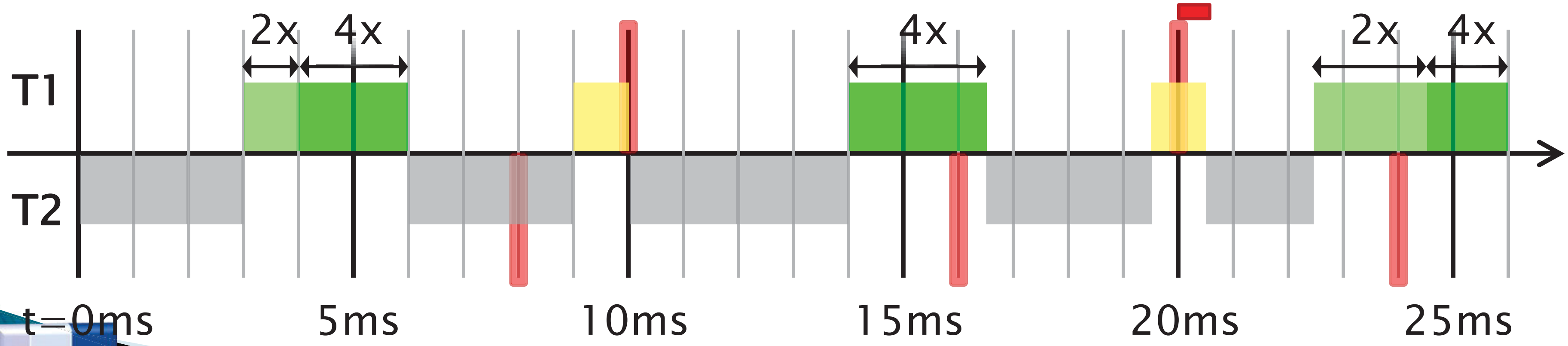
Kernel 1:
- Software: 10ms
- After 2ms reconf: 5ms (2x faster)
- After 4ms reconf: 2.5ms (4x faster)

Kernel 2:
- Software: 6ms
- After 3ms reconf: 1ms (6x faster)

Task **T2**: Deadline: 8ms

Kernel 1:
- Software: 5ms



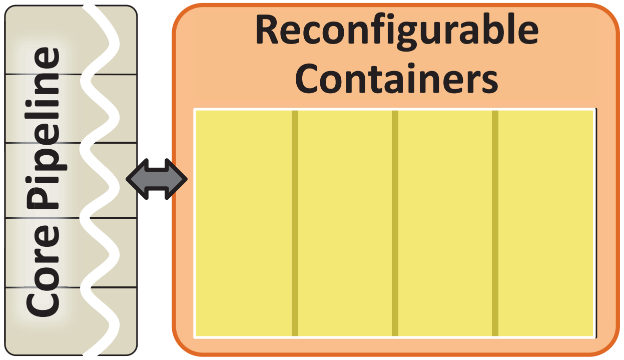


A better schedule

Task **T1**: Deadline: 10ms

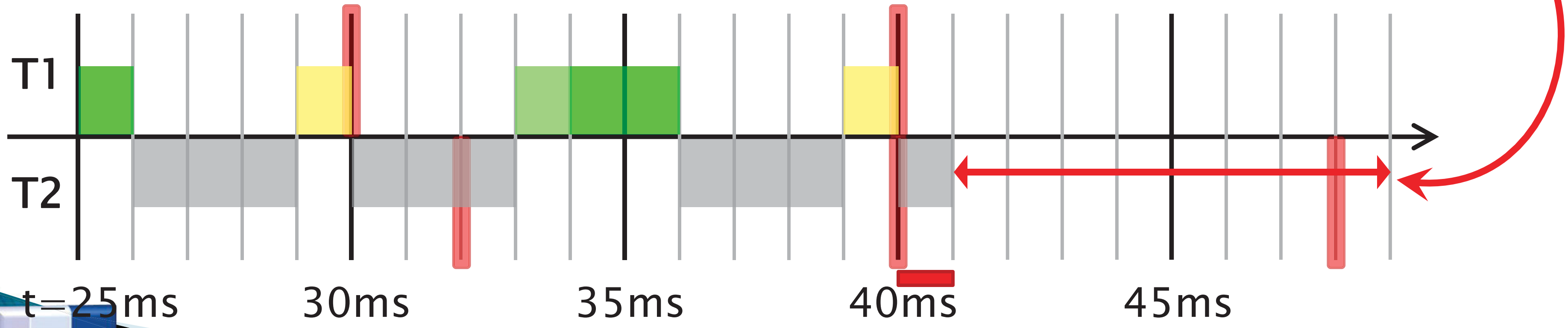
Kernel 1:
- Software: 10ms
- After 2ms reconf: 5ms (2x faster)
- After 4ms reconf: 2.5ms (4x faster)

Kernel 2:
- Software: 6ms
- After 3ms reconf: 1ms (6x faster)



Task **T2**: Deadline: 8ms

Kernel 1:
- Software: 5ms



Core Idea: Performance Level

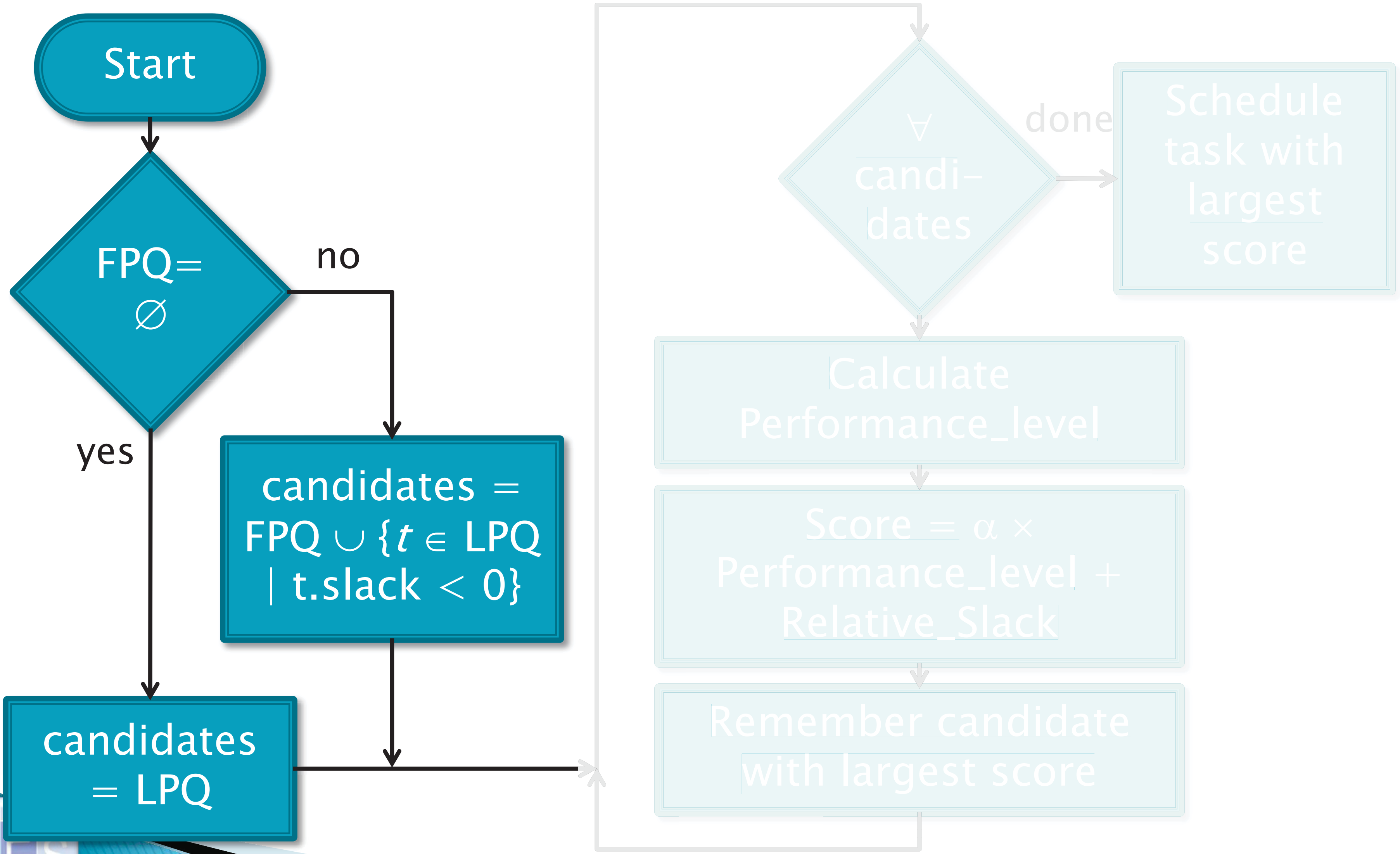
- To calculate the Performance Level ∈ [0,1] for Task *T* that executes Kernel *K* at time *t* we consider:
  - Which accelerators are requested for Kernel *K*
  - How many accelerators are attained at time *t*
  - → Calculate the average latency of the Special Instructions given the currently available accelerators compared to the latency after all requested accelerators are available

$$\begin{cases} \dfrac{\displaystyle\sum_{\substack{\text{SIs } S \text{ invoked} \\ \text{in Kernel } K}} \frac{S.latency\left(T.reqAcc(K)\right)}{S.latency\left(T.attAcc(t)\right)}}{\left|\text{SIs } S \text{ invoked in Kernel } K\right|} & \text{, if } \left|\begin{matrix}\text{SIs } S \text{ invoked} \\ \text{in Kernel } K\end{matrix}\right| > 0 \\ 1 & \text{, else} \end{cases}$$

Maintaining the task's state

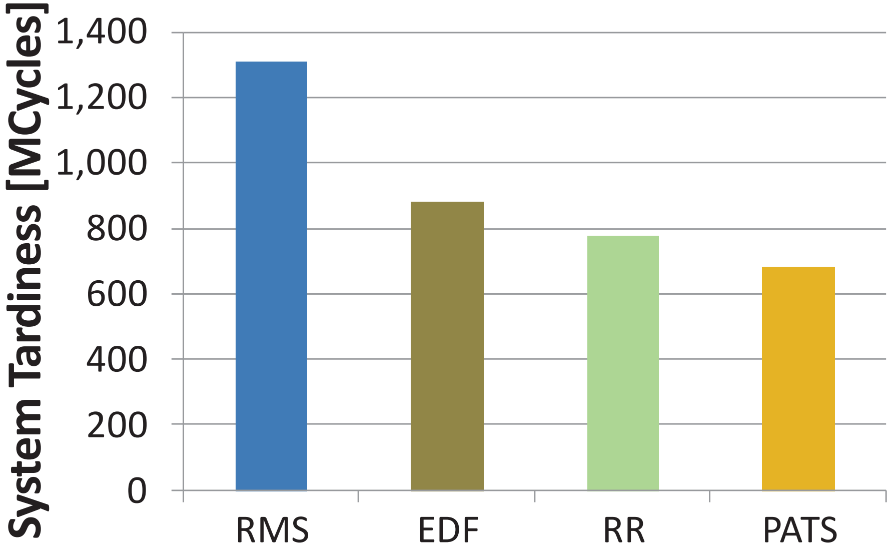
- NRQ: Not Released Queue – these tasks cannot be scheduled, as the previous job (if any) has completed and the next job is not released yet
- LPQ: Low Performance Queue – tasks can be scheduled but they would run at a reduced Performance Level due to not yet reconfigured accelerators
- FPQ: Full Performance Queue – these tasks can be scheduled and all requested accelerators are available
- (Re-)assigning tasks to queues is managed
  - at context switch
  - when a reconfiguration completes
  - when a task requests different accelerators

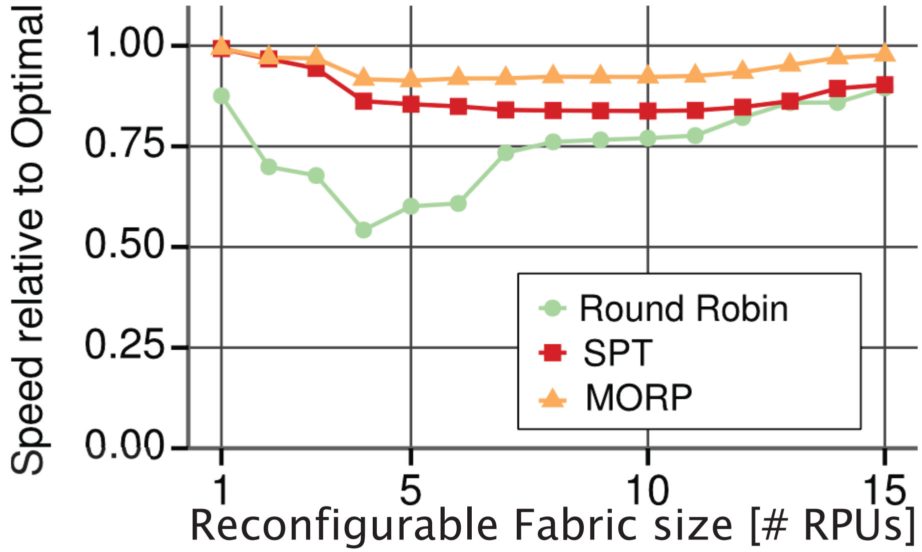
PATS: Performance Aware Task Scheduling


Start
FPQ= ∅
no
yes
candidates = FPQ ∪ {t ∈ LPQ | t.slack < 0}
candidates = LPQ
∀ candi-dates
done
Schedule task with largest score
Calculate Performance_level
Score = α × Performance_level + Relative_Slack
Remember candidate with largest score

Task Scheduling Results

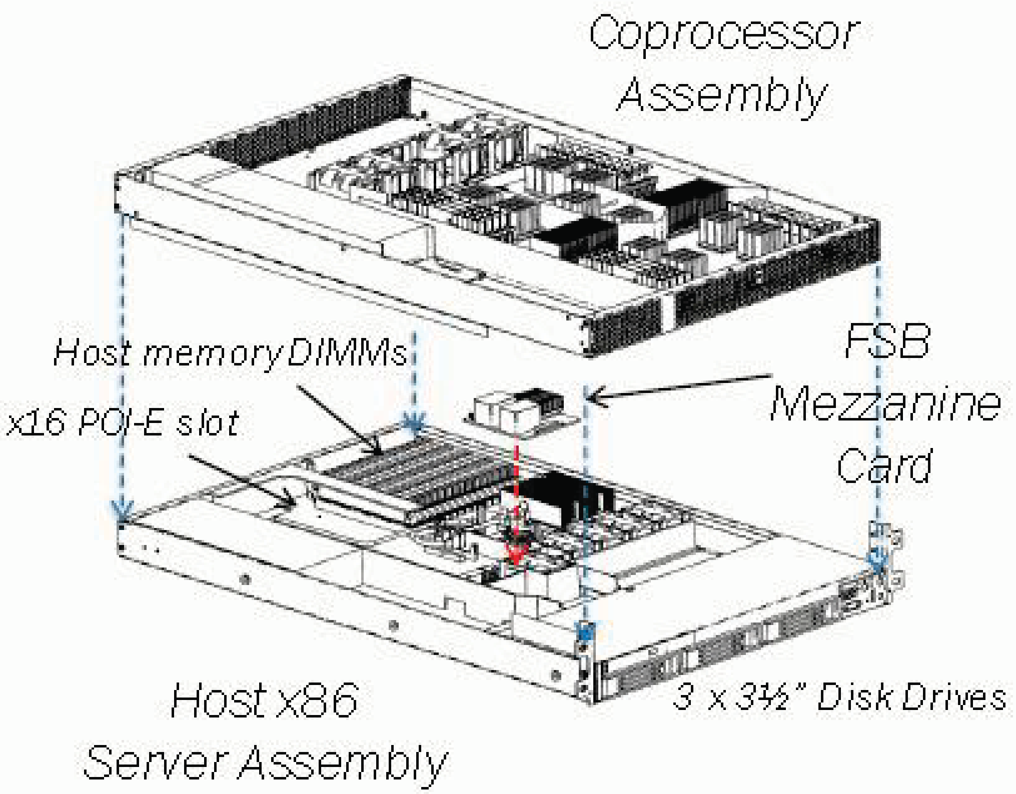
- **PATS**: Performance Aware Task Scheduler
  - Def. System Tardiness: Sum of all times that jobs finished too late
  - Outperforms the other scheduler in nearly all cases
  - Only in rare cases slightly beaten by EDF
  - Sometimes RR is the closest competitor, sometimes the worst performer
  - PATS is on average 1.92x, 1.29x and 1.14x faster than RMS, EDF, and RR, respectively
- **MORP**: Makespan Optimization for Reconfigurable Processors
  - Def. Makespan: Time until all concurrently started tasks of a task set are completed
  - Hybrid task-scheduling and area-allocation approach
  - Makespans are only 5.8% (mean) worse than upper bound





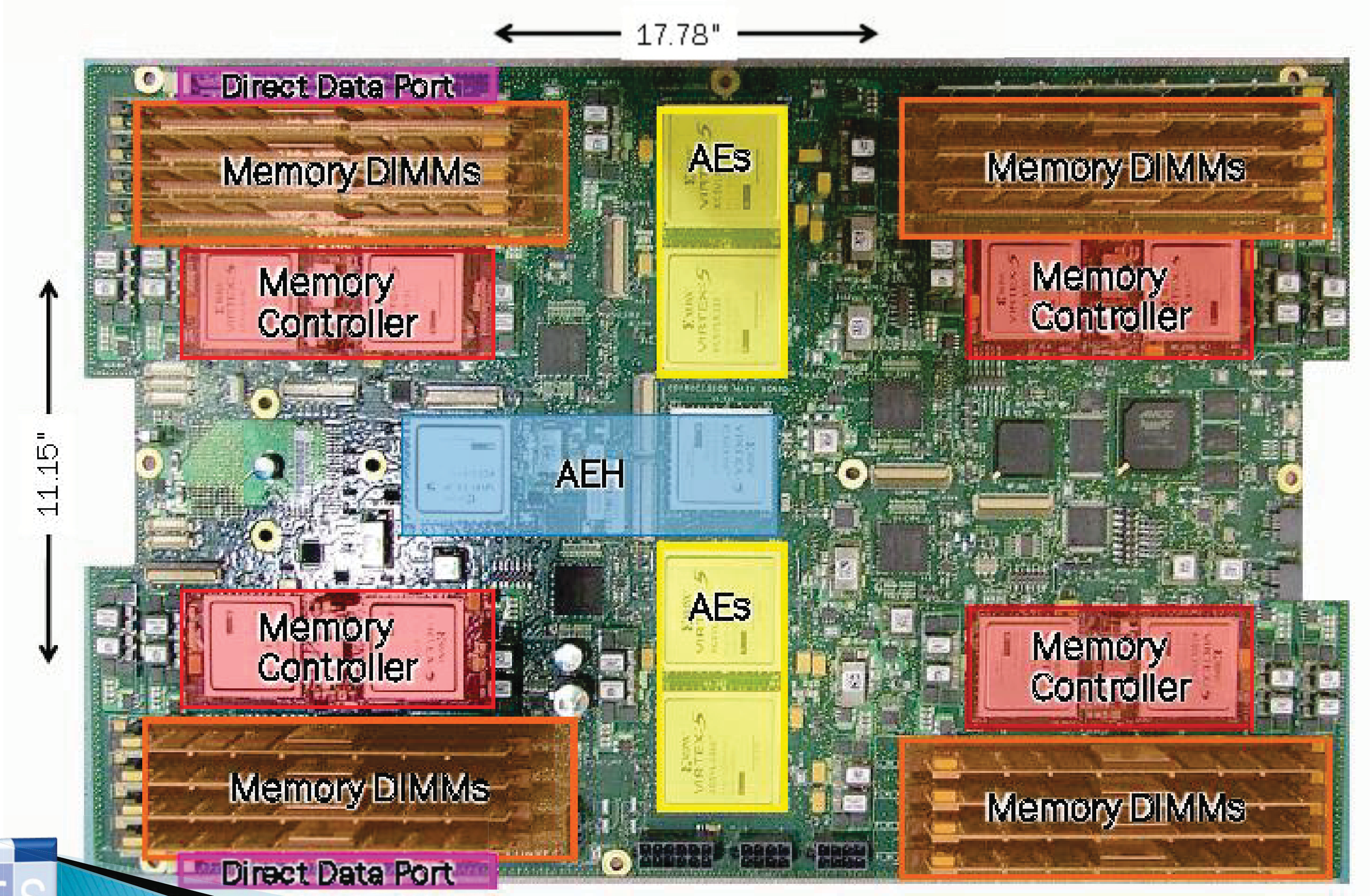
7.4.6 High-Performance Computing (HPC) Domain: Convey HC-1

- **2U enclosure:**
  - Top half of 2U platform contains the coprocessor
  - Bottom half contains Intel motherboard



src: Convey Workshop 2010; http://www.conveycomputer.com/

HC-1 Physical Layout



src: Convey Workshop 2010; http://www.conveycomputer.com/