CHAIR FOR COMPILER CONSTRUCTION

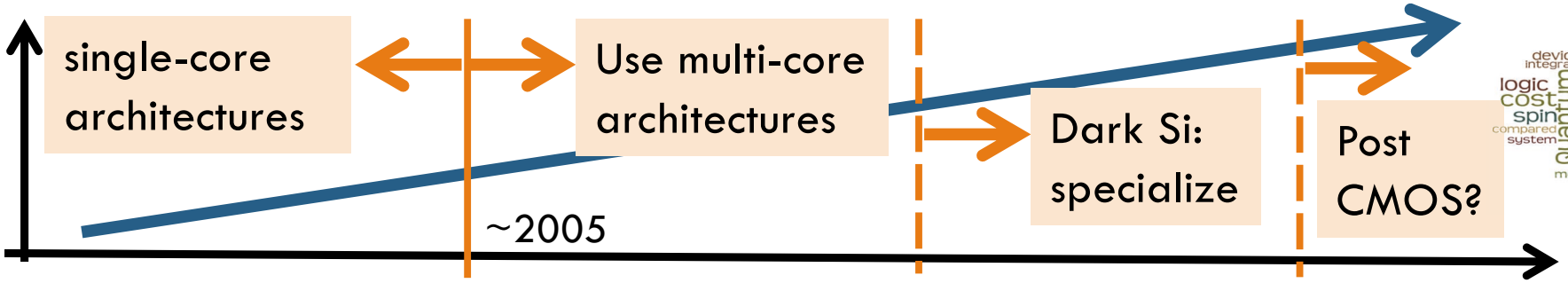
cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Compilers for Processors and Systems

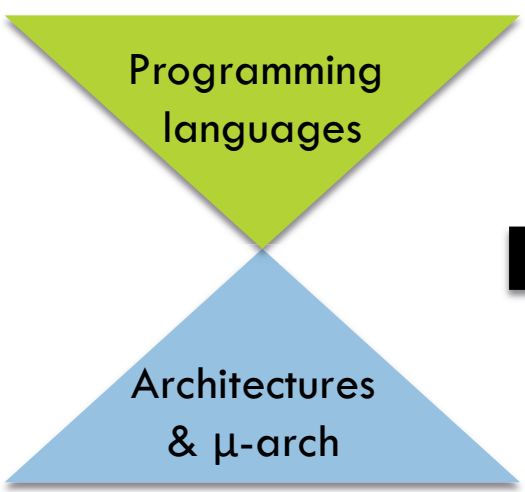
Jeronimo Castrillon
Chair for Compiler Construction
TU Dresden
jeronimo.castrillon@tu-dresden.de

Forschungslinie – Einführung in die Forschung. Modul INF-D-910
April 2019

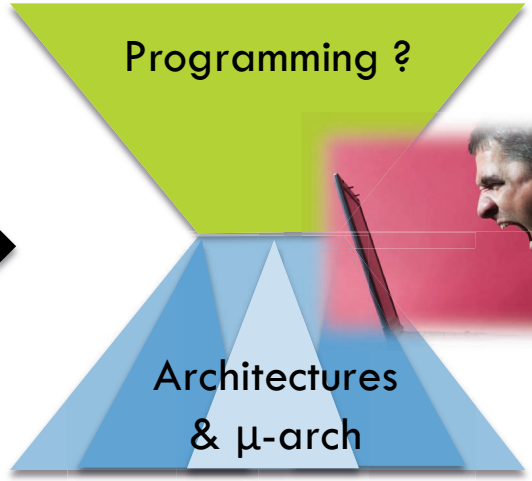
Inflection points in computing and programming



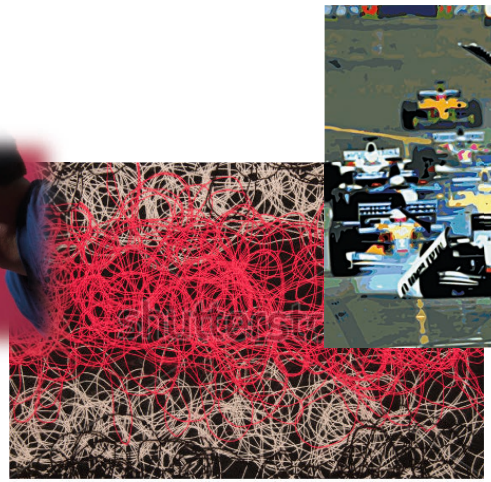
- The programming interface continues to broadens as hardware evolves



Single core



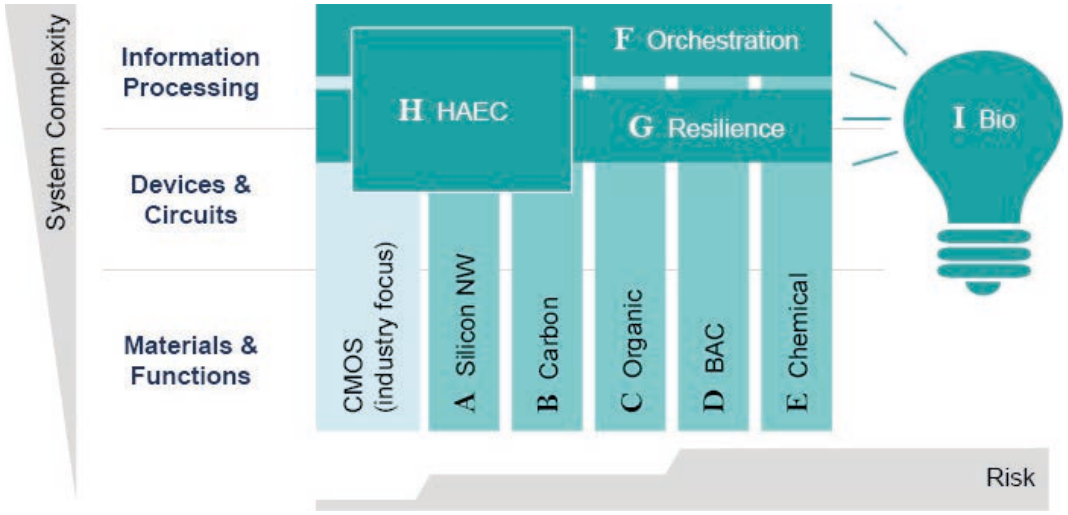
(het.) Multi-processors



http://nerds-central.blogspot.de/2011/11/atomicinteger-volatile-synchronized-and.html

Research at the Chair for Compiler Construction

- Context: The Center for Advacing Electronics Dresden (cf**aed**)
- Research Overview
  - Programming parallel heterogeneous systems
  - Domain-specific languages and optimization
  - Tools and methodologies for Post-CMOS systems
  - Optimization: Performance, energy efficiency & resilience

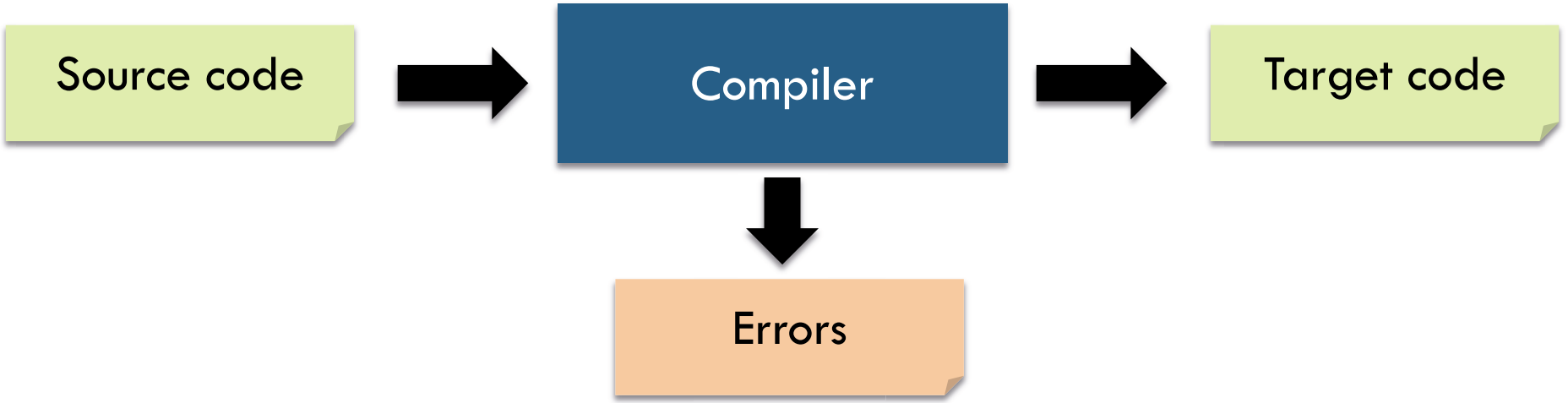


In this lecture

- Classical compilers (refresh)
- Multi-cores: Parallel programming
- Emerging topics in compilers

Classical compilers

What is a compiler



- Compiler translates an input source code to a target code
  - Typical: target code closer to machine code (e.g., C ➔ assembly)
  - Must recognize illegal code and generate correct code
  - Must agree with lower layers (e.g., storage, linker and runtime)
- Today: just-in-time compilers, auto-tuning, continuous compilation, …

Structure of a compiler: Front-end + Middle-end

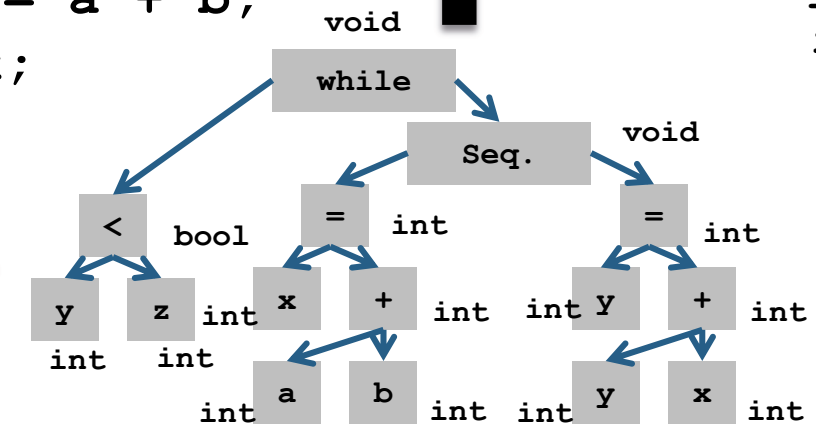
- Lexical (scanner): Maps a character stream into words (tokens)
- Syntax (parser): Recognizes "sentences of tokens" according to a grammar
- Semantic analysis: Adds information and checks – types, declarations, …
- IR-generation: Abstract representation for optimization and code generation
- IR-optimization: Simplification & improvements (e.g., remove redundancies)

Structure of a compiler: Backend

- Code selection: Decide which instructions should implement the IR
- Register allocation: Decide in which register to place variables
- Scheduling: Decide when to execute the instructions (e.g., ordering in assembly program) & ensure conformance with interfaces and constraints

IR generation & optimization – Example


```
while (y < z)
{
  int x = a + b;
  y += x;
}
```




```
Loop: x = a + b
      y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

Abstract representation of the program, closer to the target

```
      x = a + b
Loop: y = x + y
      _t1 = y < z
      if _t1 goto Loop
```

LLVM IR: Three-address code (TAC)

- Example: LLVM Framework

```
int main()
{
  int a, b, c;
  a = 2; b = 3;c = 5;
  c += (a*b) >> a;
  c += foo(a);
  return 0;
}
```

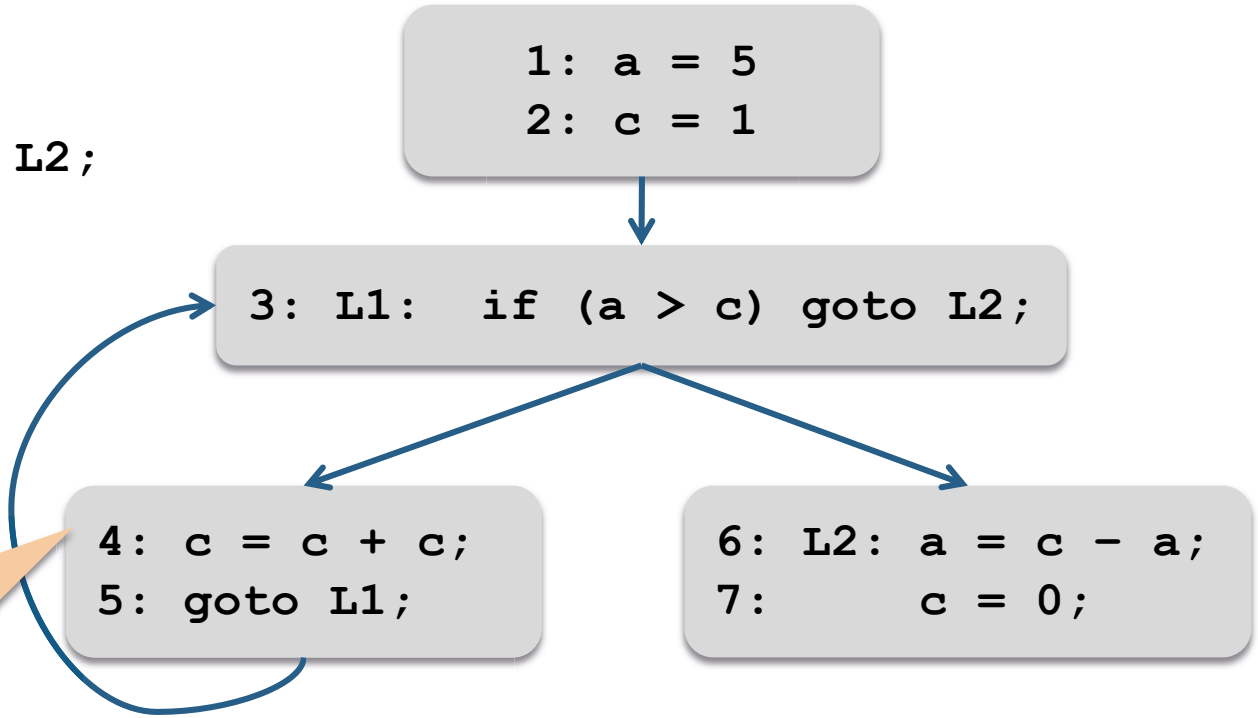
```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %a = alloca i32, align 4
  %b = alloca i32, align 4
  %c = alloca i32, align 4
  store i32 0, i32* %1
  store i32 2, i32* %a, align 4
  store i32 3, i32* %b, align 4
  store i32 5, i32* %c, align 4
  %2 = load i32* %a, align 4
  %3 = load i32* %b, align 4
  %4 = mul nsw i32 %2, %3
  %5 = load i32* %a, align 4
  %6 = ashr i32 %4, %5
  %7 = load i32* %c, align 4
  %8 = add nsw i32 %7, %6
  store i32 %8, i32* %c, align 4
  %9 = load i32* %a, align 4
  %10 = call i32 @foo(i32 %9)
  %11 = load i32* %c, align 4
  %12 = add nsw i32 %11, %10
  store i32 %12, i32* %c, align 4
  ret i32 0}
```

Lots of room for optimization!

Graph representations: control flow graph

- Control flow graph: Represent the branching structure of programs

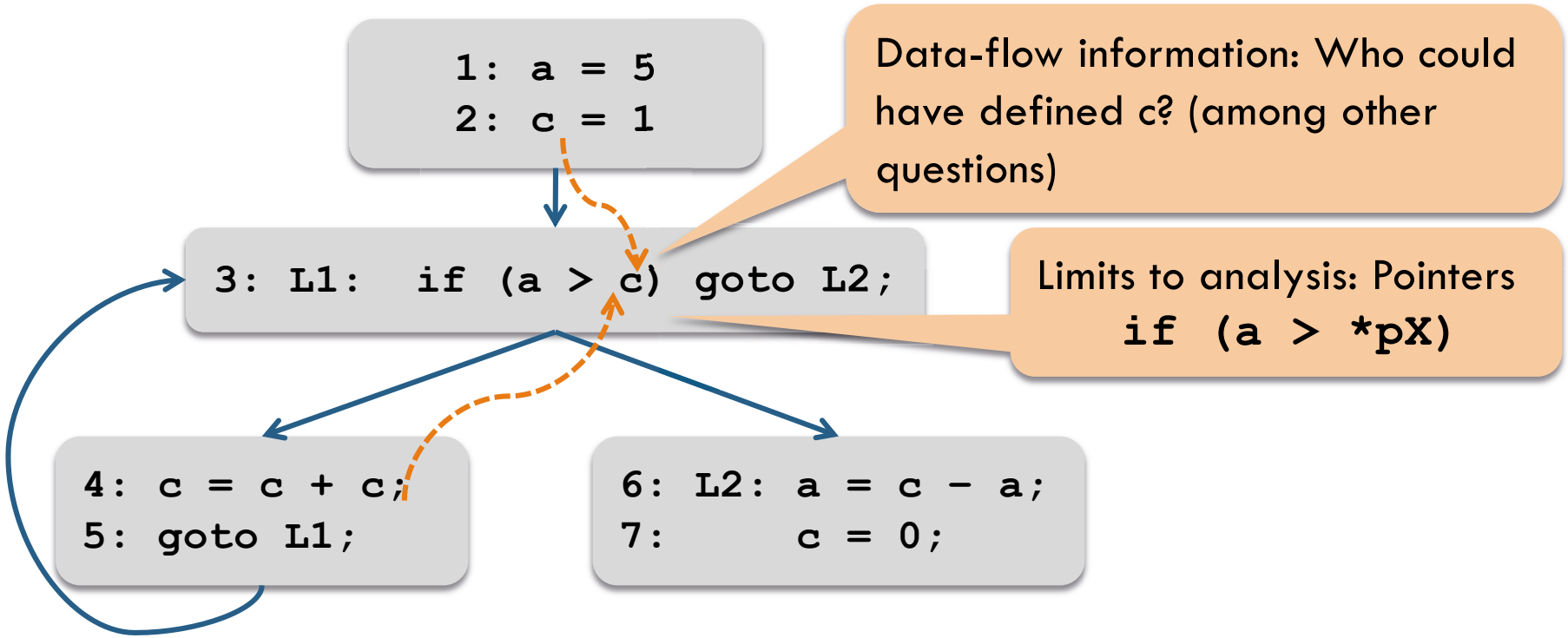
```
1:       a = 5;
2:       c = 1;
3: L1:   if (a > c) goto L2;
4:       c = c + c;
5:       goto L1;
6: L2:   a = c - a;
7:       c = 0;
```



Basic-blocks: sequence of statements w/o branching in between

Graph representations: data-flow information

- The compiler has to know where the data is coming from for optimizations



Sample optimization: Function inlining

- Inlining: Replace function call by inserting the function body in the code
- Why
  - Increase potential for other optimizations (w/o complex inter-procedural analysis)
  - Reduce stack management overhead
  - Remove jumps (call and return)
- Common example of the tradeoff between code-size & performance
- Minimal example

```
int f1(int x)
{ return x+1; }

int f2(int x)
{ return f1(x)+2; }
```

```
int f2(int x)
{ return x+1+2; }
```

```
int f2(int x)
{ return x+3; }
```

Function inlining (2)

- Lots of support in compilers
- Intuition
  - Static calls impact code size
  - Dynamic calls impact performance (need profiling)
  - When to inline?
    - Small functions (comparable to calling overhead)
    - Single static call
    - Static calls within loops (high dyn. calls)
    - Functions with single switch-case and often called with constant parameters

```
-fno-inline
-finline-small-functions
-findirect-inlining
-finline-functions
-finline-functions-called-once
-fearly-inlining
-finline-limit=n
-fno-keep-inline-dllexport
-fkeep-inline-functions
-fpartial-inlining
-flto[=n] (for linked-time opt)
--param name=value
  max-inline-insns-single
  max-inline-insns-auto
  inline-min-speedup
  large-function-insns
  large-function-growth
  large-unit-insns
  inline-unit-growth
  max-inline-insns-recursive
  max-inline-insns-recursive-auto
...
```

Function inlining under code size constraints

- Consider a call graph CG = (V,E,B,C,D), with a node for every function, an edge $e=(f_i,f_k)$ if f_i calls f_k and
  - $B(f_i)$ the code size of f_i w/o inlining
  - $C(e)$ the number of static calls of f_k in f_i
  - $D(f_i)$ the number of dynamic calls to f_i
- **Code size:** Still quite alive (best-paper award **CGO'19**)
- **Problem formulation:** Inline a set of functions so that performance is optimized while keeping the code size below a threshold L
  - Not trivial due to **mutual dependence** between inlining of different functions

Function inlining: Formulation

- Given a CG=(V,E,B,C,D), let $b_i = \begin{cases} 0 & f_i \text{ not inlined} \\ 1 & f_i \text{ inlined} \end{cases}$
- Find inlining $B = (b_1, \ldots, b_{|V|}) \in \{1,0\}^{|V|}$ such that $D(B) = \sum_{i:b_i=0} D(f_i)$ is minimized, subject to a size constraint

$$S(B) = \sum_{i=1}^{|V|} S(f_i) < T,$$

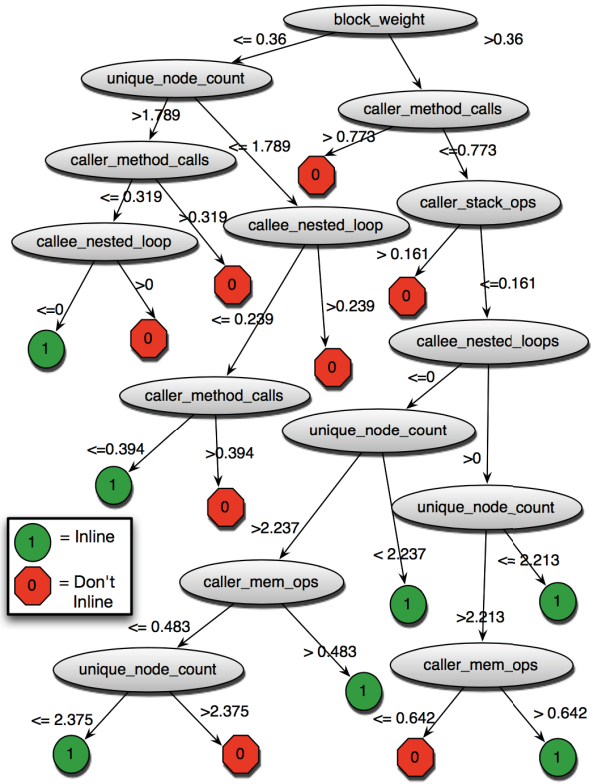
$$S(f_i) = B(f_i) + \sum_{j:b_j=1} C(f_i, f_j) \cdot S_j$$

Assumption: If inlined, then for all static call sites at once

Recursive computation (does not support cyclic call graphs)

Solution approaches

- Branch & bound solution: Expensive but amortizable for embedded applications
  - R. Leupers and P. Marwedel, "Function inlining under code size constraints for embedded processors," ICCAD'99
- Auto-tuning for dynamically compiled languages
  - J. Cavazos and M. F. P. O'Boyle, "Automatic Tuning of Inlining Heuristics," SC'05
- Using machine learning
  - S. Kulkarni, J. Cavazos, C. Wimmer and D. Simon, "Automatic construction of inlining heuristics using machine learning," CGO'13



Source: S. Kulkarni, et al, "Automatic construction of inlining heuristics using machine learning," CGO'13

Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages

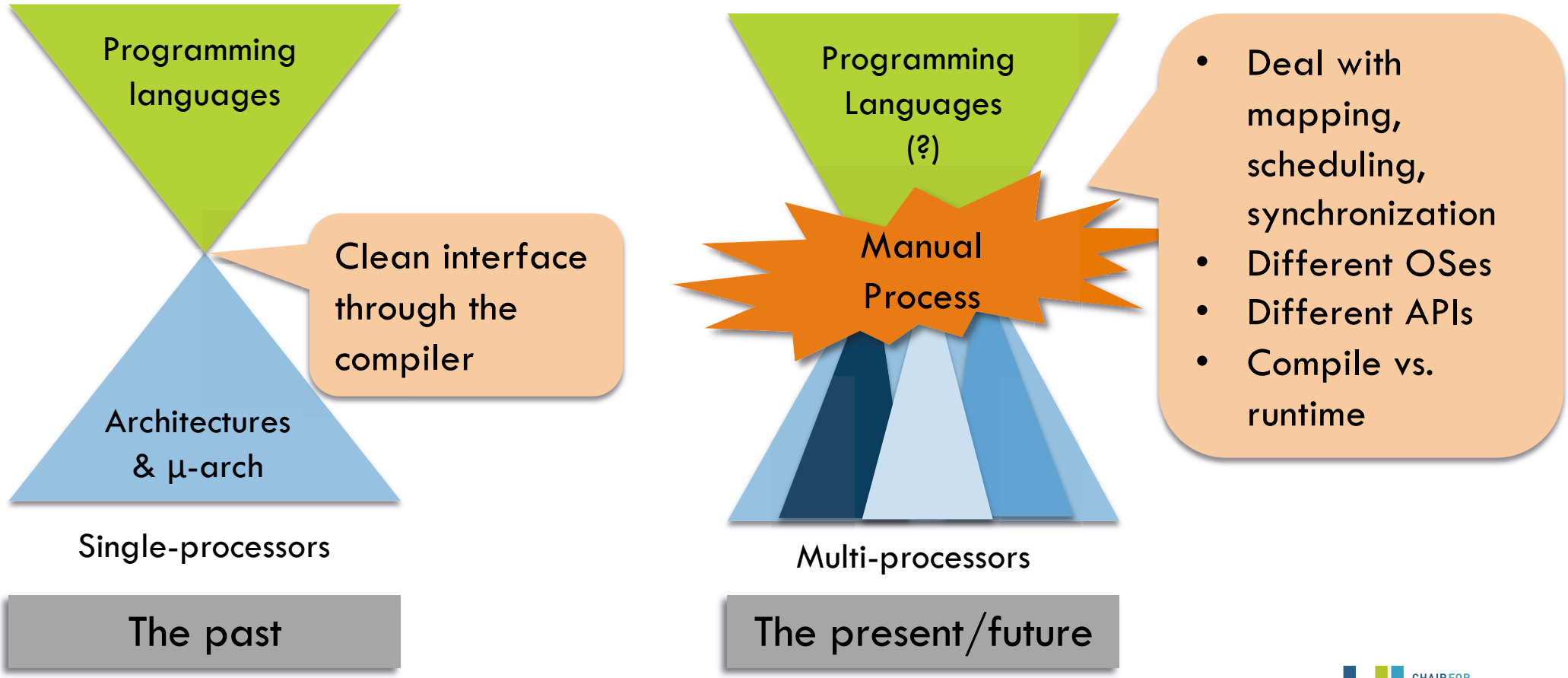
CHAIR FOR COMPILER CONSTRUCTION

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages

Challenges in multi-core compilation



Multi-core compilers

- Deal with similar problems than classical compilers
  - Parse and understand high-level parallel language constructs
  - Search for parallelism, but at a higher level of abstraction (higher than ILP)
  - Requires a model of the target architecture, but at coarser level
  - Allocation and scheduling of data to memories and tasks to processors
  - Code generation via source-to-source compilation

On parallel programming models

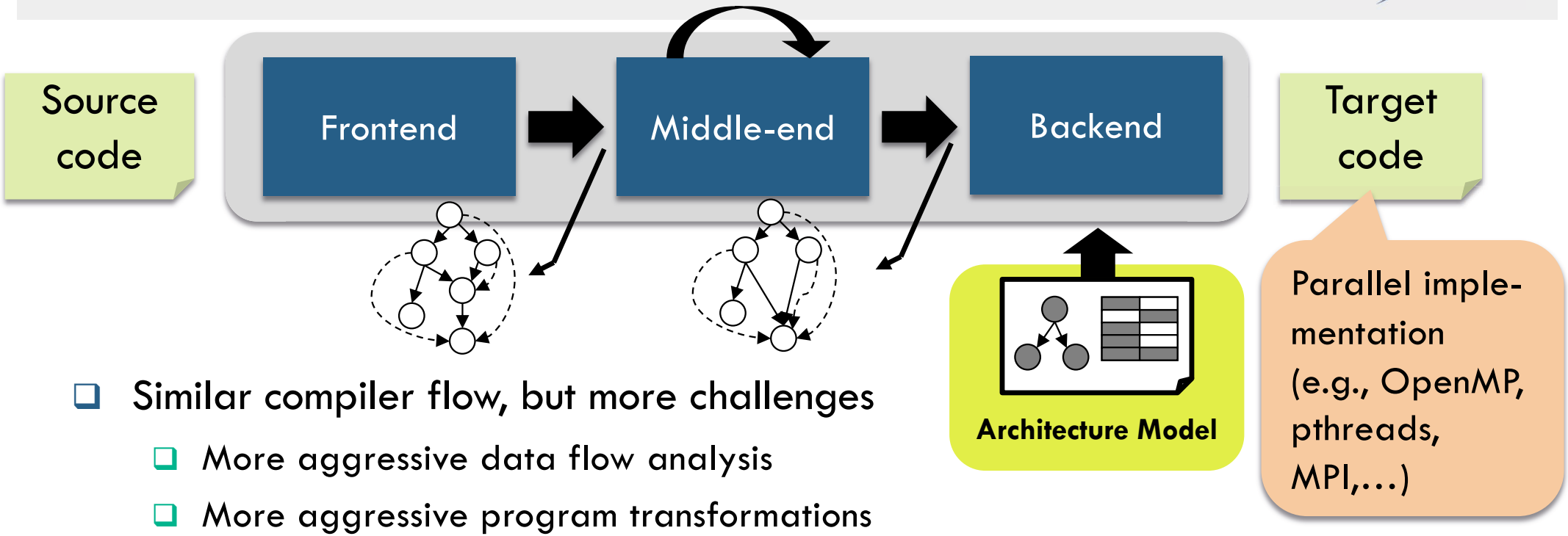
- There are many, each with a different impact on compilers
- Sequential programming models: C/C++, Matlab, …
- Parallel programming models
  - Shared memory: pthreads, OpenMP, Intel TBB, Cilk, …
  - Distributed memory: MPI, Charm++, …

- In this lecture: More automatism (not all general purpose)
  - Extracting coarse-grained parallelism from C code
  - Parallel dataflow models
  - Domain-specific languages

Multi-cores: Parallel programming

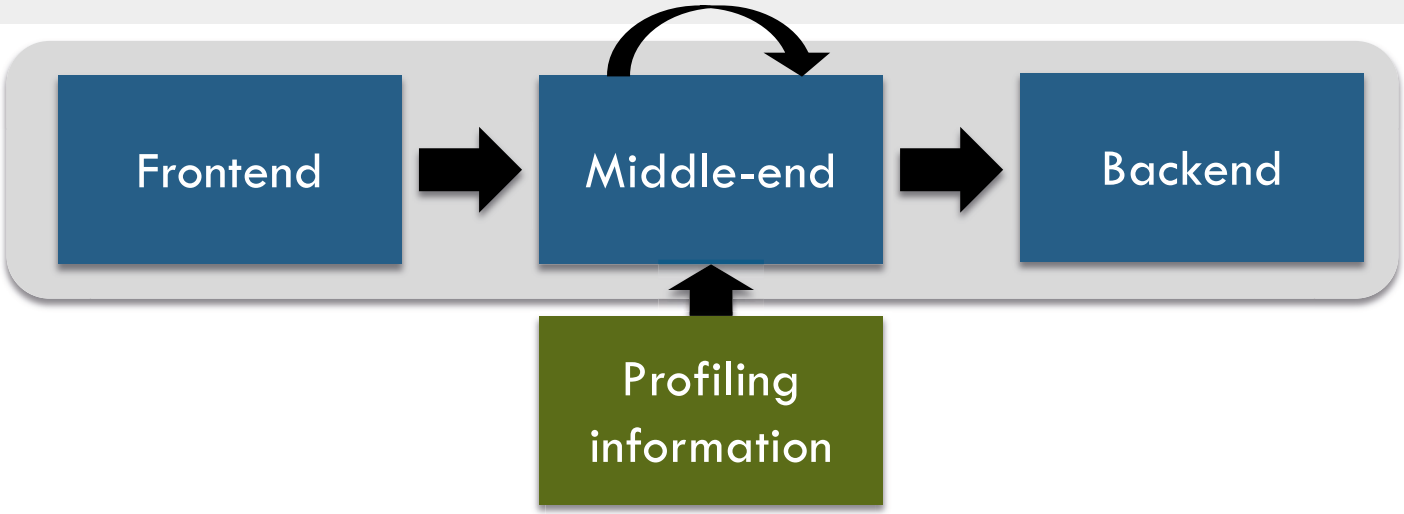
- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages

Principle of operation



- Similar compiler flow, but more challenges
  - More aggressive data flow analysis
  - More aggressive program transformations
  - Different granularity (basic-blocks?, functions?)
  - Focus on coarse-grained parallelism patterns
  - Whole program analysis

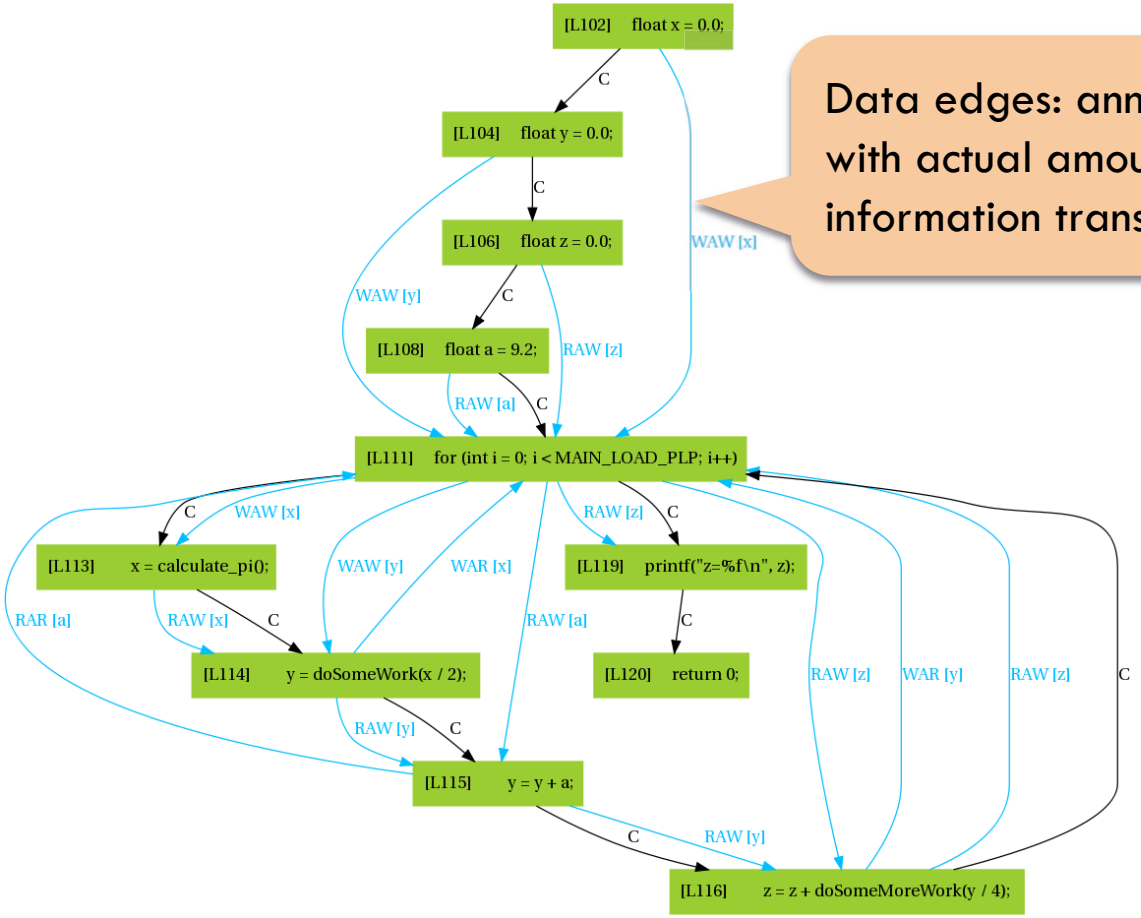
Data-flow analysis



- **Dynamic** data flow analysis via execution traces
  - More exact: Find exact portions of memory being read/written
  - Not sound: Cannot completely rely on dynamic information

Dynamic data-flow analysis

```
int main(void)
{

    float x = 0.0;

    float y = 0.0;

    float z = 0.0;

    float a = 9.2;


    for (int i = 0; i < MAIN_LOAD_PLP;
    {
        x = calculate_pi();
        y = doSomeWork(x / 2);
        y = y + a;
        z = z + doSomeMoreWork(y / 4);
    }

    printf("z=%f\n", z);
    return 0;
}
```

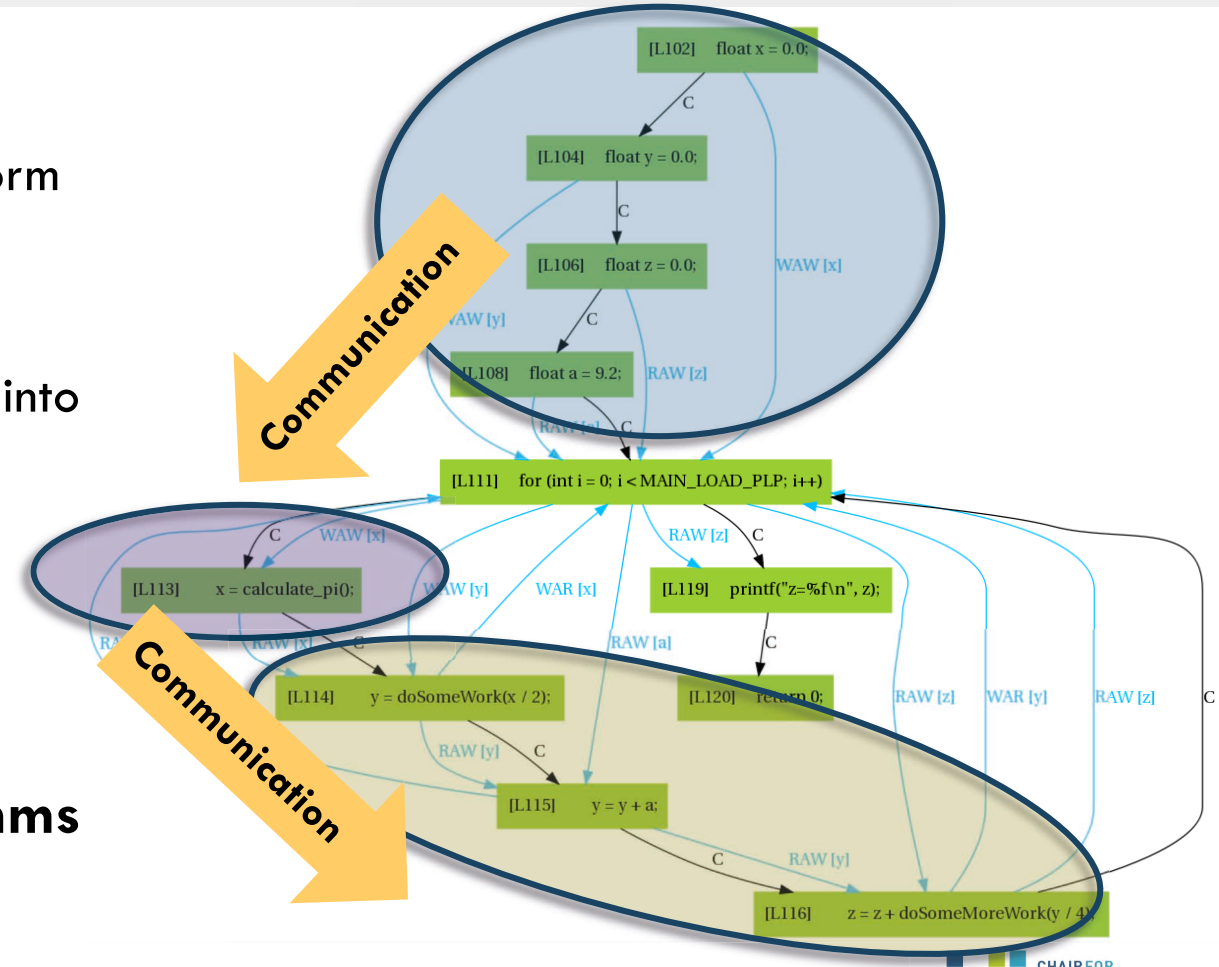


Data edges: annotated with actual amount of information transported

On granularity

- Granularity for analysis
  - Depends on the target platform
  - Whole-program information: Costs of called functions
  - Need to take communication into account
  - Is not given by traditional compiler boundaries: basic-blocks or functions

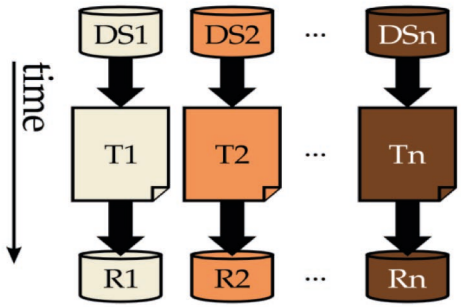
➔ **Use graph clustering algorithms**



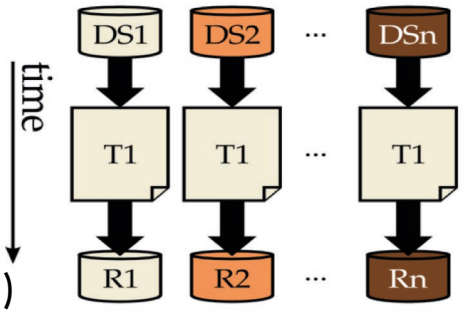
Coarse-grained parallelism patterns

- Search for known parallelism patterns
  - Task-level parallelism
  - Data-level parallelism
  - Pipeline-level parallelism
  - Others: Reduction, commutative operations, …

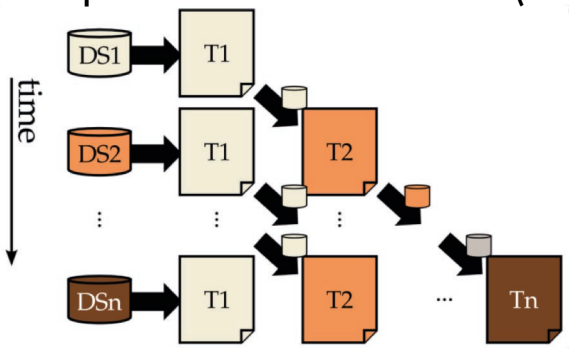
Task Level Parallelism (TLP)



Data Level Parallelism (DLP)

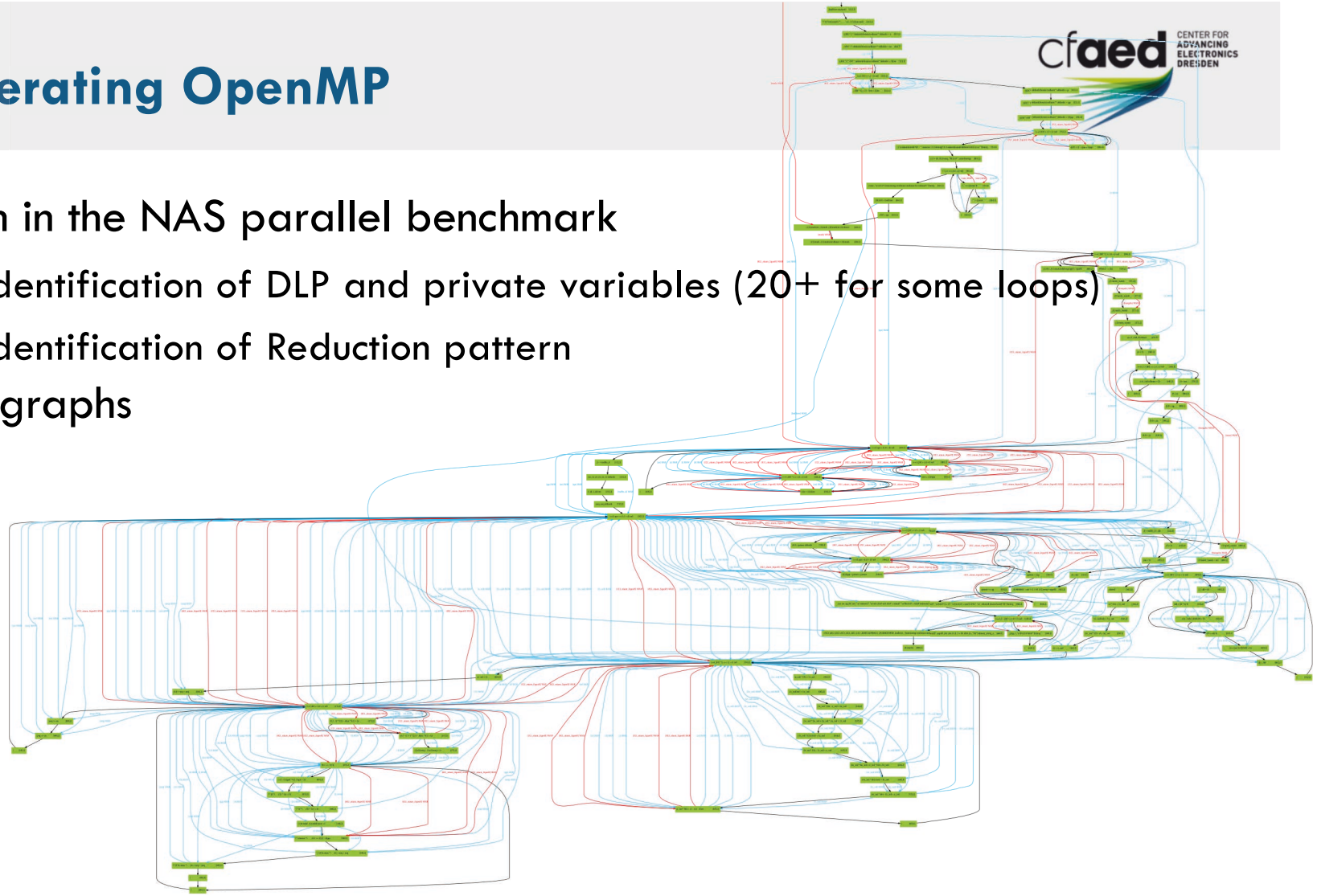


Pipeline Level Parallelism (PLP)



Example: Generating OpenMP

- EP: Application in the NAS parallel benchmark
  - Automatic identification of DLP and private variables (20+ for some loops)
  - Automatic identification of Reduction pattern
- Very complex graphs



Example: Generating OpenMP (2)

- Manually parallelized code
  - Multiple nesting
  - Multiple annotations

```
219 #pragma acc parallel loop reduction(+:sx,sy)
220         for (k = 1; k <= np; k++)
221         {
222             kk = k_offset + k;
223             t1 = S;
224             t2 = an;
225
226 /*      Find starting seed t1 for this kk. */
227 #pragma acc loop seq
228             for (i = 1; i <= 100; i++)
229             {
230                 ik = kk / 2;
```

OpenMP Annotated Code

Reference pragma

Reduction Correctly Identified

```
#pragma acc parallel loop reduction(+:sx,sy)
        // Loop automatically annotated with OpenMP
#pragma omp parallel for private(i,ik,kk,l,...,psx,psy,t1,t2,t3,t4,x1,x2) reduction(+:sx,sy)
        for (k = 1; k <= np; k++)
        {
            k
            t1 = S;
```

Generated pragma

Several Private Variables

```
252             loc_t1 = loc_a1 * loc_x2 + loc_a2 * loc_x1;
253             loc_t2 = (int)(r23 * loc_t1);
254             loc_z = loc_t1 - t23 * loc_t2;
```

Problems with auto-parallelization

- Difficult to find all dependencies (often impossible at compile-time)
- Coding style and the illusion of infinite shared memory
- Dependencies can sometimes be violated!

```
   for (i = 1; i <= 100; i++)
      for (j = 1; j <= 100; j++) {
S1:     X[i][j] = X[i][j] + Y[i-1][j];
S2:     Y[i][j] = Y[i][j] + X[i][j-1];
      }
```

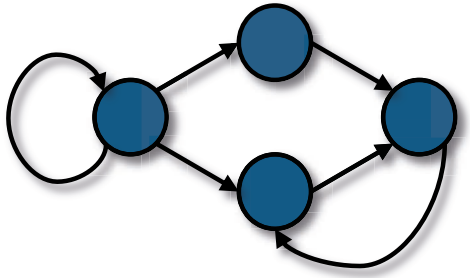
```
19  while(!queue.empty())
20  {
21    // Dequeue a vertex from queue
22    s = queue.front();
23    queue.pop_front();
24
25    // Apply function f to s, accumulate values
26    result += f(s);
27
28    // Get all adjacent vertices of s.
29    // If an adjacent node hasn't been visited,
30    // then mark it as visited and enqueue it
31    for(i=adj[s].begin(); i!=adj[s].end(); ++i)
32    {
33      if(!visited[*i])
34      {
35        visited[*i] = true;
36        queue.push_back(*i);
37      }
38    }
39  }
40
41  return result;
42 }
```

Multi-cores: Parallel programming

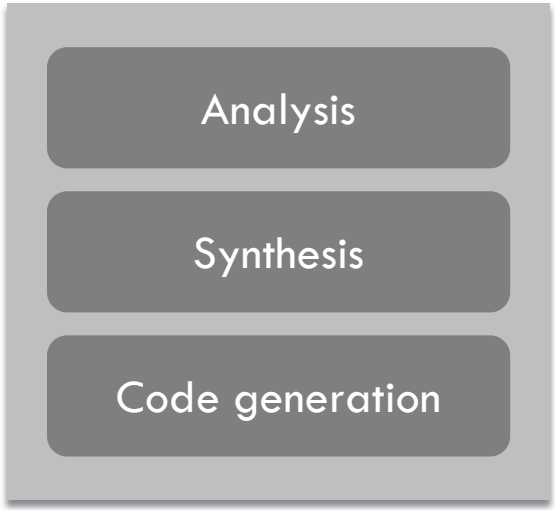
- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages

Sample compiler flow

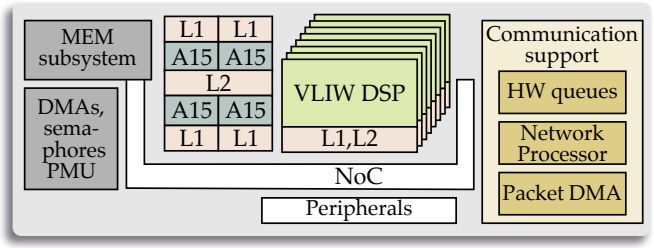
KPN Application



Non-functional specification



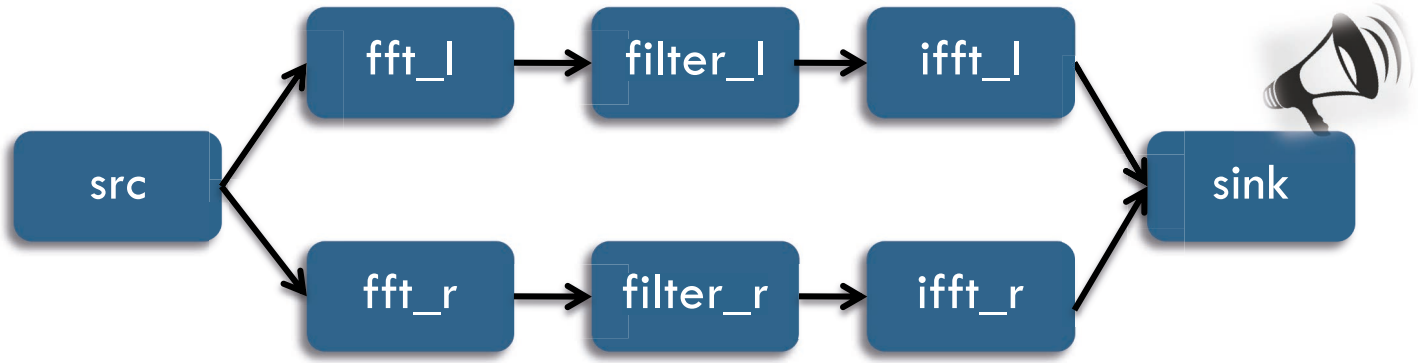
Architecture model

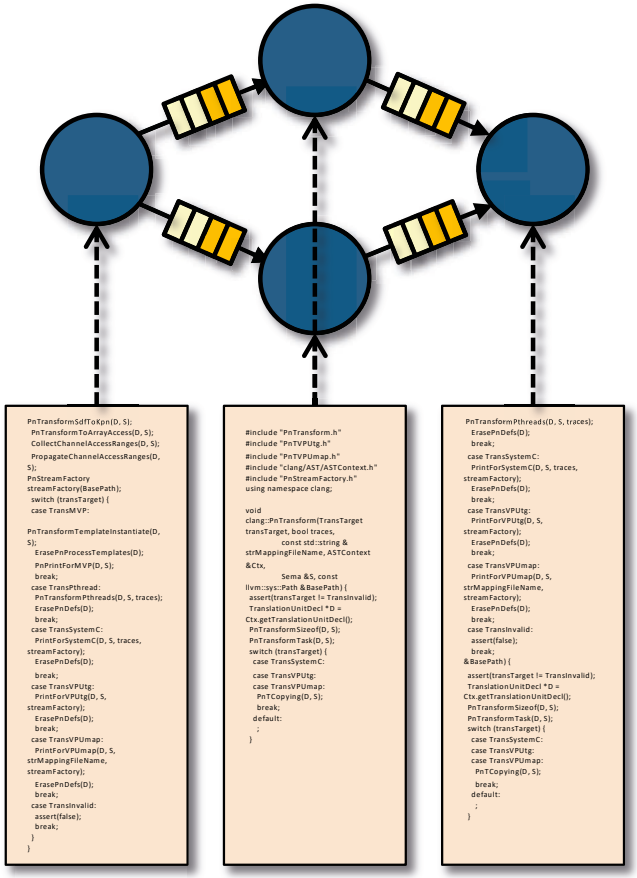



```
PNargs_ifft_r.ID = 6U;
    PNargs_ifft_r.PNchannel_freq_coef = filtered_coe
    PNargs_ifft_r.PNnum_freq_coef = 0U;
    PNargs_ifft_r.PNchannel_time_coef = sink_right;
    PNargs_ifft_r.channel = 1;
    sink_left = IPCllmrf_open(3, 1, 1);
    sink_right = IPCllmrf_open(7, 1, 1);
    PNargs_sink.ID = 7U;
    PNargs_sink.PNchannel_in_left = sink_left;
    PNargs_sink.PNnum_in_left = 0U;
    PNargs_sink.PNchannel_in_right = sink_right;
    PNargs_sink.PNnum_in_right = 0U;
    taskParams.arg0 = (xdc_UArg)&PNargs_src;
    taskParams.priority = 1;

ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_Func
&taskParams, &eb);
    glob_proc_cnt++;
    hasProcess = 1;
    taskParams.arg0 = (xdc_UArg)&PNargs_fft_l;
    taskParams.priority = 1;

ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_Func
ft_Templ, &taskParams, &eb);
    glob_proc_cnt++;
    hasProcess = 1;
    taskParams.arg0 = (xdc_UArg)&PNargs_ifft_r;
    taskParams.priority = 1;

ti_sysbios_knl_Task_create((ti_sysbios_knl_Task_Func
fft_Templ, &taskParams, &eb);
    glob_proc_cnt++;
    hasProcess = 1;
    taskParams.arg0 = (xdc_UArg)&PNargs_sink;
    taskParams.priority = 1;
```

Kahn Process Networks (KPNs)

- Graph representation of applications
  - Processes **communicate only** over FIFO buffers
  - Good model for streaming applications
  - Good match for signal processing & multi-media (cross-domain)
- Stereo digital audio filter





Language: C for process networks

- FIFO Channels

```
typedef struct { int i; double d; } my_struct_t;
__PNchannel my_struct_t S;
__PNchannel int A = {1, 2, 3}; /* Initialization */
__PNchannel short C[2], D[2], F[2], G[2];
```

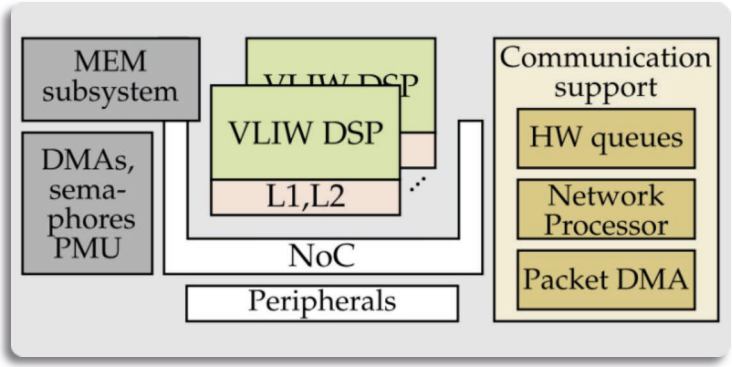
- Processes & networks

```
__PNkpn AudioAmp __PNin(short A[2]) __PNout(short B[2])
                 __PNparam(short boost){
  while (1)
    __PNin(A) __PNout(B) {
      for (int i = 0; i < 2; i++)
        B[i] = A[i]*boost;
    }}
__PNprocess Amp1 = AudioAmp __PNin(C) __PNout(F) __PNparam(3);
__PNprocess Amp2 = AudioAmp __PNin(D) __PNout(G) __PNparam(10);
```

W. Sheng, et al., "A compiler infrastructure for embedded heterogeneous MPSoCs", Parallel Comput. 40, 2. 51-68. 2014

Architecture model for heterogeneity

- System model including:
  - Topology, interconnect, memories
  - Computation: cost tables (as backup)
  - Communication: cost function (no contention)
- Example: Texas Instruments Keystone



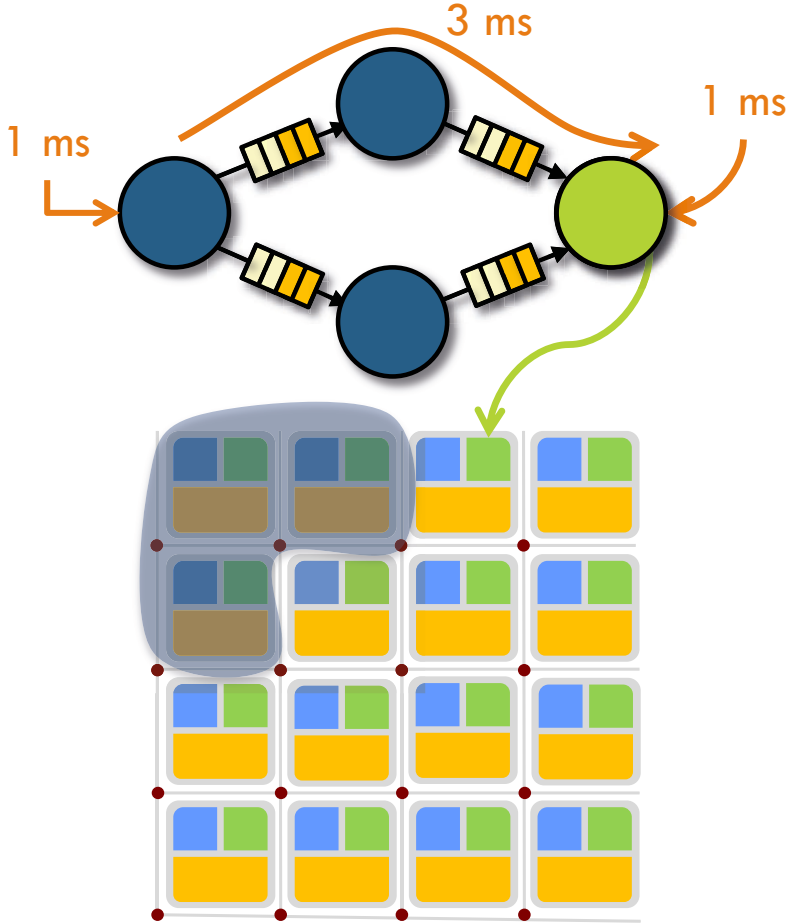

```
-<Platform>
   <Processors List="dsp0 dsp1 dsp2 dsp3 dsp4 dsp5 dsp6 dsp7"/>
   <Memories List="local_mem_dsp0_L2 local_mem_dsp1_L2 local_mem_dsp2_L2
   local_smem_dsp1_L2 local_smem_dsp2_L2 local_smem_dsp3_L2 local_smem_dsp
   local_mem_dsp3_DDR local_mem_dsp4_DDR local_mem_dsp5_DDR local_mem_ds
   <CommPrimitives List="IPCll_SL2 IPCll_DDR EDMA3_SL2 EDMA3_DDR EDMA
 </Platform>
 <Processor Name="dsp0" CoreRef="DSPC66"/>
 <Processor Name="dsp1" CoreRef="DSPC66"/>
    ...
 <Processor Name="dsp7" CoreRef="DSPC66"/>
-<Memory>
   <LocalMemory Name="local_mem_dsp0_L2" Size="524288" BaseAddress_hex="00800000" ProcessorRef="dsp0"/>
 </Memory>
    ...
```

```
-<Core Name="DSPC66" CoreType="DSPC66" Category="DSP">
 -<MultiTaskingInfo MaxNumberOfTasks="-1">
   <ContextSwitchInfo StoreTime="1000" LoadTime="1000"/>
   <SchedulingPolicies List="FIFO PriorityBased"/>
  </MultiTaskingInfo>
 -<CostTable>
  -<Operation Name="Load">
   -<VariableType Name="Char">
     <Cost>1</Cost>
    </VariableType>
   -<VariableType Name="Double">
```

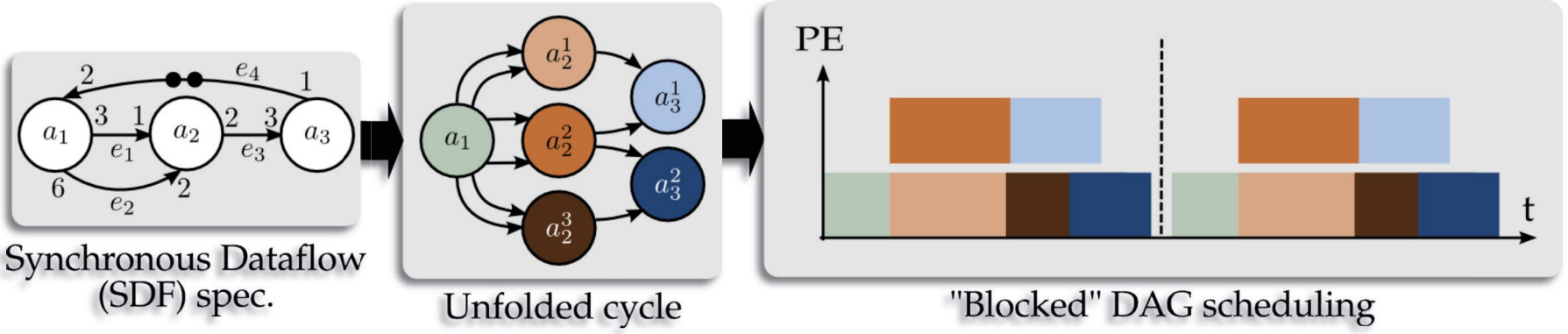
M. Odendahl, et al., "Split-cost communication model for improved MPSoC application mapping", In International Symposium on System on Chip (SoC) pp. 1-8, 2013

Constraints

- Timing constraints
  - Process throughput
  - Latencies along paths
  - Time triggering
- Mapping constraints
  - Processes to processors
  - Channels to primitives
- Platform constraints
  - Subset of resources (processors or memories)
  - Utilization

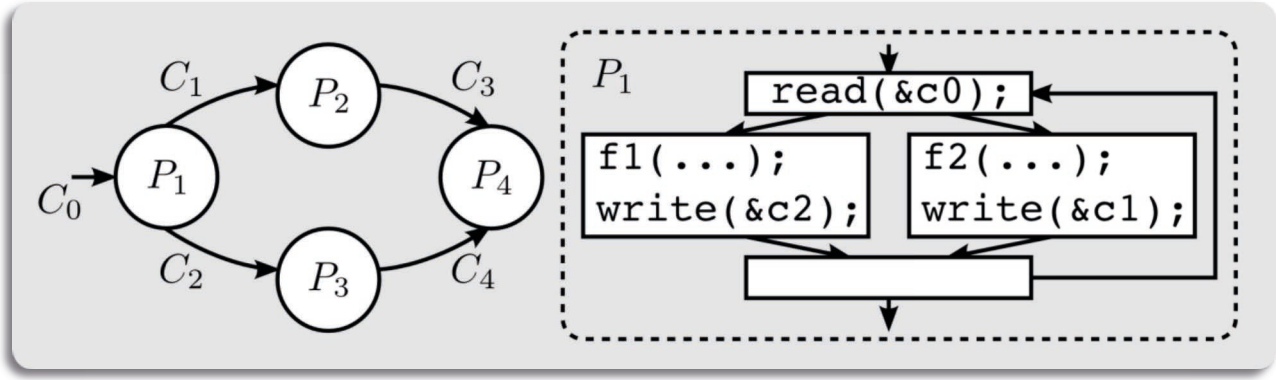


Static models: Synchronous Dataflow (SDF)



- Fully specified rates, allow more compiler analysis
- Transformation: **repetition vector** serve to unroll the graph [1 3 2]
- Perform mapping and scheduling on the resulting **directed acyclic graph** (DAG)

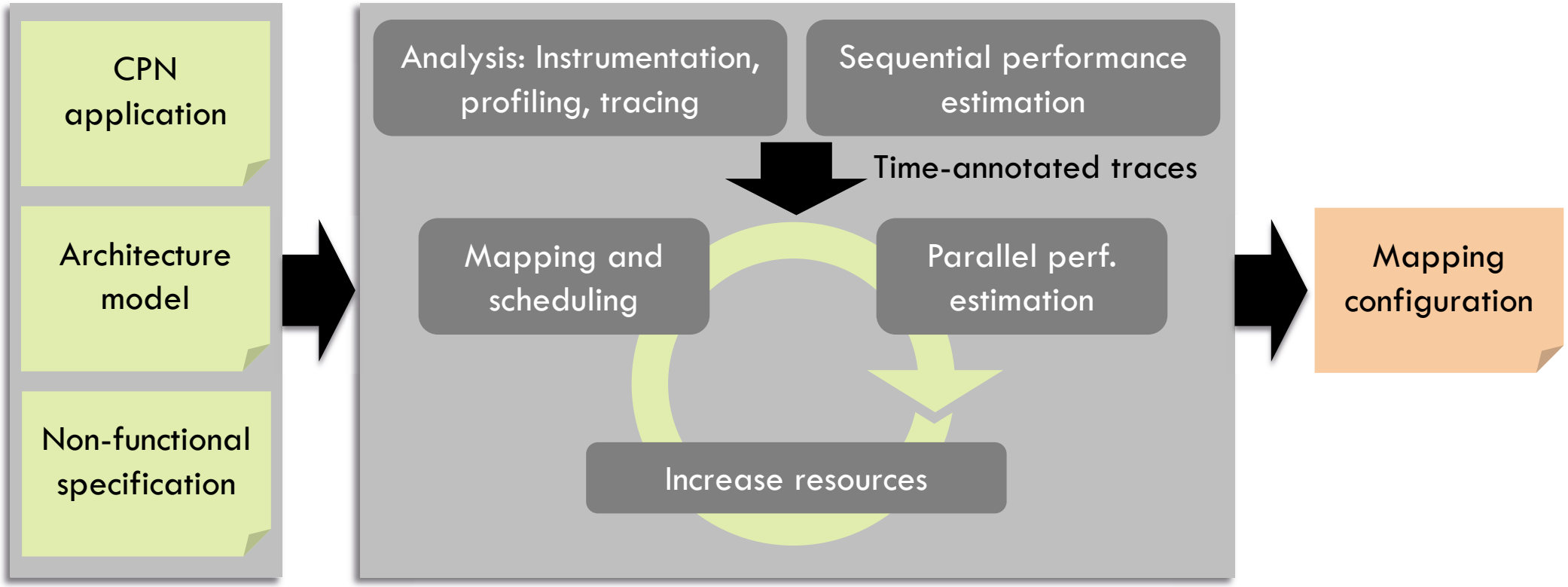
Dynamic models: KPNs



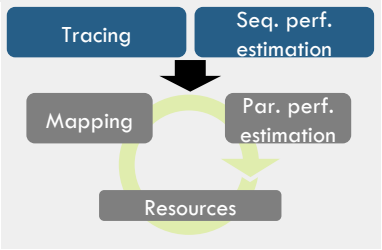
- Channel accesses not visible at the graph
- ➔ Need to look inside the processes

- Solutions: Use dynamic scheduling
- Methods: Employ simulations, genetic algorithms or devise heuristics

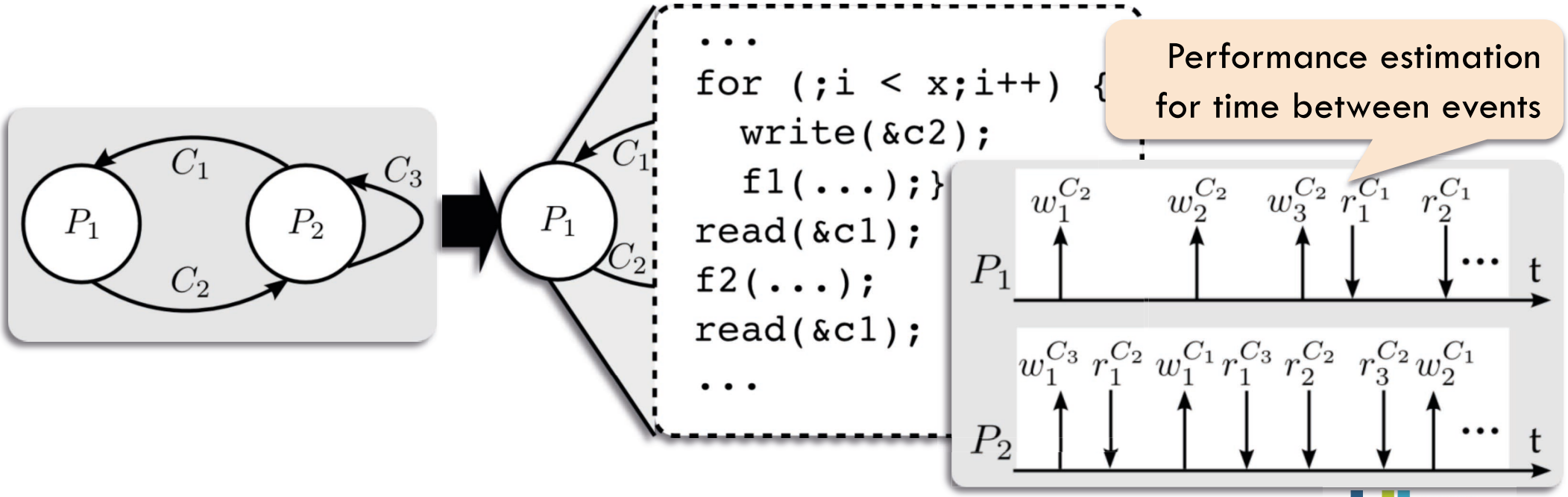
Analysis and synthesis: Overview



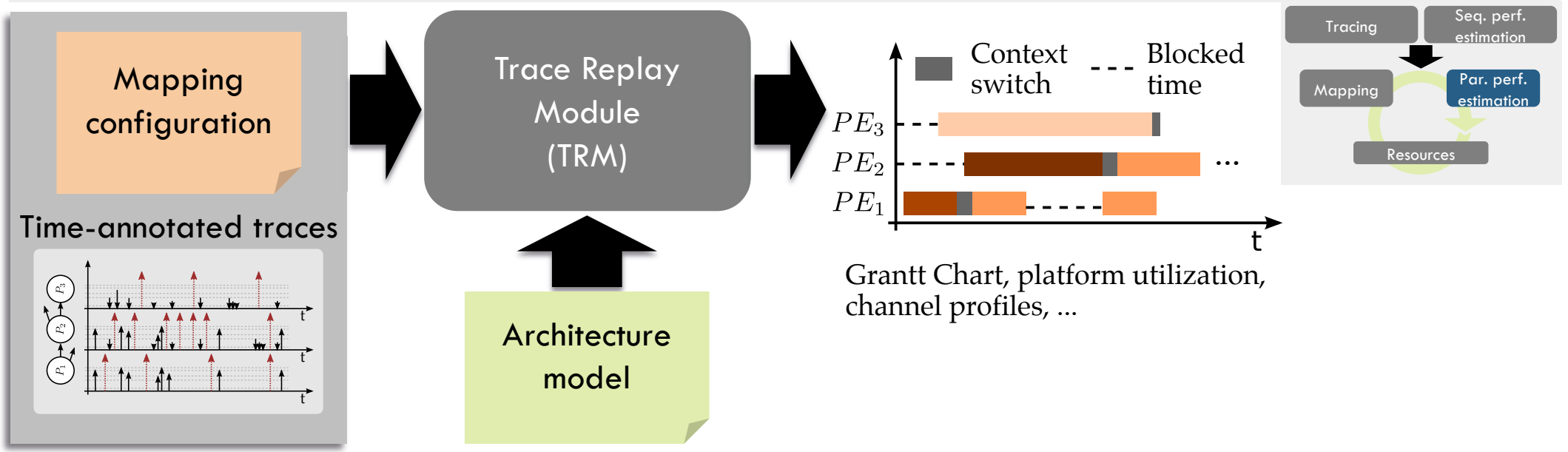
Tracing: Dealing with dynamic behavior



- KPNs do not have firing semantics
- **White model of processes**: source code analysis and tracing
- Tracing: instrumentation, token logging and event recording

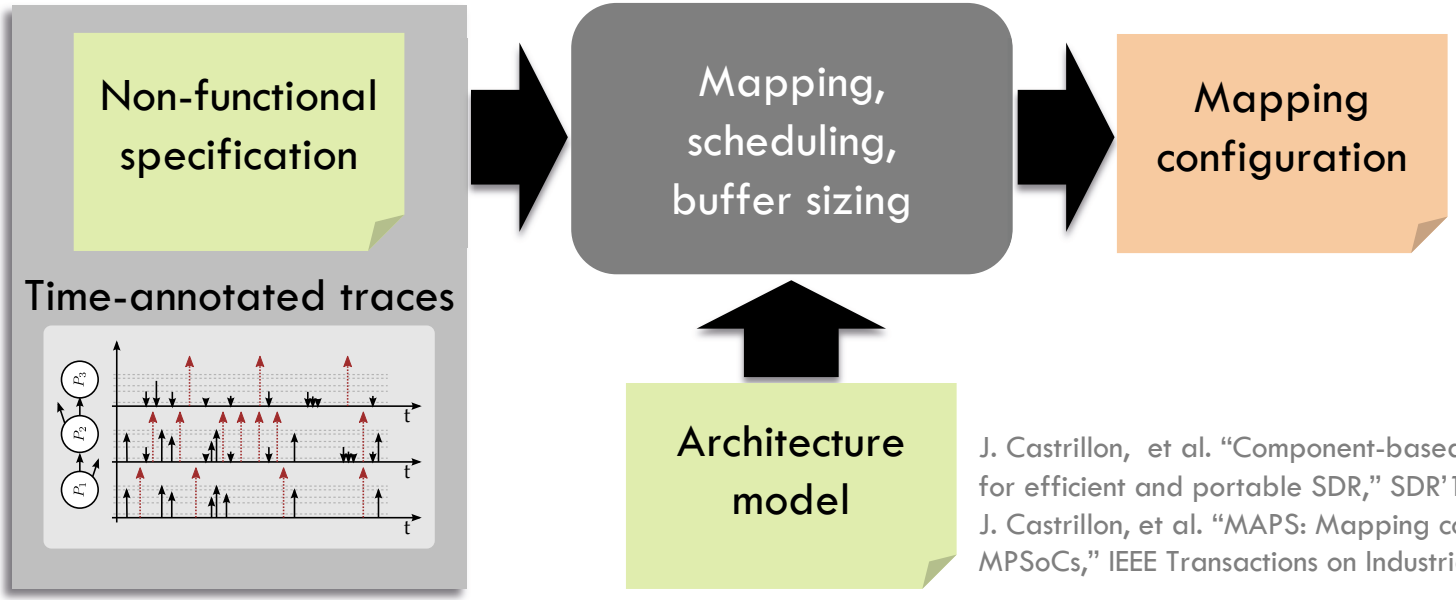


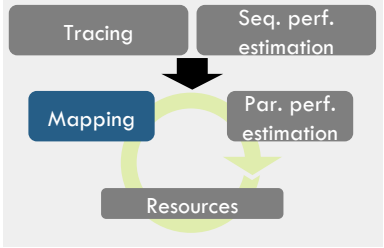
Parallel performance estimation



- Discrete event simulator to evaluate a solution
  - Replay traces according to mapping
  - Extract costs from architecture file (NoC modeling, context switches, communication)

Trace-based synthesis

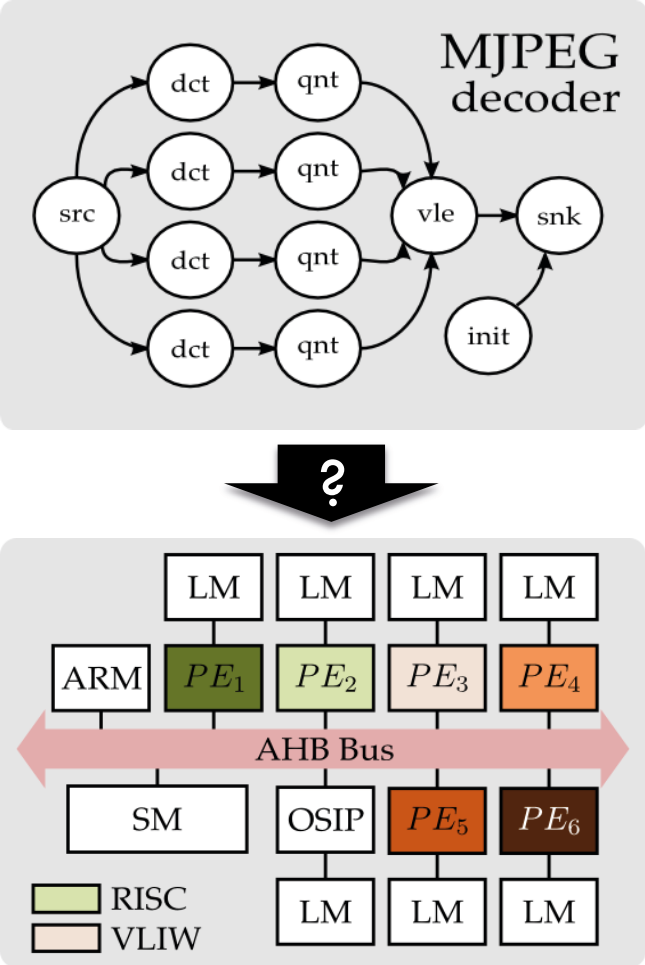


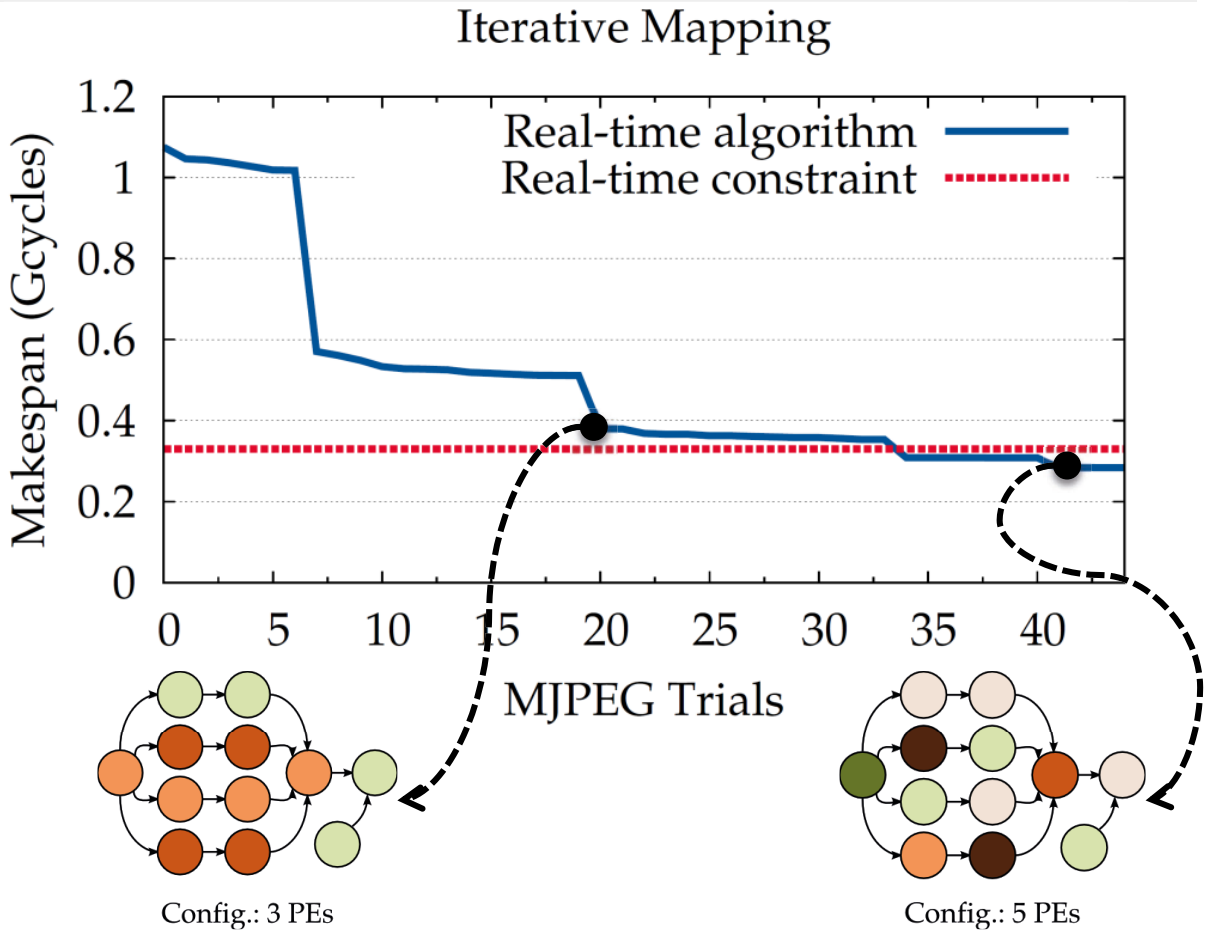


J. Castrillon, et al. "Component-based waveform development: The nucleus tool flow for efficient and portable SDR," SDR'10
J. Castrillon, et al. "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," IEEE Transactions on Industrial Informatics, vol. 9, no. 1, pp. 527–545, 2013

- Synthesis based on code and trace analysis (using simple heuristics)
  - Mapping of processes and channels
  - Scheduling policies
  - Buffer sizing

Sample results from mapping exploration

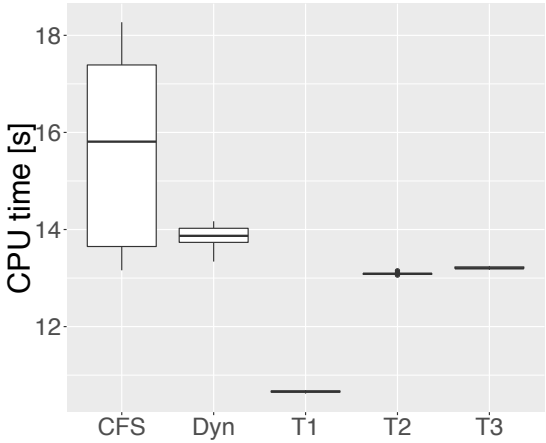


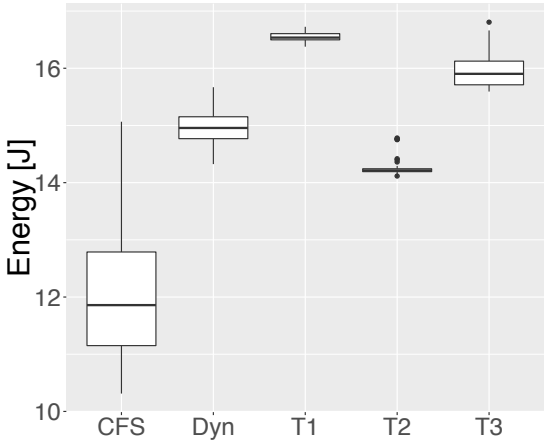


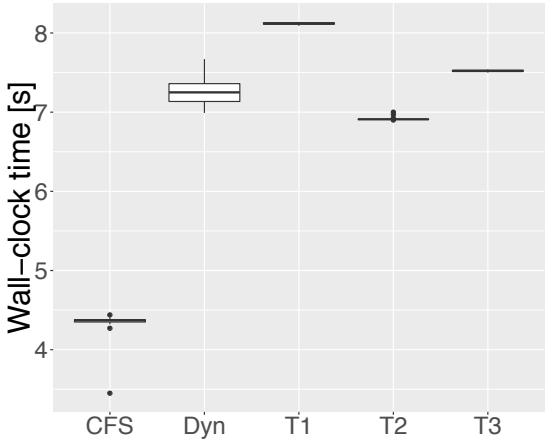
J. Castrillon, "Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap", Springer, 2014

Flexible mappings: Run-time analysis

- Modified linux kernel to make it aware of symmetries
- Platform: Odroid XU4 (big.LITTLE)
- Multi-application scenarios: audio filter (AF) and MIMO
  - 1x AF,
  - 4 x AF
  - 2 x AF + 2 x MIMO
- 3 mappings to two processors
  - T1: Best CPU time
  - T2: Best wall-clock time
  - T3: GBM heuristic



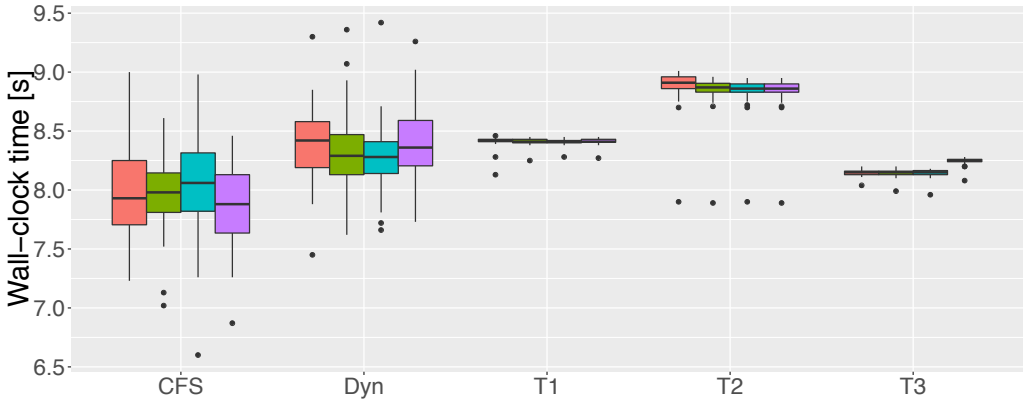


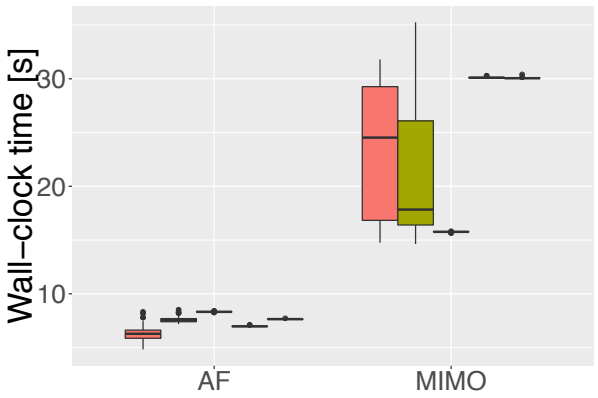


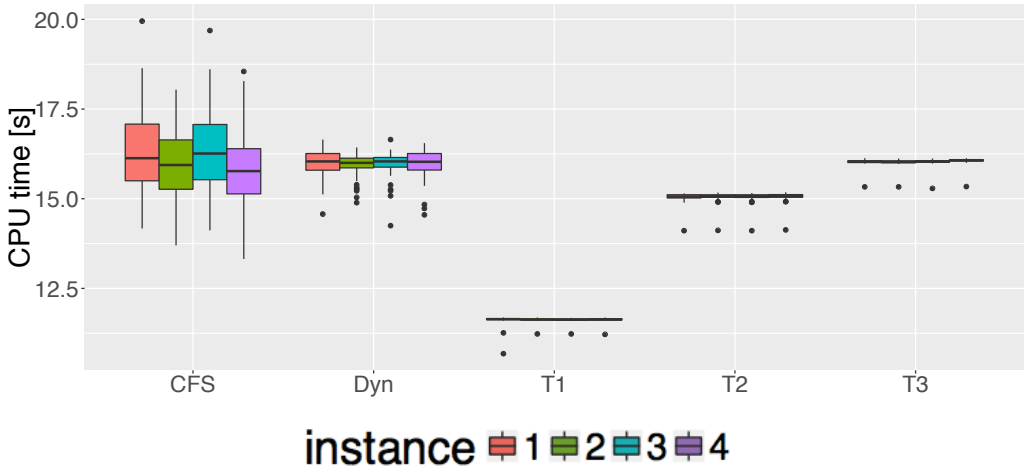
Single AF

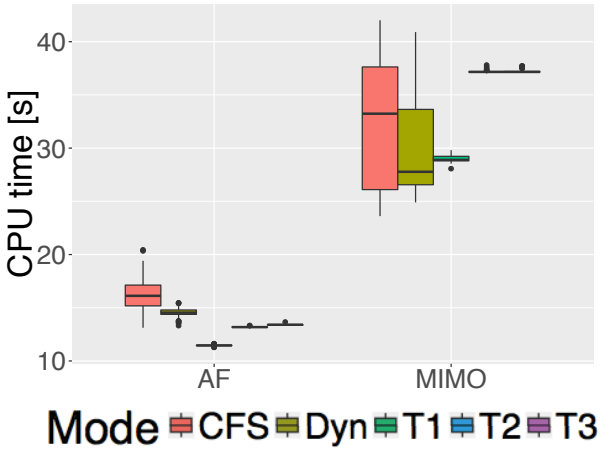
Goens, A. et al. "TETRiS: a Multi-Application Run-Time System for Predictable Execution of Static Mappings", SCOPES'17

Flexible mappings: Multi-application results (1)









Way more predictable performance

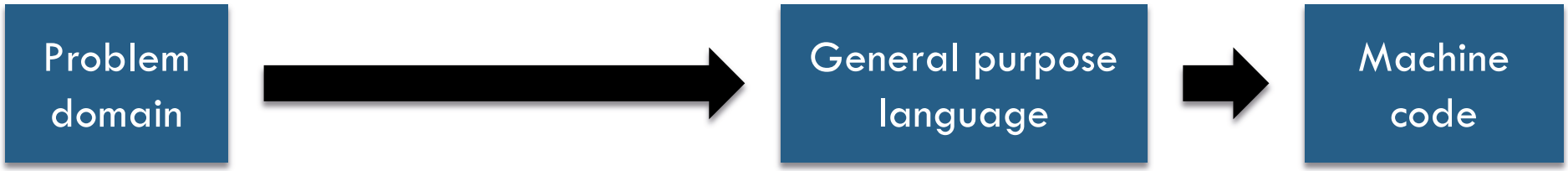
Comparable performance to dynamic mapping

Multi-cores: Parallel programming

- Introduction
- Auto-parallelization
- Dataflow programming
- Domain-specific languages

What are DSLs?

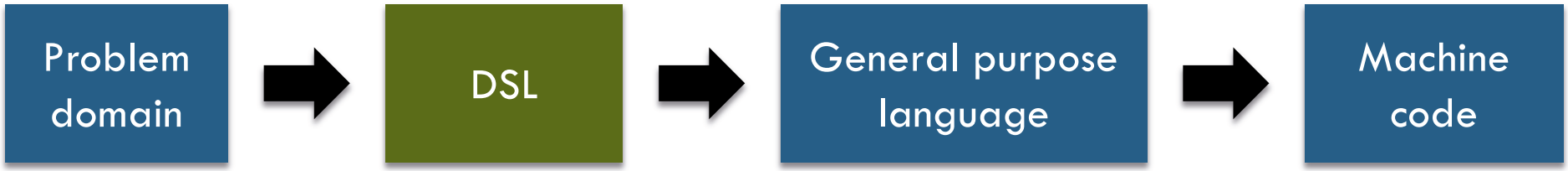
- DSLs help bridge the gap between problem domain and general purpose languages



Adapted from lecture: "Concepts of Programming Languages", Eelco Visser, TU Delft

What are DSLs?

- DSLs help bridge the gap between problem domain and general purpose languages



Adapted from lecture: "Concepts of Programming Languages", Eelco Visser, TU Delft

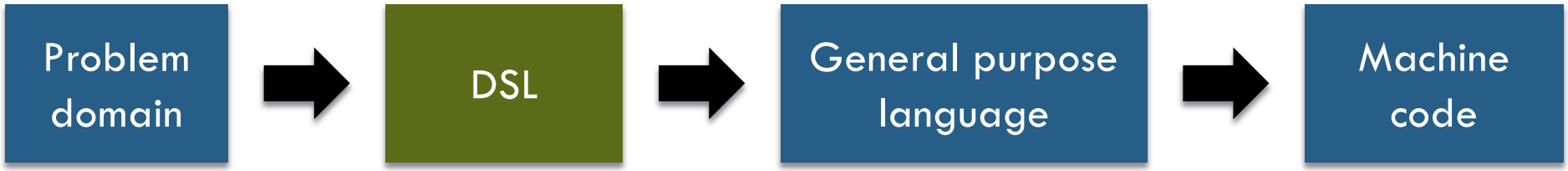
  - Natural vocabulary for concepts are fundamental to problem domain
  - Faster way to write common concepts (**concrete syntax**)
  - Explanation is a domain-specific, task-specific, user-specific inference

Wick, Michael R., and William B. Thompson. "Reconstructive expert system explanation." Artificial Intelligence 54.1-2 (1992): 33-70.

  - Optimization potential due to domain-specific information

Why DSLs now?

- DSLs help bridge the gap between problem domain and general purpose languages



Adapted from lecture: "Concepts of Programming Languages", Eelco Visser, TU Delft

- Programming permeating other disciplines with their own vocabulary
  - "High-level" language receives a different connotation
- Complexity of hardware, low-level APIs, low-level software
- Push from different domains abusing existing less powerful tools (e.g., XML)
- Maturity of tools (interpreters, compilers, IDEs, code generators)

DSL: Definitions

A **DSL** is a computer language **specialized** to a particular application domain

Source: Wikipedia

A **DSL** is a programming language or executable specification language that offers, through **appropriate notations and abstractions,** expressive power focused on, and usually restricted to, a particular problem domain

A. Van Deursen, et al. "Domain-specific languages: An annotated bibliography." ACM Sigplan Notices35.6 (2000)

A **DSL** is a computer programming language of **limited expressiveness** focused on a particular domain

"Domain-Specific Languages" by Martin Fowler, Terry White, Addison-Wesley Professional. September 23, 2010. Print ISBN-10: 0-321-71294-3

DSLs at the Chair for Compiler Construction

- Working on different fronts, touching different topics
  - Performance: Avoid the *"abstraction toll"*
  - Type systems, operational and denotational semantics
- Ohua
  - Functional abstraction on top of dataflow execution models
  - Applicable to system's programming
- PPME: Parallel particle-mesh programming environment
  - For particle-based simulation
  - Applicable, e.g., for computational biology
- CFDlang: for computational fluid dynamics
  - Tensors and tensor operations in the syntax

Ohua: Implicit parallelism for systems

- Functional programming abstraction to implicitly create dataflow graphs
  - Functional program: High-level algorithm (clojure)
  - State-full functions: actual application logic (clojure, java, scala, …)

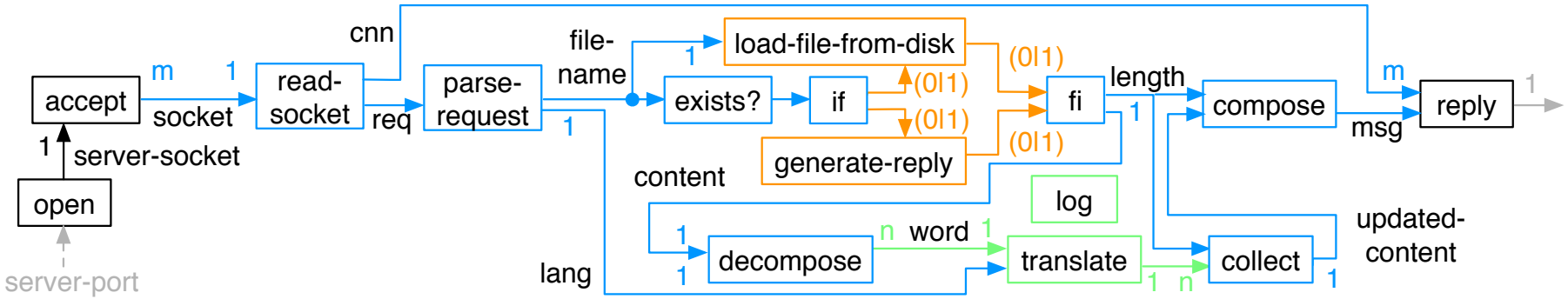
```
1  (ohua :import [web.translation]) ; import the namespace where the used
2                                   ; functions are defined
3  (defn translate [server-port]
4    (ohua (let [[cnn req] (read-socket (accept (open server-port)))
5                [_ file-name _ lang] (parse-request req)
6                [^List content length] (if (exists? file-name)
7                                           (load-file-from-disk file-name)
8                                           (generate-reply "No such file."))
9                ^String word (decompose content) ; poor man's translation
10               _ (log "translating word")
11               updated-content (collect length (translate word lang))]
12           (reply cnn (compose length updated-content)))
```

Sebastian Ertel, Justus Adam, Jeronimo Castrillon, "Supporting Fine-grained Dataflow Parallelism in Big Data Systems" , PMAM'18, ACM, pp. 41–50, New York, NY, USA, Feb 2018

Ohua compiler

- Clean language semantics and state characterization allow automatic derivation of dataflow graph

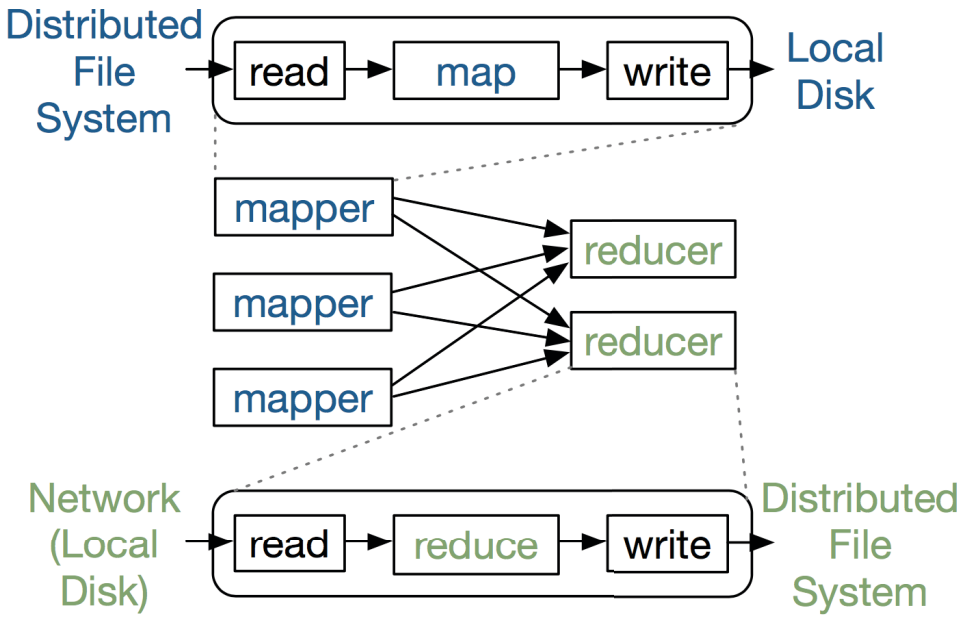
```
(ohua :import [web.translation]) ; import the namespace where the used
                                 ; functions are defined
(defn translate [server-port]
  (ohua (let [[cnn req] (read-socket (accept (open server-port)))
              [_ file-name _ lang] (parse-request req)
              [^List content length] (if (exists? file-name)
                                         (load-file-from-disk file-name)
                                         (generate-reply "No such file."))
              ^String word (decompose content) ; poor man's translation
              _ (log "translating word")
              updated-content (collect length (translate word lang))]
          (reply cnn (compose length updated-content)))
```

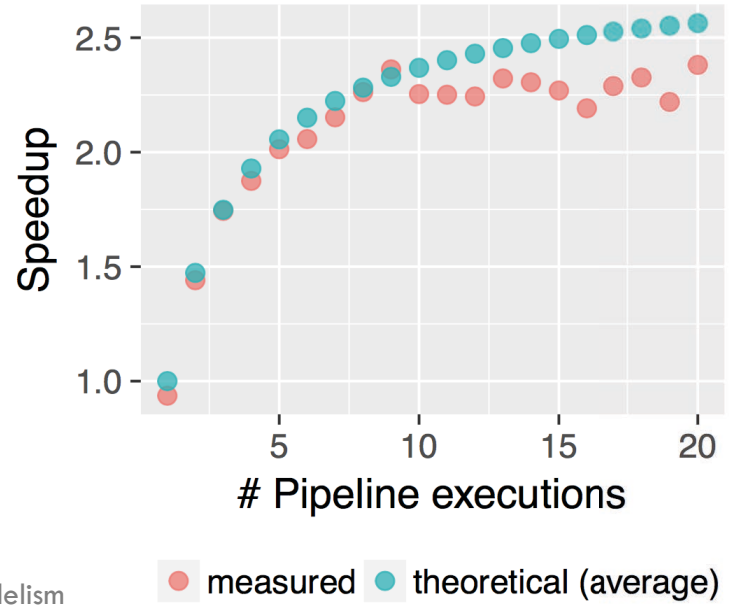


Sebastian Ertel, Justus Adam, Jeronimo Castrillon, "Supporting Fine-grained Dataflow Parallelism in Big Data Systems" , PMAM'18, ACM, pp. 41–50, New York, NY, USA, Feb 2018

Ohua applications: Map & Reduce

- Data processing pipeline within Hadoop
- Automatic exploitation of parallelism by compiler and runtime

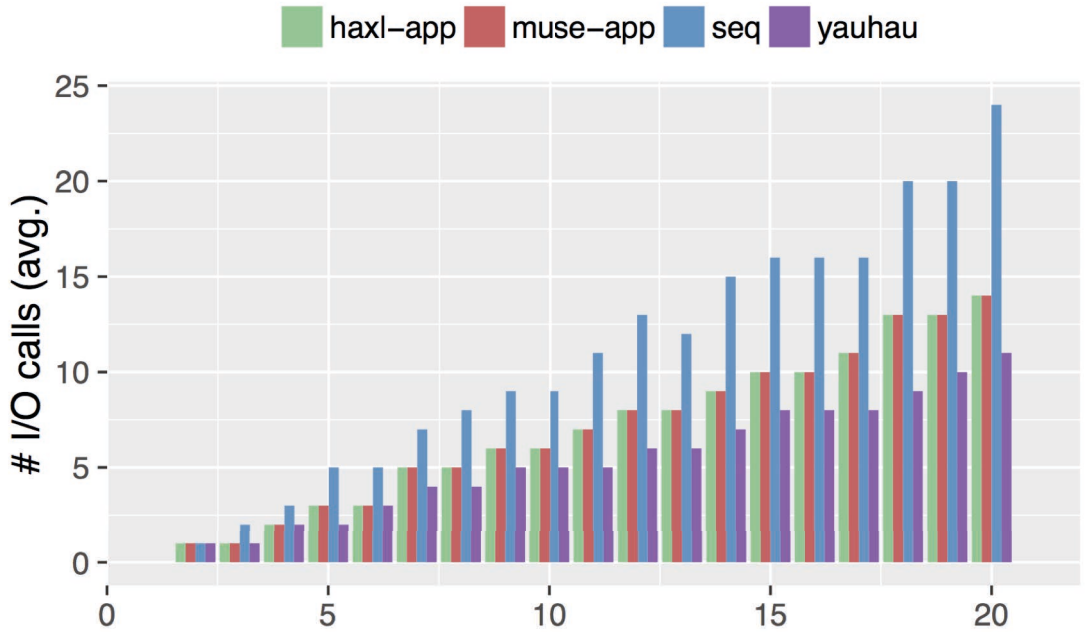




Sebastian Ertel, Justus Adam, Jeronimo Castrillon, "Supporting Fine-grained Dataflow Parallelism in Big Data Systems" , PMAM'18, ACM, pp. 41–50, New York, NY, USA, Feb 2018

Ohua applications: Micro-services

- Automatic batching of I/O in micro-services: Via graph rewrites
- Similar performance than e.g., Facebook, with better code style

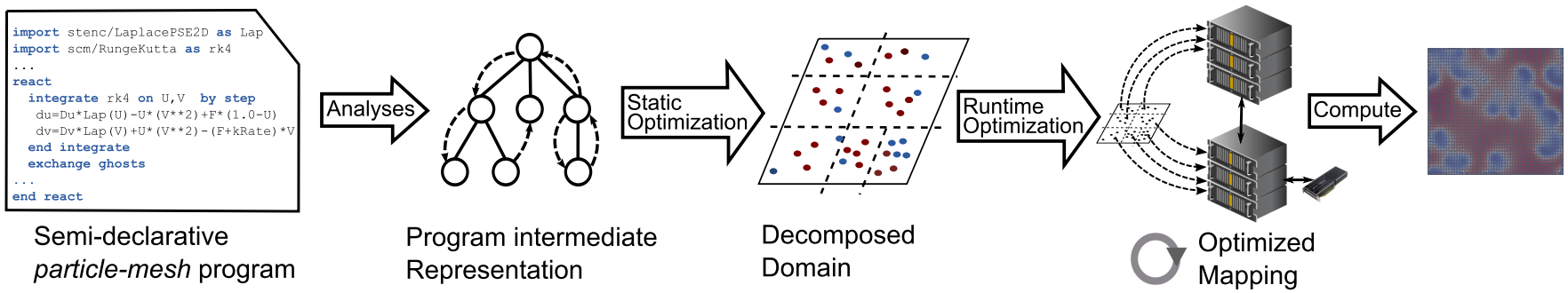


Sebastian Ertel, Andrés Goens, Justus Adam, Jeronimo Castrillon, "Compiling for Concise Code and Efficient I/O" , Proceedings of the 27th International Conference on Compiler Construction (CC 2018), ACM, pp. 104–115, New York, NY, USA, Feb 2018.

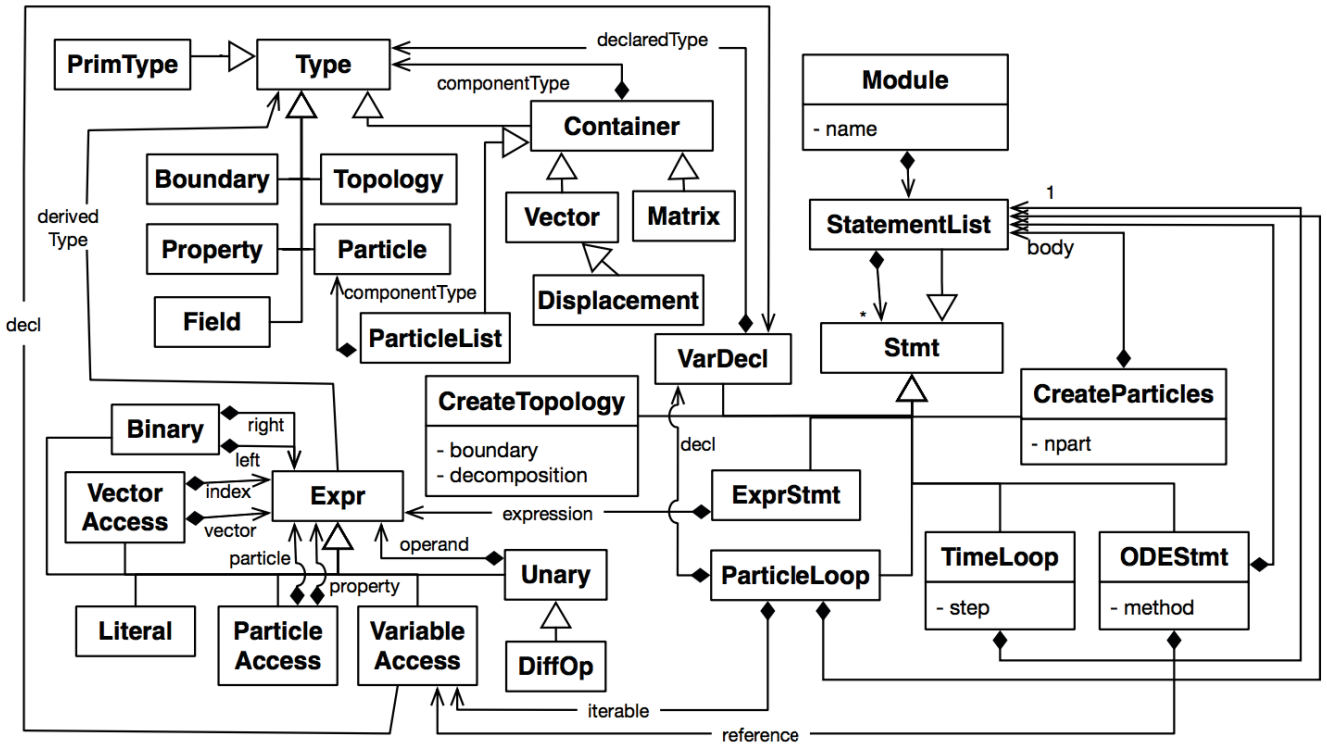
PPME: Parallel particle-mesh environment

- Domain: Python-based DSL for Particle-Mesh simulations
  - Parallel particle-mesh library (PPM): abstractions for **computational biology**
  - Collaboration with the mosaic group (http://mosaic.mpi-cbg.de/)

- Provide more information about structure, data-usage and computation patterns to the compiler for parallel execution



PPME: Data model



Sven Karol, Tobias Nett, Jeronimo Castrillon, Ivo F. Sbalzarini, "A Domain-Specific Language and Editor for Parallel Particle Methods" , In ACM Transactions on Mathematical Software (TOMS), ACM, vol. 44, no. 3, pp. 32, New York, NY, USA, Mar 2018.

PPME: Type system

- High-level semantics checks (e.g., units)

$$\textsc{Var}\ \frac{\Gamma(v)=\tau}{\Gamma\vdash v:\tau}\qquad \textsc{VarDecl}\ \frac{}{\Gamma\vdash \tau\ x:\Gamma\cup\{x=\tau\}}\qquad \textsc{VarInit}\ \frac{\Gamma\vdash e:\tau' \quad \tau'\leq\tau}{\Gamma\vdash \tau\ x=e:\Gamma\cup\{x=\tau\}}$$

$$\textsc{Param}\ \frac{\Gamma\vdash e:\tau}{}\qquad \textsc{Assign}\ \frac{\Gamma\vdash x:\tau \quad \Gamma\vdash e:\tau' \quad \tau'\leq\tau}{}$$

$$\frac{\dfrac{\dfrac{\dfrac{\Gamma\vdash p{\rightarrow}F:[\mathbb{R},m\cdot a] \quad \Gamma\vdash mass:[\mathbb{R},m]}{\Gamma\vdash p{\rightarrow}F/mass:[\mathbb{R},a]}\ (1) \qquad \Gamma\vdash p{\rightarrow}a:[\mathbb{R},a]}{\Gamma\vdash p{\rightarrow}a+p{\rightarrow}F/mass:[\mathbb{R},a]}\ (2) \qquad \Gamma\vdash 0.5:[\mathbb{R},\emptyset]}{\Gamma\vdash 0.5*(p{\rightarrow}a+p{\rightarrow}F/mass):[\mathbb{R},a]}\ (3) \qquad \Gamma\vdash delta\_t^2:[\mathbb{R},t^2]}{\Gamma\vdash 0.5*(p{\rightarrow}a+p{\rightarrow}F/mass)*delta\_t^2:[\mathbb{R},a\cdot t^2]}\ (4)$$

$$\frac{\Gamma\vdash p{\rightarrow}v:[\mathbb{R},v] \qquad \Gamma\vdash 0.5*(p{\rightarrow}a+p{\rightarrow}F/mass)*delta\_t^2:[\mathbb{R},a\cdot t^2]}{\Gamma\vdash p{\rightarrow}v+0.5*(p{\rightarrow}a+p{\rightarrow}F/mass)*delta\_t^2:\mathbb{E}}\ (E)$$

$$(1)\ \tau_{/}(\mathbb{R},\mathbb{R})=\mathbb{R},\mathcal{I}_{/}(m\cdot a,m)=a \qquad (2)\ \tau_{+}(\mathbb{R},\mathbb{R})=\mathbb{R},\mathcal{I}_{+}(a,a)=a \qquad (3)\ \tau_{*}(\mathbb{R},\mathbb{R})=\mathbb{R},\mathcal{I}_{*}(\emptyset,a)=a$$
$$(4)\ \tau_{*}(\mathbb{R},\mathbb{R})=\mathbb{R},\mathcal{I}_{*}(a,t^2)=a\cdot t^2 \qquad (E)\ \tau_{+}(\mathbb{R},\mathbb{R})=\mathbb{R},\mathcal{I}_{+}(v,a\cdot t^2)=\bot$$

$$\textsc{ErrUnary}\ \frac{\Gamma\vdash e:\tau \quad \tau_{\ominus}(\tau)=\bot}{\Gamma\vdash \ominus(e):\mathbb{E}}\qquad \textsc{ErrBin}\ \frac{\Gamma\vdash e_1:\tau_1 \quad \Gamma\vdash e_2:\tau_2 \quad \tau_{\otimes}(\tau_1,\tau_2)=\bot}{\Gamma\vdash e_1\otimes e_2:\mathbb{E}}$$

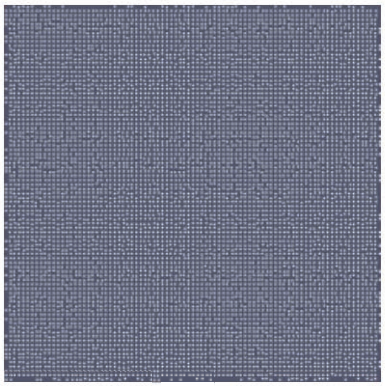
$$\ominus\in\{-,!,\sqrt{}\},\ \otimes_{arith}\in\{+,-,*,/,a^b\},\ \otimes_{log}\in\{\&\&,||\},\ \otimes_{rel}\in\{==,!=,<,>,<=,>=\}$$
$$\mathbb{Z}=Integer,\ \mathbb{R}=Real,\ \mathbb{P}=Particle,\ \mathbb{V}=Vector,\ \mathbb{M}=Matrix,\ \mathcal{E}=Field/Property,\ \mathbb{E}=Error$$

Sven Karol, Tobias Nett, Jeronimo Castrillon, Ivo F. Sbalzarini, "A Domain-Specific Language and Editor for Parallel Particle Methods" , In ACM Transactions on Mathematical Software (TOMS), ACM, vol. 44, no. 3, pp. 32, New York, NY, USA, Mar 2018.

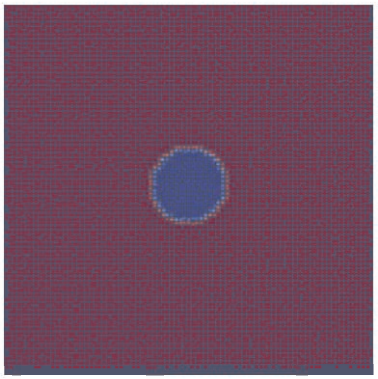
PPME: Language

- DSL: Closer to domain expert
- Easier to execute in parallel
- Integration with other tools for expression optimization

- Example: Gray-Scott simulation



$t = \emptyset$



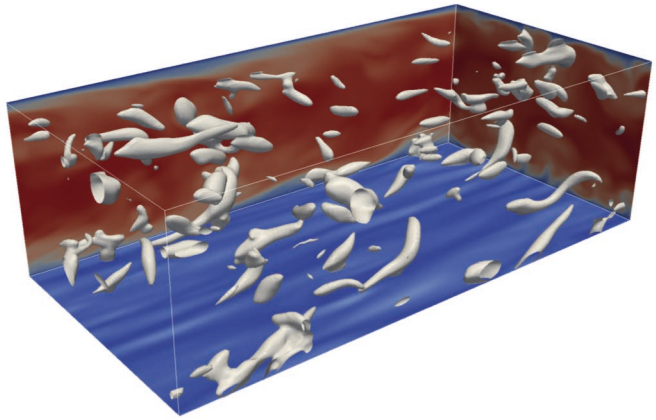
$t = 0$

GrayScott (single phase) ×

```
        foreach particle p in c do
            p→U = 1.0
            p→V = 0.0
            if (p→pos[1] - 0.5)^2 + (p→pos[2] - 0.5)^2 < 0.05_mk then
                p→U = 0.5 + 0.01 * random
                p→V = 0.25 + 0.01 * random
            end if
        end foreach

        create neighborlist n for c with
          skin: <no skin>
          cutoff 4.0 * c→hAvg
          symmetry <no symmetry>

        timeloop t do
            deqn method "rk4" on c
                ∂ c→U / ∂ t = constDu * ∇^2 c→U - c→U * c→V^2 + F * (1.0 - c→U)
                ∂ c→V / ∂ t = constDv * ∇^2 c→V + c→U * c→V^2 - (F + kRate) * c→V
            end deqn
            print c→U, c→V , 1
        end do timeloop t

    end all-in-one

end module GrayScott (single phase)
```

Sven
Trans

CFDlang for computational fluid dynamics (CFD)



- Tensor expressions typically occur in numerical codes

$$\mathbf{v}_e = (\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A})\, \mathbf{u}_e$$

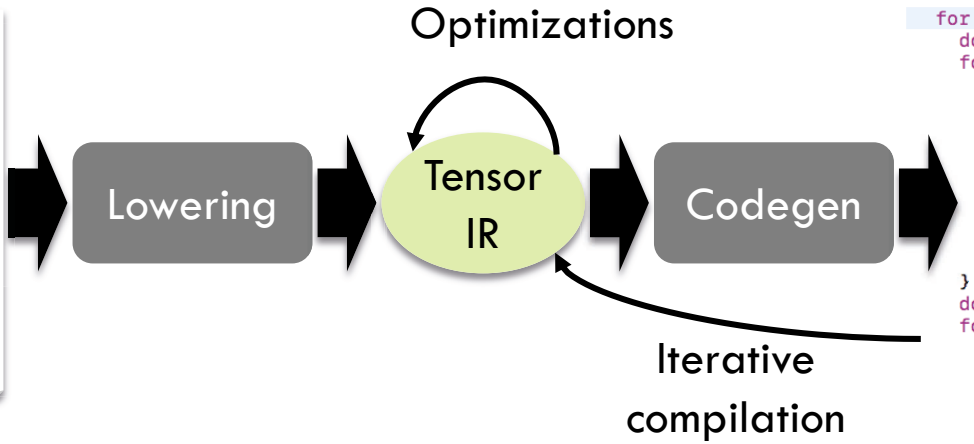
  - Tensor product notation popular in the CFD domain
- On performance
  - Matrixes are small, so libraries like BLAS don't always help
  - Expressions result in deeply nested for-loops
  - Performance highly depend on the *shape* of the loop nests

- **No need to do complex polyhedral analysis if we know the tensors and the semantics of the operators!**

CFDlang and tool flow

```
source =
type matrix    : [mp np]          &
type tensorIN  : [np np np ne]    &
type tensorOUT : [mp mp mp me]    &
                                  &
var input A    : matrix           &
var input u    : tensorIN         &
var input output v  : tensorOUT   &
var input alpha : []              &
var input beta  : []              &
                                  &
v = alpha * (A # A # A # u .
    [[5 8] [3 7] [1 6]]) + beta * v
```

Fortran embedding



Linkable C code

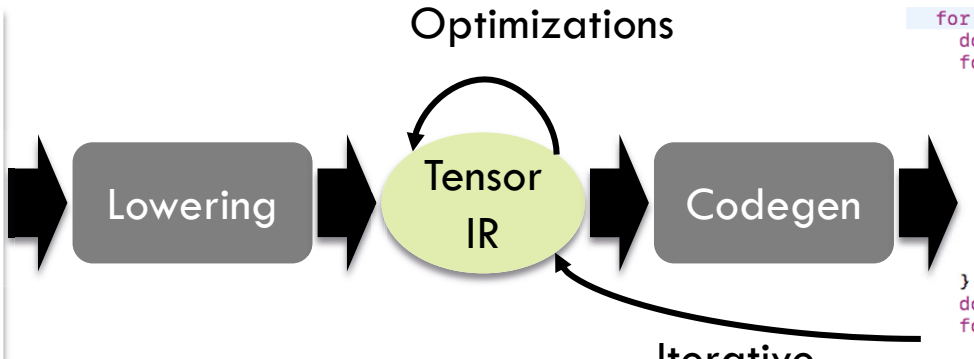
```
for (unsigned i0 = 0; i0 < 1000; i0++) {
  double t6[18];
  for (unsigned i3 = 0; i3 < 3; i3++) {
    for (unsigned i2 = 0; i2 < 3; i2++) {
      for (unsigned i1 = 0; i1 < 2; i1++)
        t6[(i1 + 2*(i2 + 3*(i3)))] = 0.0;
        for (unsigned i4_contr = 0; i4_contr < 3; i4_contr++) {
          t6[(i1 + 2*(i2 + 3*(i3)))] += A[(i1 + 2*(i4_contr))]
              * u[(i2 + 3*(i3 + 3*(i4_contr + 3*(i0))))];
        }
      }
    }
  }
  double t7[12];
  for (unsigned i7 = 0; i7 < 3; i7++) {
    for (unsigned i6 = 0; i6 < 2; i6++) {
      for (unsigned i5 = 0; i5 < 2; i5++) {
        t7[(i5 + 2*(i6 + 2*(i7)))] = 0.0;
        for (unsigned i8_contr = 0; i8_contr < 3; i8_contr++) {
          t7[(i5 + 2*(i6 + 2*(i7)))] += A[(i5 + 2*(i8_contr))]
              * t6[(i6 + 2*(i7 + 3*(i8_contr)))];
        }
      }
    }
  }
  double t8[1];
  double t9[1];
  for (unsigned i11 = 0; i11 < 2; i11++) {
    for (unsigned i10 = 0; i10 < 2; i10++) {
      for (unsigned i9 = 0; i9 < 2; i9++) {
        t9[0] = 0.0;
        for (unsigned i12_contr = 0; i12_contr < 3; i12_contr++
            ) {
          t9[0] += A[(i9 + 2*(i12_contr))] * t7[(i10 + 2*(i11 +
              2*(i12_contr)))];
        }
        t8[0] = alpha[0] * t9[0];
        double t10[1];
        t10[0] = beta[0] * v[(i9 + 2*(i10 + 2*(i11 + 2*(i0))))]
            ;
        v[(i9 + 2*(i10 + 2*(i11 + 2*(i0))))] = t8[0] + t10[0];
      }
    }
  }
}
```

N. A. Rink, I. Huismann, A. Susungi, J. Castrillon, J. Stiller, J. Fröhlich, and C. Tadonki. “CFDlang: High-level code generation for high-order methods in fluid dynamics”. RWDSL 2018

CFDlang and tool flow

```
source =
type matrix    : [mp np]          &
type tensorIN  : [np np np ne]    &
type tensorOUT : [mp mp mp me]    &
                                  &
var input A    : matrix           &
var input u    : tensorIN         &
var input output v  : tensorOUT   &
var input alpha : []              &
var input beta  : []              &
                                  &
v = alpha * (A # A # A # u .
    [[5 8] [3 7] [1 6]]) + beta * v
```

Fortran embedding



Linkable C code

```
for (unsigned i0 = 0; i0 < 1000; i0++) {
  double t6[18];
  for (unsigned i3 = 0; i3 < 3; i3++) {
    for (unsigned i2 = 0; i2 < 3; i2++) {
      for (unsigned i1 = 0; i1 < 2; i1++)
        t6[(i1 + 2*(i2 + 3*(i3)))] = 0.0;
        for (unsigned i4_contr = 0; i4_contr < 3; i4_contr++) {
          t6[(i1 + 2*(i2 + 3*(i3)))] += A[(i1 + 2*(i4_contr))]
              * u[(i2 + 3*(i3 + 3*(i4_contr + 3*(i0))))];
      }
    }
  }
}
double t7[12];
for (unsigned i7 = 0; i7 < 3; i7++) {
  for (unsigned i6 = 0; i6 < 2; i6++) {
    for (unsigned i5 = 0; i5 < 2; i5++) {
      t7[(i5 + 2*(i6 + 2*(i7)))] = 0.0;
      for (unsigned i8_contr = 0; i8_contr < 3; i8_contr++) {
        t7[(i5 + 2*(i6 + 2*(i7)))] += A[(i5 + 2*(i8_contr))]
            * t6[(i6 + 2*(i7 + 3*(i8_contr)))];
```

- DSL: Embedded in Fortran, including APIs for calling and optimizing the kernel
- Tensor intermediate representation (IR)
  - Representation of iterators and implicit loops
  - Commands for loop transformations (parallelization, fusion, interchange, …)
- Code generation
  - Just-in-time generation of C code (for autotuning)

Example: Interpolation operator

- Interpolation: $\mathbf{v}_e = (\mathbf{A} \otimes \mathbf{A} \otimes \mathbf{A})\, \mathbf{u}_e$

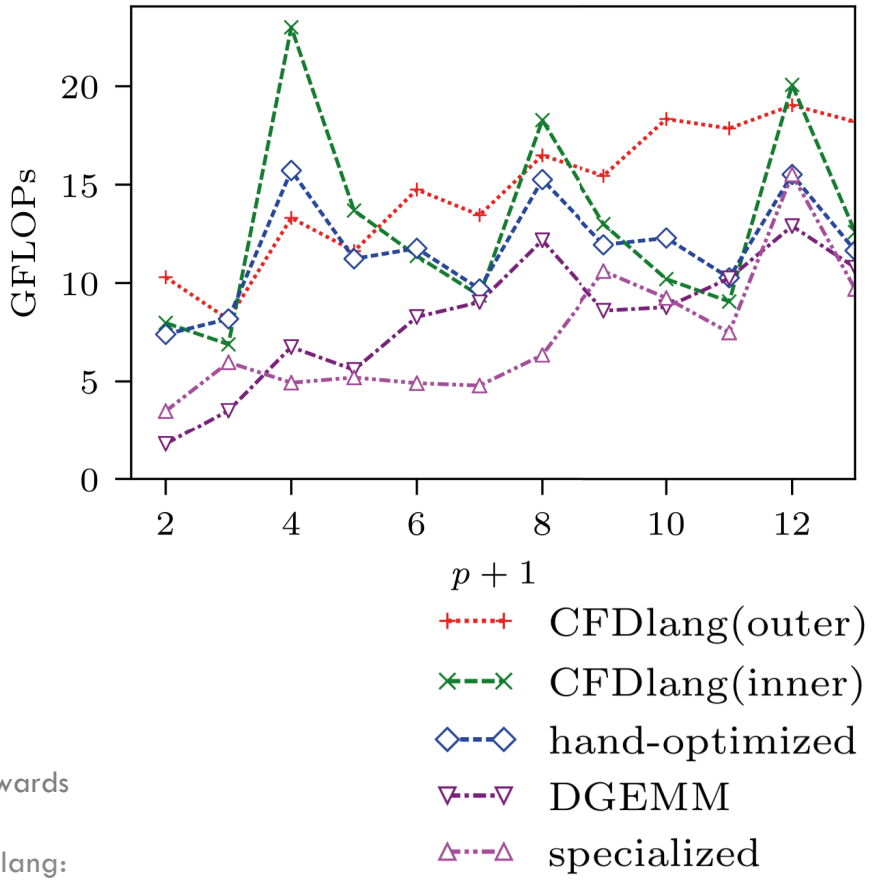
$$v_{ijk} = \sum_{l,m,n} A_{kn} \cdot A_{jm} \cdot A_{il} \cdot u_{lmn}$$

- Three alternative orders (besides naïve)

E1: $$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot (A_{jm} \cdot (A_{il} \cdot u_{lmn}))$$

E2: $$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot A_{jm}) \cdot (A_{il} \cdot u_{lmn})$$

E3: $$v_{ijk} = \sum_{l,m,n} (A_{kn} \cdot ((A_{jm} \cdot A_{il}) \cdot u_{lmn}))$$



A. Susungi, N. A. Rink, J. Castrillon, I. Huismann, A. Cohen, C. Tadonki, J. Stiller, J. Fröhlich, "Towards Compositional and Generative Tensor Optimizations" GPCE 17

N. A. Rink, I. Huismann, A. Susungi, J. Castrillon, J. Stiller, J. Fröhlich, and C. Tadonki. "CFDlang: High-level code generation for high-order methods in fluid dynamics". RWDSL 2018

TeML: Meta-programming for Tensor Optimizations

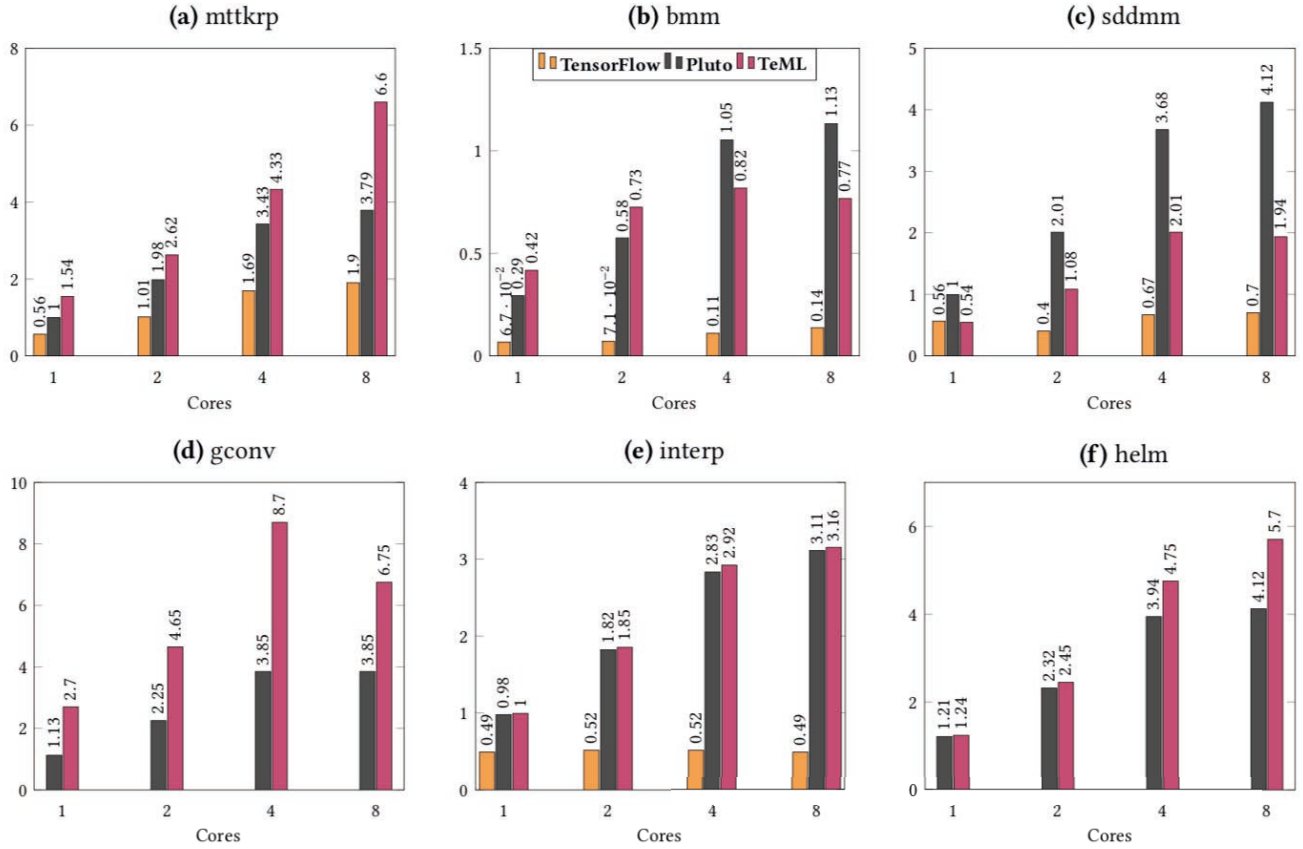
- Extend grammar and semantics with semantic-preserving, composable transformations
  - For optimization expert or for automated search

$$
\begin{aligned}
\langle Lexpression \rangle \quad ::= \quad & \texttt{build}\,(\langle id \rangle) \\
\mid \quad & \texttt{stripmine}\,(\langle id \rangle, \langle int \rangle, \langle int \rangle) \\
\mid \quad & \texttt{interchange}\,(\langle id \rangle, \langle int \rangle, \langle int \rangle) \\
\mid \quad & \texttt{fuse\_outer}\,(\langle id \rangle, \langle id \rangle, \langle int \rangle) \\
\mid \quad & \texttt{fuse\_inner}\,(\langle id \rangle, \langle int \rangle) \\
\mid \quad & \texttt{unroll}\,(\langle id \rangle, \langle int \rangle)
\end{aligned}
$$

A. Susungi, et al. " "Meta-programming for cross-domain tensor optimizations" GPCE'18, 79-92

TeML: Results

- Extra control allow for new optimization (vs pluto): changing shapes
- General tensor semantics allow covering more benchmarks than TensorFlow



A. Susungi, et al. " "Meta-programming for cross-domain tensor optimizations" GPCE'18, 79-92

Emerging topics in compilers

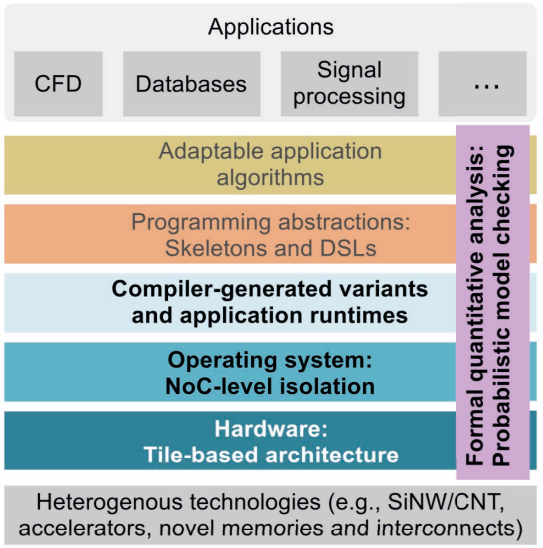
Cfaed and Haec

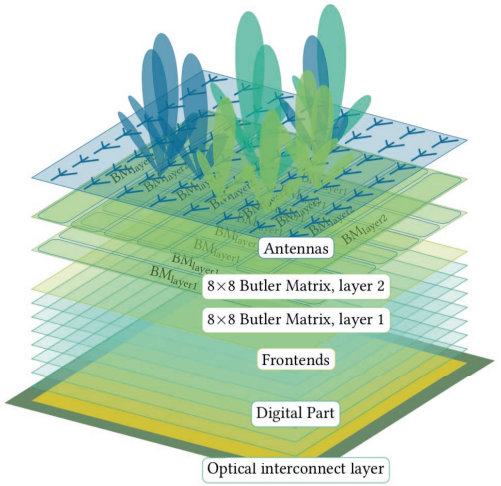
- Excellence Cluster Cfaed
  - New devices/paradigms: Silicon nanowires, carbon nanotubes, organics, bio-inspired computing, …
  - Software: Domain-specific languages, micro-kernels, formal analysis methods, …

Castrillon, J., Lieber, M., Klüppelholz, S., et al. "A Hardware/Software Stack for Heterogeneous Systems", IEEE Transactions on Multi-Scale Computing Systems, 2018, 4, 243-259.

- HAEC: Highly adaptive energy efficient computing
  - On board optical interconnects
  - Wireless chip-to-chip communication

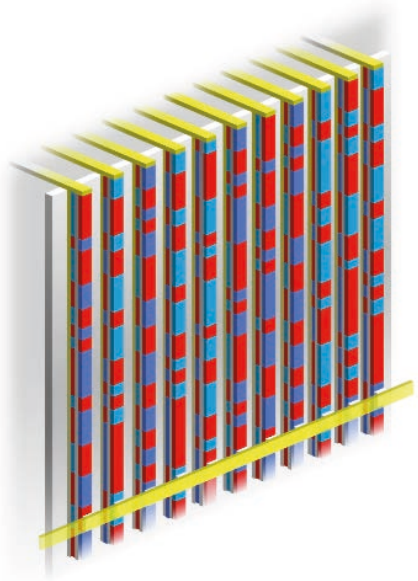
Fettweis, G., Dörpinghaus, M., Castrillon, et al. "Architecture and Advanced Electronics Pathways Towards Highly Adaptive Energy-Efficient Computing", Proceedings of the IEEE, 2019, 107, 204-231.





Emerging techs. example: Racetrack memories

- Racetrack memories: one of many future alternatives
- Predicted extreme density at low latency
  - 3D nano-wires with magnetic domains
  - One port shared for many bits
  - Domains move at high speeds (1000 ms^{-1})
- Sequential: Game changer for current HW/SW stack
  - Memory management
  - Integration with other memory architectures
  - Data layout and allocation



Parkin, Stuart, and See-Hun Yang. "Memory on the racetrack." Nature nanotechnology 10.3 (2015): 195.

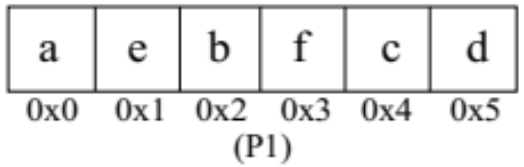
Sample problem: Variable allocation on the stack

- Order of allocation not a problem with random-access

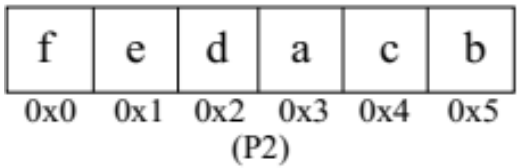
Variables (V): a b c d e f

Memory trace (S): b c a e f d a c e d a c a d e f

Placement 1



Placement 2



Shifts cost:

	b	c	a	e	f	d	a	c	e	d	a	c	a	d	e	f
P1 →	2	4	1	2	2	5	4	3	4	5	4	4	5	4	2	51
P2 →	1	1	2	1	2	1	1	3	1	1	1	1	1	1	1	19

Efficient data placement in RTM can reduce shifts by more than 200% !

"New" compiler optimization

- Model the accessing variables with a mapping given by β: V → N

$$C = \left(\sum_{i=0}^{|S|-2} \Delta(S_i, S_{i+1}) \right) \qquad \Delta(u,v) = |\beta(u) - \beta(v)| \quad \forall u, v \in \mathcal{V}$$

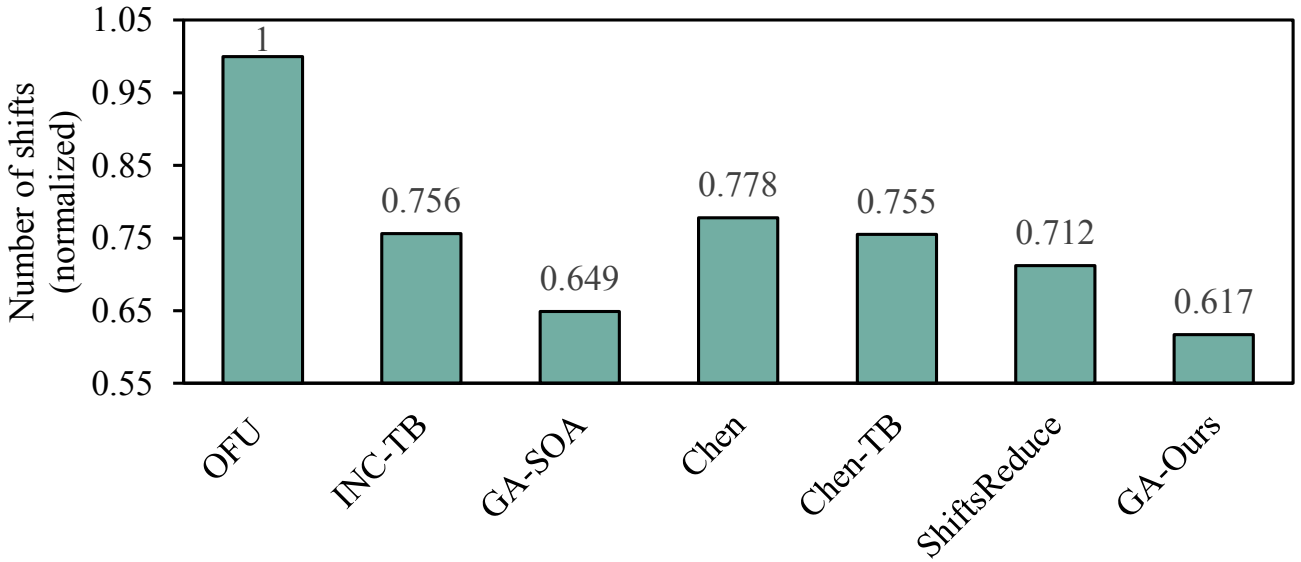
- Linearization: Integer Linear Programming (details ommitted, 8 constraints)

$$C = \min \left(\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-2} w_{v_i v_j} \cdot (p_{ij} + q_{ij}) \right)$$

Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart Parkin, Jeronimo Castrillon, "ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0" , In ArXiv arXiv:1903.03597, Mar 2019

Results

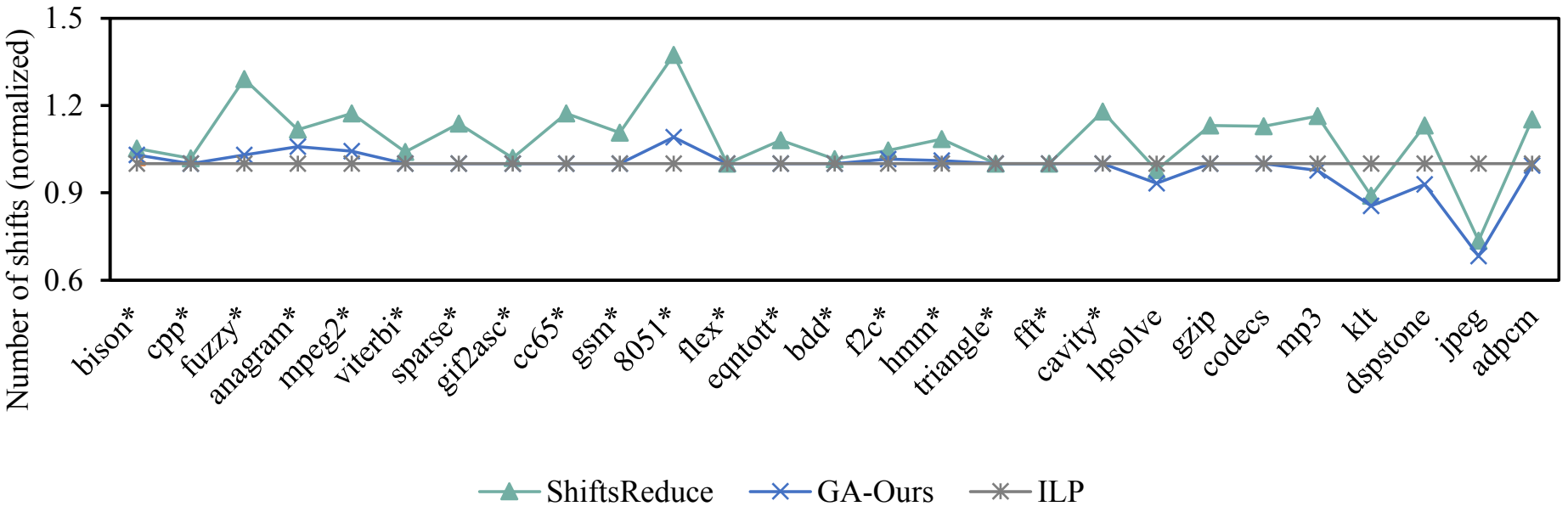
- Comparing previous heuristic with our heuristic, and when possible against ILP
  - Comparison of offset assignment heuristics



Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart Parkin, Jeronimo Castrillon, "ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0" , In ArXiv arXiv:1903.03597, Mar 2019

Results

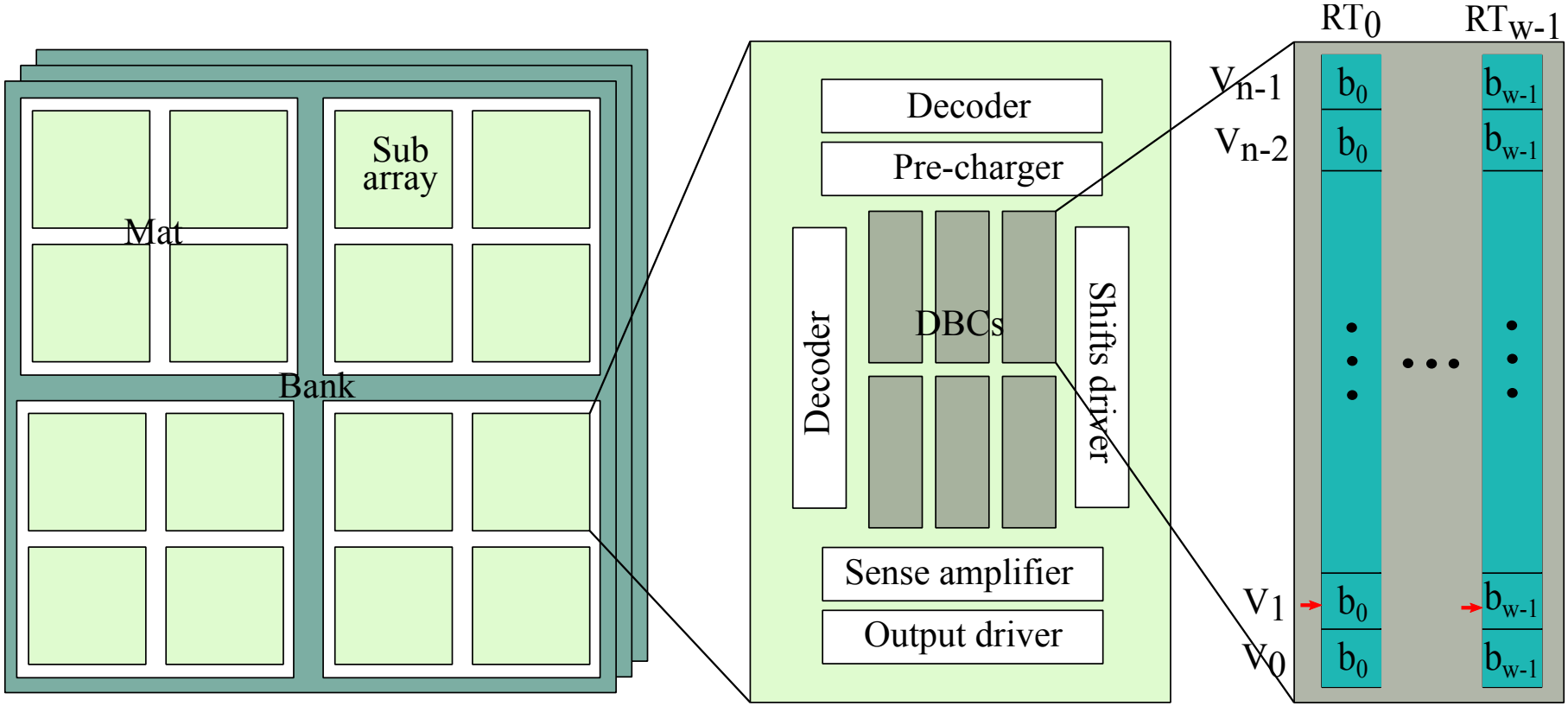
- Comparing previous heuristic with our heuristic, and when possible against ILP



Asif Ali Khan, Fazal Hameed, Robin Bläsing, Stuart Parkin, Jeronimo Castrillon, "ShiftsReduce: Minimizing Shifts in Racetrack Memory 4.0" , In ArXiv arXiv:1903.03597, Mar 2019

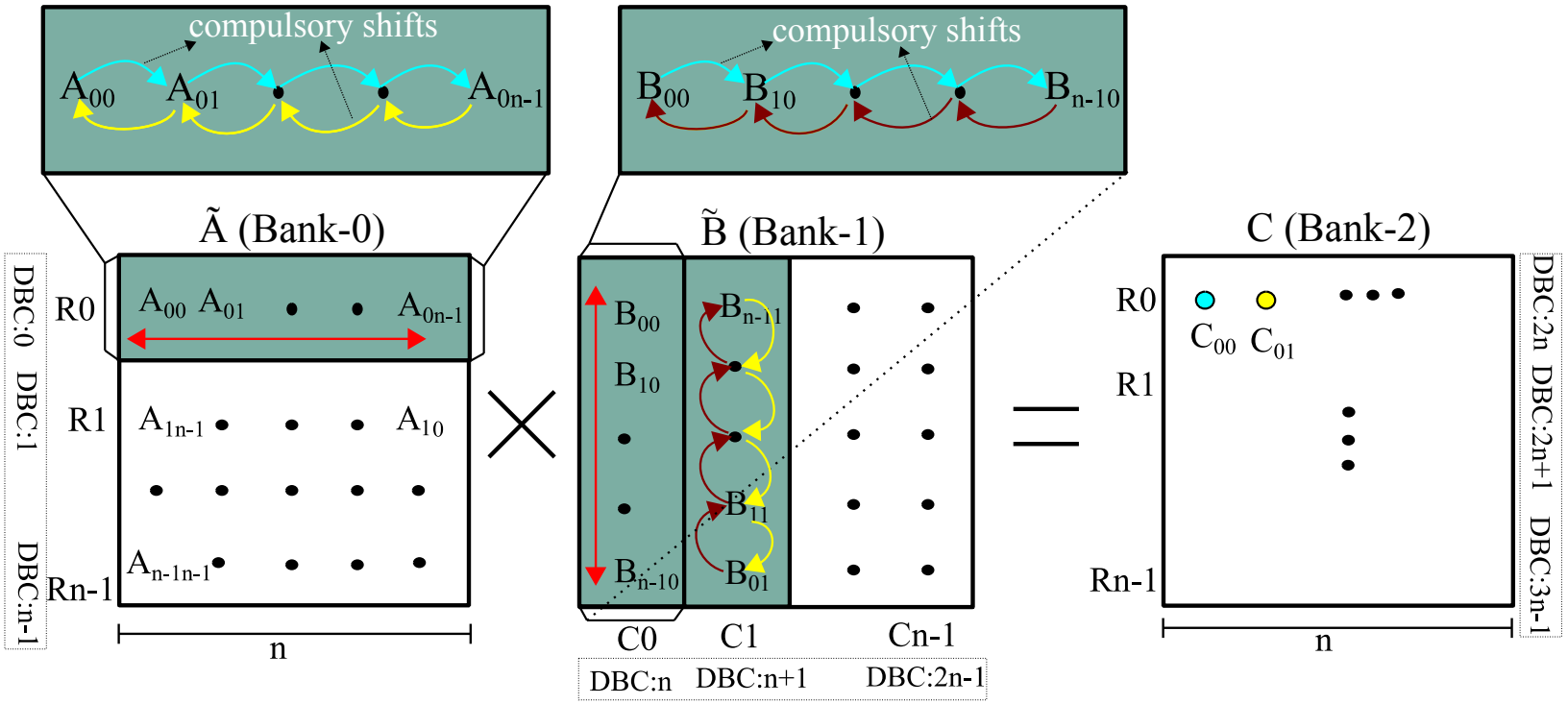
Data-layout optimization

- Basic architecture



Data-layout optimization: Tensor contraction

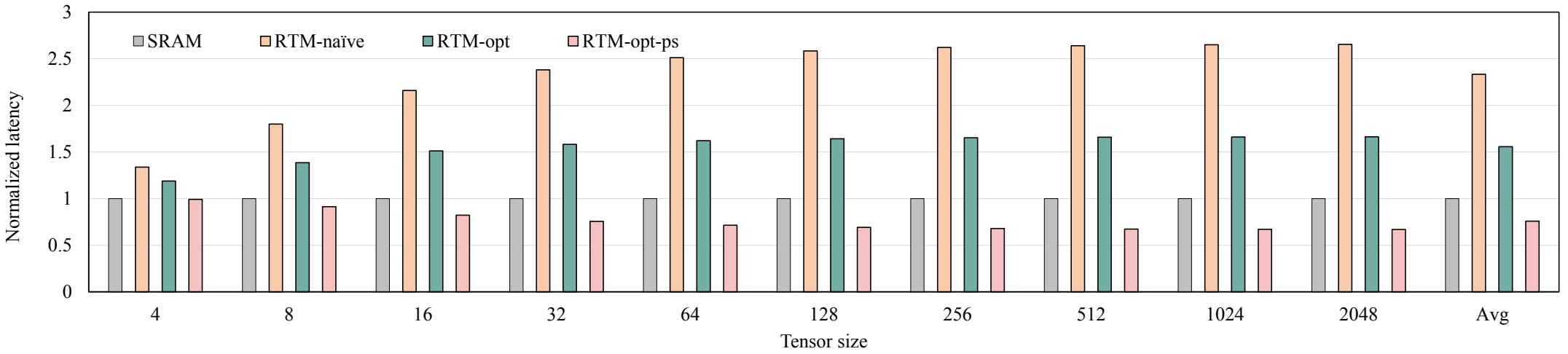
- Optimized layout



Khan, A. A., Rink, N. A., Hameed, F., Castrillon, J. "Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads", Proceedings of the 20th ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM, pp. 12pp, New York, NY, USA, Jun 2019

Results: Tensor contractions on racetracks

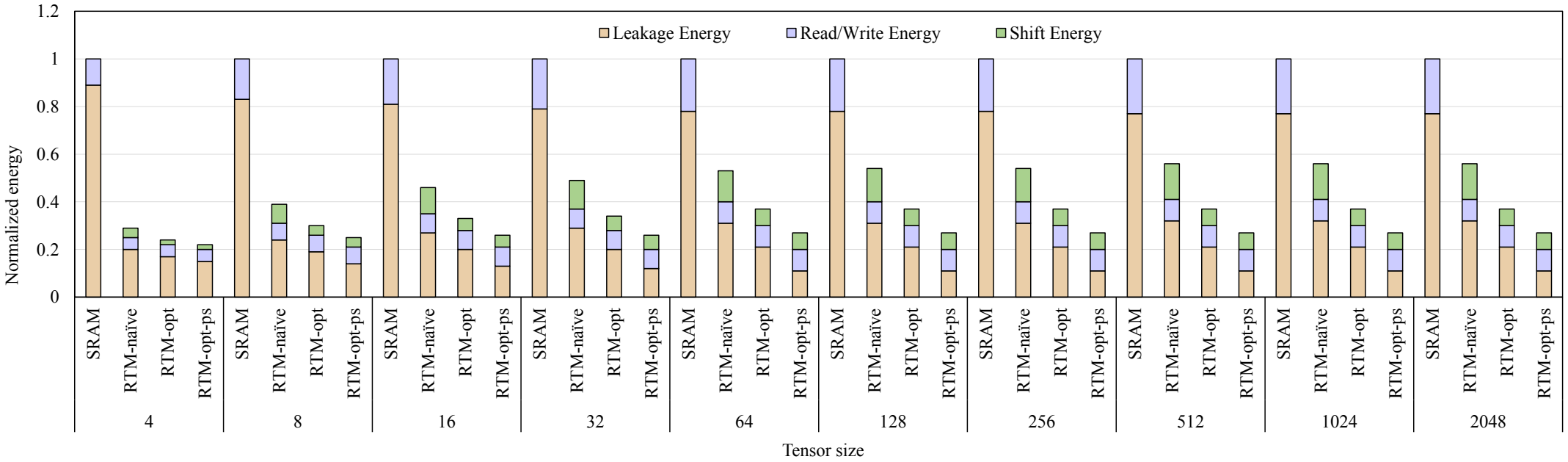
- Improved latency and energy consumption



Khan, A. A., Rink, N. A., Hameed, F., Castrillon, J. "Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads", Proceedings of the 20th ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM, pp. 12pp, New York, NY, USA, Jun 2019

Results: Tensor contractions on racetracks

- Improved latency and energy consumption



Khan, A. A., Rink, N. A., Hameed, F., Castrillon, J. "Optimizing Tensor Contractions for Embedded Devices with Racetrack Memory Scratch-Pads", Proceedings of the 20th ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM, pp. 12pp, New York, NY, USA, Jun 2019

Summary

In this presentation

- Basics topics on compilers
- Programming heterogeneous multi and many-cores
  - Efforts to extract parallelism
  - Cross-domain dataflow models
  - DSLs: domain-specific higher level of abstraction
    - Cleaner syntax and semantics: Semantic preserving optimizations
- Insight into new problem formulations for emerging computer architectures
- Outlook
  - Right balance: Compile vs. run-time
  - New goals: Energy efficiency & resilience

Thanks for the attention!
Questions?