MAPS-TCT:
MPSoC Application Parallelization and Architecture Exploration Framework

Tsuyoshi ISSHIKI

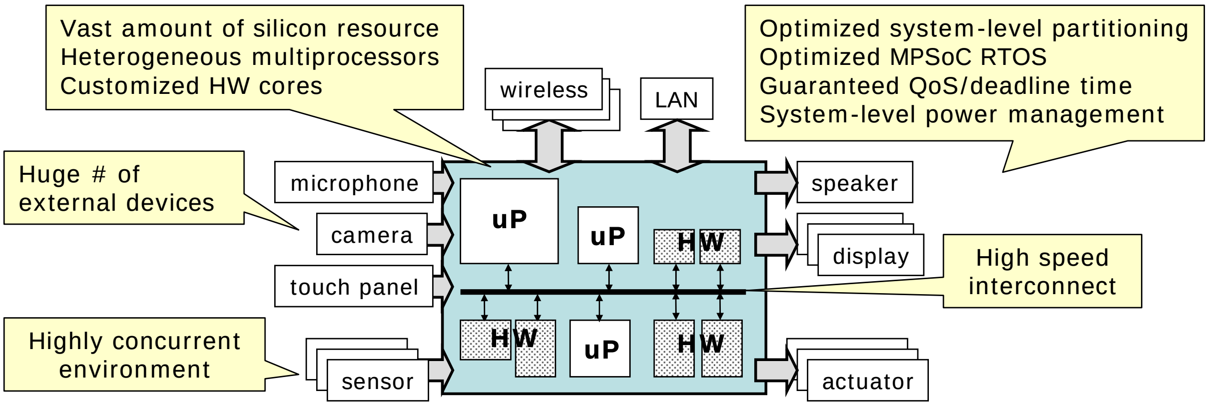
isshiki@vlsi.ss.titech.ac.jp
Dept. of Communications and Integrated Systems,
Tokyo Institute of Technology

June 26th, 2008

Outline

- Introduction: MPSoC design challenges
- MAPS-TCT Framework Overview
- Tightly-Coupled Thread (TCT) Model
  - TCT programming model and execution model
  - TCT compiler and tools
  - TCT-MPSoC Hardware Platform
- MAPS: MPSoC Application Programming Studio
  - Program analysis
  - Partitioning
- Summary and Ongoing Developments

MPSoC Design Challenges



Vast amount of silicon resource
Heterogeneous multiprocessors
Customized HW cores

Optimized system-level partitioning
Optimized MPSoC RTOS
Guaranteed QoS/deadline time
System-level power management

Huge # of external devices

Highly concurrent environment

High speed interconnect

- Huge design complexity (SW/HW)
- Highly-parallel heterogeneous system architecture
- Complex system environment (#external devices, concurrency)
- Fully optimized at system-level
  - SW: algorithms, parallelization, coding
  - HW: CPU-cores, dedicated hardware IPs, inteconnect

Application Design Issues

- Application design
  - Algorithm: definition of system functionality
  - Parallelization (CPUs, HW blocks)
    - Concurrency extraction (task partitioning)
    - Communication/synchronization insertion
- Existing approaches
  - Algorithm designs on concurrent execution model
    - *"Model of Computation"* : Kahn Process Network, Dataflow Process Network, Synchronous Dataflow Graph, etc.
    - Parallel programming languages and APIs

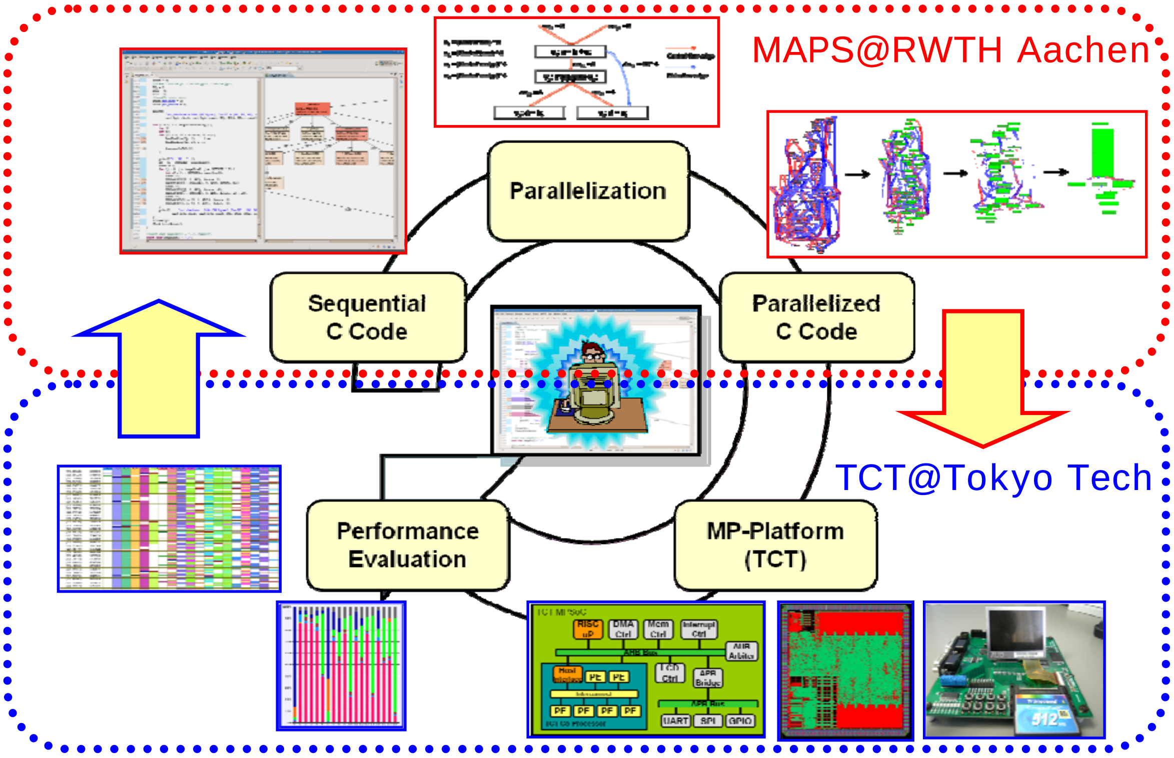
    →time-consuming, error-prone, hard to debug
  - Parallelization compilers
    - Focused mainly on scientific applications (HPC)
    - Hard to optimize for heterogeneous MPSoCs

MAPS-TCT Framework Overview

- Algorithm design on C programs
  - – Used by almost everyone
  - – Rich tool environment
  - – Vast amount of legacy codes and reference codes
  - → Algorithm debugging and tuning on C: most efficient
- Tool support for concurrency extraction (MAPS@RWTH Aachen)
  - – Powerful analysis and code partitioning engines
  - – Fully driven by programmer's intervention
  - – Rich feedback to guide programmer's decision
  - → Allows efficient design space exploration for optimal system modeling
- Parallel execution code generator (TCT@Tokyo Tech)
  - – Input: "threaded C" (from MAPS or manual editing)
  - – Automatic communication insertion: message-passing instructions
  - – Allows parallelism on any granularity (statements, loops, functions)
  - – Guarantees identical behavior with original sequential C
  - → Frees programmer from dealing with communication details
  - → Wide variety of parallelisms: task-level, functional pipeline, fine-grain

---

MAPS-TCT Framework



MAPS@RWTH Aachen

Parallelization

Sequential C Code

Parallelized C Code

TCT@Tokyo Tech

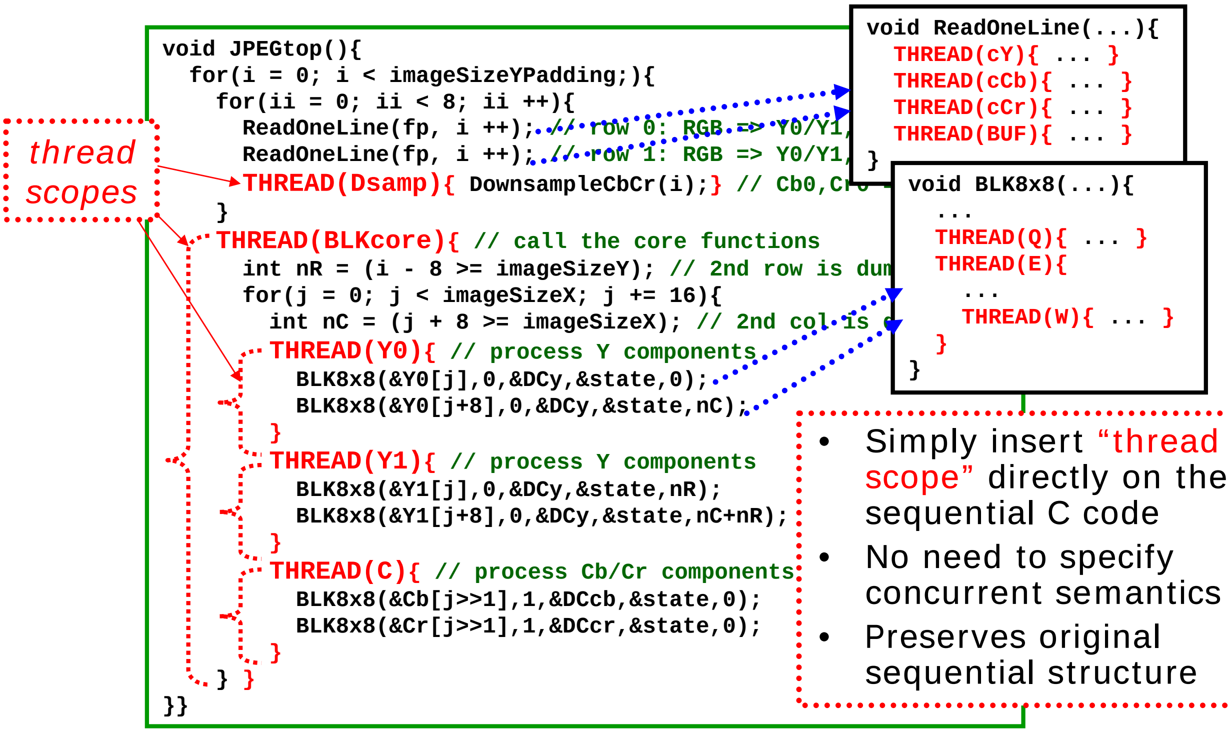
Performance Evaluation

MP-Platform (TCT)

Tightly-Coupled Thread (TCT) Model

- **TCT model** is a new framework which generates a concurrent execution model of *"tightly-coupled threads"* for functional blocks in MPSoCs.
  - *TCT programming model* : seamless transition from *sequential C codes*
  - *TCT concurrent execution model* : *functional pipelining, task parallelisms*
  - *TCT compiler* : automatic insertion of communication and synchronization instructions for *message passing*
  - *TCT MPSoC Platform* : execution platform for TCT model
    - Processing elements with dedicated communication module
    - Full crossbar interconnect for high bandwidth communication
    - Verified on actual silicon (0.18um process)

TCT Programming Model



```
void JPEGtop(){
  for(i = 0; i < imageSizeYPadding;){
    for(ii = 0; ii < 8; ii ++){
      ReadOneLine(fp, i ++); // row 0: RGB => Y0/Y1,
      ReadOneLine(fp, i ++); // row 1: RGB => Y0/Y1,
      THREAD(Dsamp){ DownsampleCbCr(i);} // Cb0,Cr
    }
    THREAD(BLKcore){ // call the core functions
      int nR = (i - 8 >= imageSizeY); // 2nd row is dum
      for(j = 0; j < imageSizeX; j += 16){
        int nC = (j + 8 >= imageSizeX); // 2nd col is
        THREAD(Y0){ // process Y components
          BLK8x8(&Y0[j],0,&DCy,&state,0);
          BLK8x8(&Y0[j+8],0,&DCy,&state,nC);
        }
        THREAD(Y1){ // process Y components
          BLK8x8(&Y1[j],0,&DCy,&state,nR);
          BLK8x8(&Y1[j+8],0,&DCy,&state,nC+nR);
        }
        THREAD(C){ // process Cb/Cr components
          BLK8x8(&Cb[j>>1],1,&DCcb,&state,0);
          BLK8x8(&Cr[j>>1],1,&DCcr,&state,0);
        }
    } }
}}
```

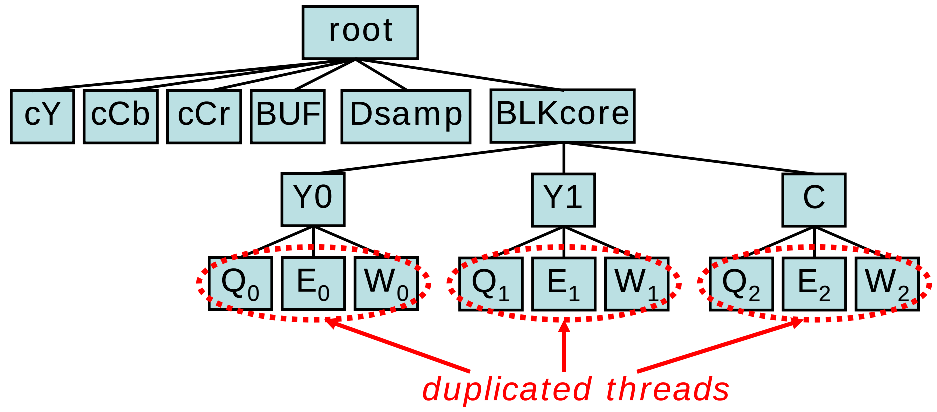
```
void ReadOneLine(...){
  THREAD(cY){ ... }
  THREAD(cCb){ ... }
  THREAD(cCr){ ... }
  THREAD(BUF){ ... }
}
```

```
void BLK8x8(...){
  ...
  THREAD(Q){ ... }
  THREAD(E){
    ...
    THREAD(W){ ... }
  }
}
```

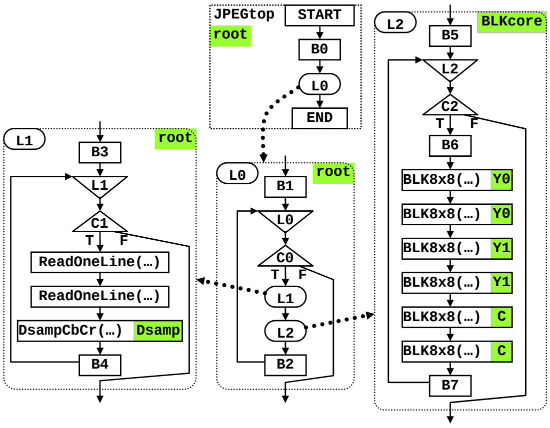
- Simply insert "thread scope" directly on the sequential C code
- No need to specify concurrent semantics
- Preserves original sequential structure

Application Slicing Structure

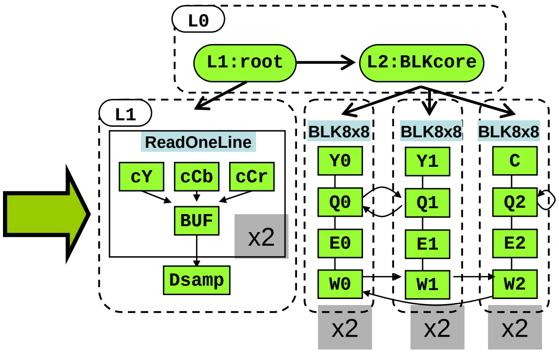
- Global thread slicing tree
  - Thread nesting structure of the entire application
  - Thread duplication through function calls from threads



TCT Concurrent Execution Model



Control flow graph

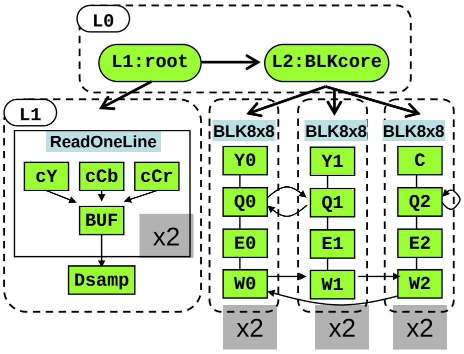


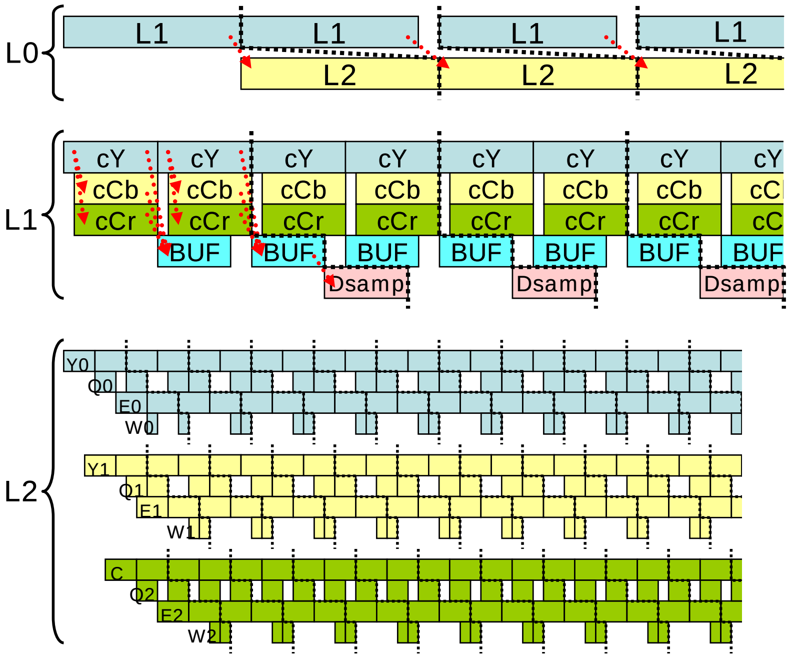
Functional pipelines and task parallelism exploited in the TCT model.

- *Hierarchical pipeline structure*
  - *Layers of pipeline structures operating in parallel*

  → Combination of *functional pipelining* and *task parallelism* with complex data flow

Hierarchical Pipelining



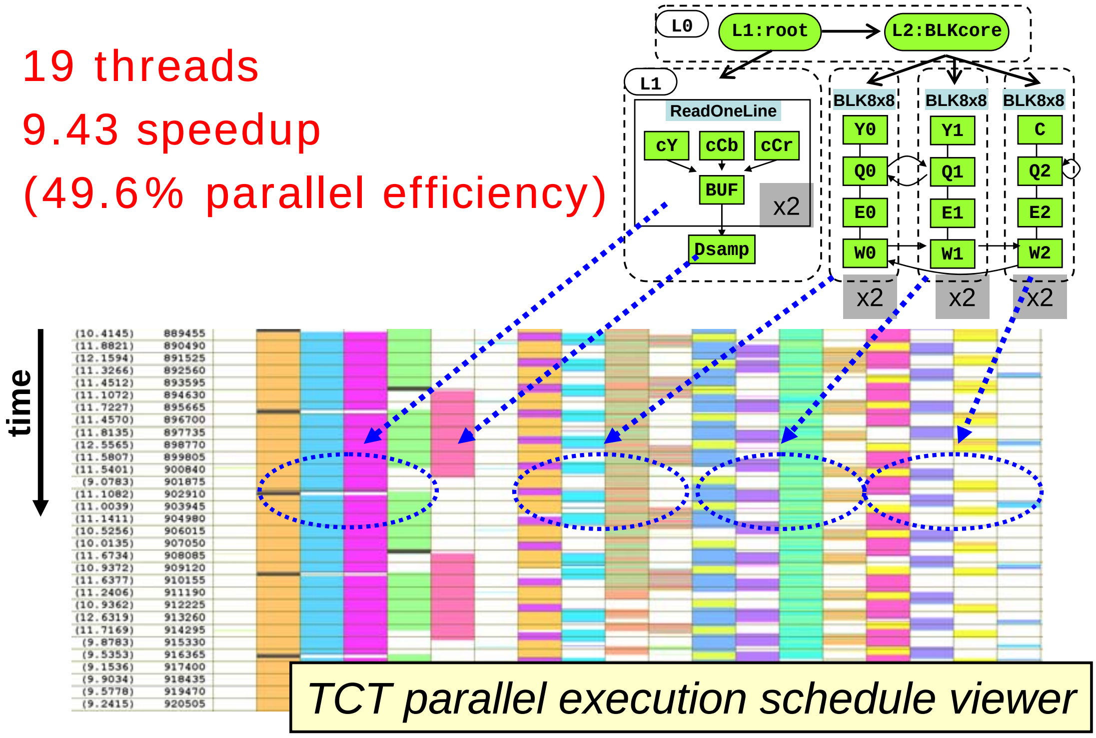


Layers of pipelines with:

- Different throughputs (may be variable)
- Different iteration count (may be variable)
- Data dependences between pipelines

Hierarchical Pipelining in Action …

- 19 threads
- 9.43 speedup (49.6% parallel efficiency)



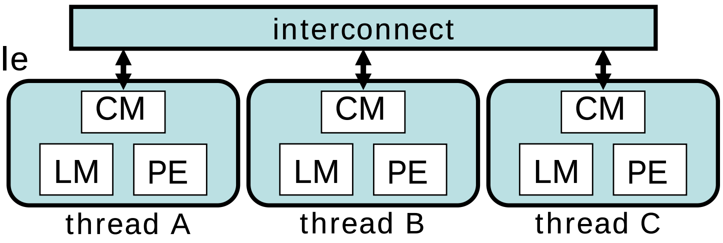
(10.4145)	889455
(11.8821)	890490
(12.1594)	891525
(11.3266)	892560
(11.4512)	893595
(11.1072)	894630
(11.7227)	895665
(11.4570)	896700
(11.8135)	897735
(12.5565)	898770
(11.5807)	899805
(11.5401)	900840
(9.0783)	901875
(11.1082)	902910
(11.0039)	903945
(11.1411)	904980
(10.5256)	906015
(10.0135)	907050
(11.6734)	908085
(10.9372)	909120
(11.6377)	910155
(11.2406)	911190
(10.9362)	912225
(12.6319)	913260
(11.7169)	914295
(9.8783)	915330
(9.5353)	916365
(9.1536)	917400
(9.9034)	918435
(9.5778)	919470
(9.2415)	920505

TCT parallel execution schedule viewer

TCT Communication Model

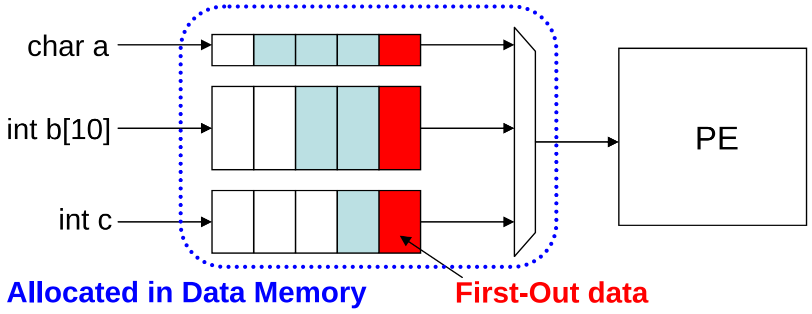
- Thread allocation: statically allocated to each processor
  - Currently assumes: 1 thread per 1 PE
- Distributed memory model: no remote memory access
- Thread communication: message passing via buffered channel
- Fully distributed control: no global scheduler and dispatchers
- Communication instructions:
  - CT (control token): activation of child thread
  - DT (data transfer): send modified data to other threads
  - DS (data synchronization): check readiness of received data

CM : communication module
LM : local memory
PE : processor element



Data Buffering Model

- Buffer structure
  - Each *data entity* managed in separate (logical) FIFO
    - Arrays and data structures handled as single data entity
    - Burst transfer on arrays and data structures
  - *First-Out* data entities accessible from the processor
    - FIFO space allocated inside local data memory
    - Fully configurable : # of data entities, data sizes

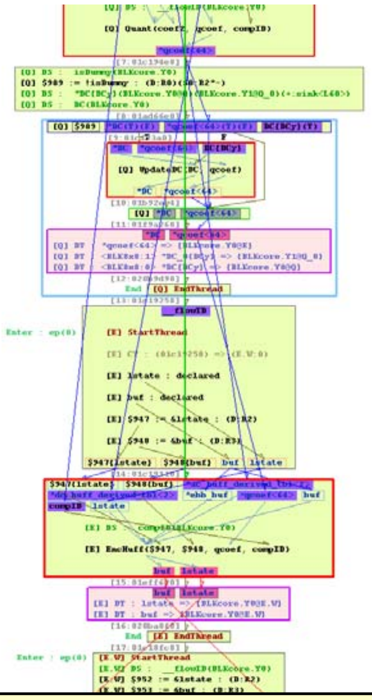


TCT Compiler

- C front-end + “thread-scope” parsing
- Interprocedural data dependence analysis
  - Interprocedural Dependence Flow Graph (IDFG)
    - Extension of static single-assignment (SSA) form which integrates *data-flow* and *control-flow* representations
    - Captures all function call side effects through globals and pointer dereferences
    - Flow-insensitive context-sensitive pointer analysis
- Communication code insertion
  - Extraction of interprocedural dependences on thread boundaries (possibly across layers of function calls)
  - Insertion of communication instructions (CT, DT, DS)
  - Buffer management codes

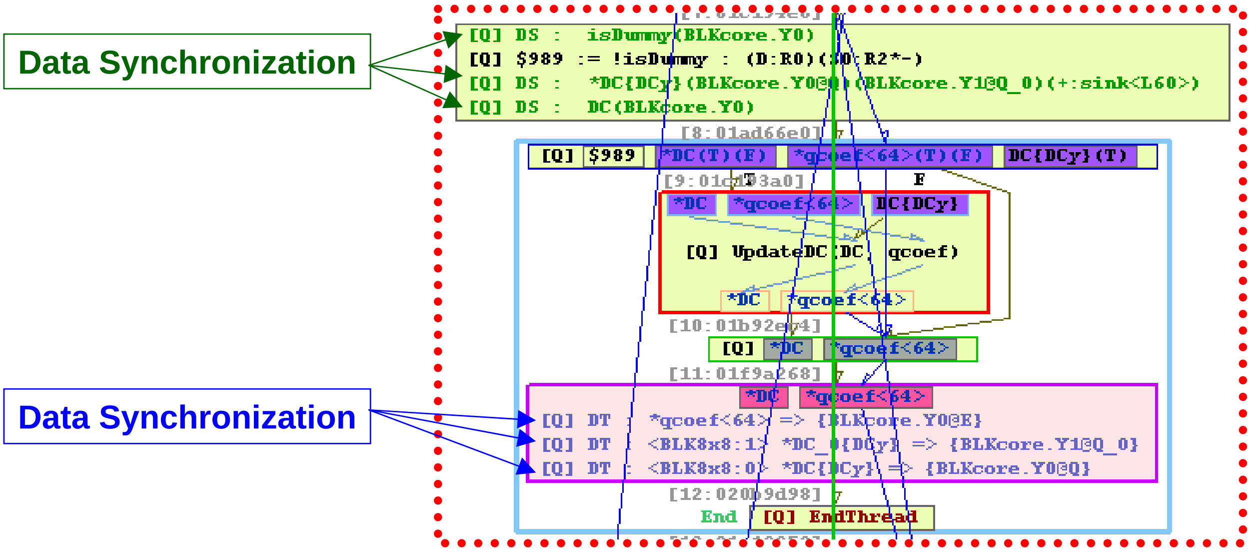
TCT Tools

- TCT Compiler
  - C parser + dependence analyzer + communication generator
  - Output:
    - parallel object codes : *TCT processor*
    - parallel C codes with communication API calls → currently translates to MPI
- TCT Simulators
  - “3-address code” IR simulator
    - Verify functional/comm. behavior
  - Instruction-set simulator
    - Incl. cycle-accurate comm. simulator
  - Trace simulator
    - Parameterized MPSoC simulator for architecture exploration
- Application visualizer tools
  - Call graph, program graph, dependence flow graph, etc.
  - Parallel execution schedule viewer



Dependence flow graph

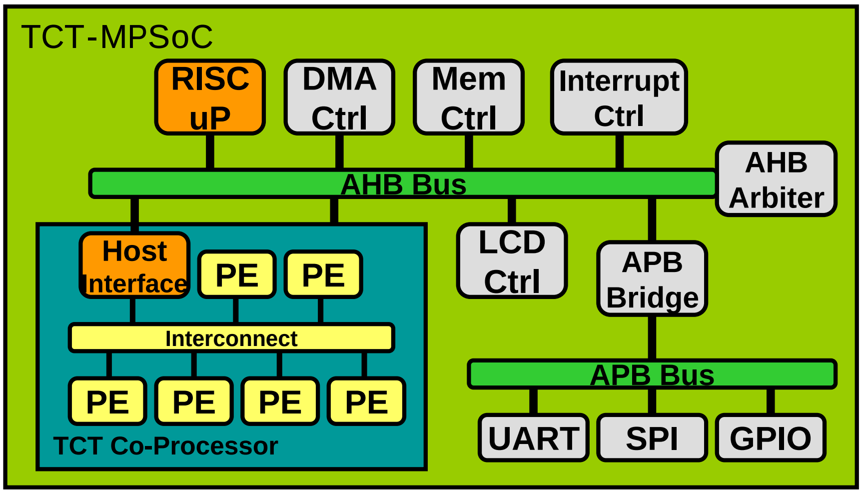
Inserted TCT Communication Codes



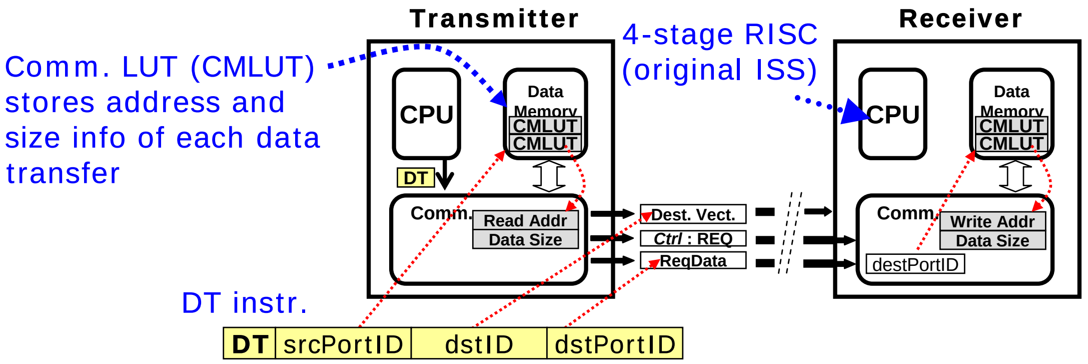
```
[Q] DS :  isDummy(BLKcore.Y0)
[Q] $989 := !isDummy : (D:R0)(S0:R2*-)
[Q] DS :  *DC{DCy}(BLKcore.Y0@Q)(BLKcore.Y1@Q_0)(+:sink<L60>)
[Q] DS :  DC(BLKcore.Y0)
[Q] $989  *DC(T)(F)  *gcoef<64>(T)(F)  DC{DCy}(T)
*DC  *gcoef<64>  DC{DCy}
[Q] UpdateDC(DC, qcoef)
*DC  *gcoef<64>
[Q] *DC  *gcoef<64>
*DC  *gcoef<64>
[Q] DT : *gcoef<64> => {BLKcore.Y0@E}
[Q] DT : <BLK8x8:1> *DC_0{DCy} => {BLKcore.Y1@Q_0}
[Q] DT : <BLK8x8:0> *DC{DCy} => {BLKcore.Y0@Q}
[Q] EndThread
```

TCT-MPSoC Prototype Chip

- TCT Coprocessor (TCoP): 6-PE array @ 100 MHz
  - Full crossbar interconnect
  - Dedicated comm. module in each PE
- Host RISC core: @ 200 MHz
  - Can be configured as the 7th PE on the PE array interconnect



TCT Communication Protocol



Comm. LUT (CMLUT) stores address and size info of each data transfer

4-stage RISC (original ISS)

DT instr.

1. Decode DT Instr.	
2. Send REQ	**1. Detect REQ**
3. Setup DMEM READ addr	**2. Check BUF status**
... waiting for ACK ..	**3. If (BUF-FULL) send NACK & exit**
	4. Setup DMEM WRITE addr
4. Detect ACK	**5. Send ACK, wait for TRANS**
5. Send TRANS + data	**6. Detect TRANS, store data**

Setup

Transmit

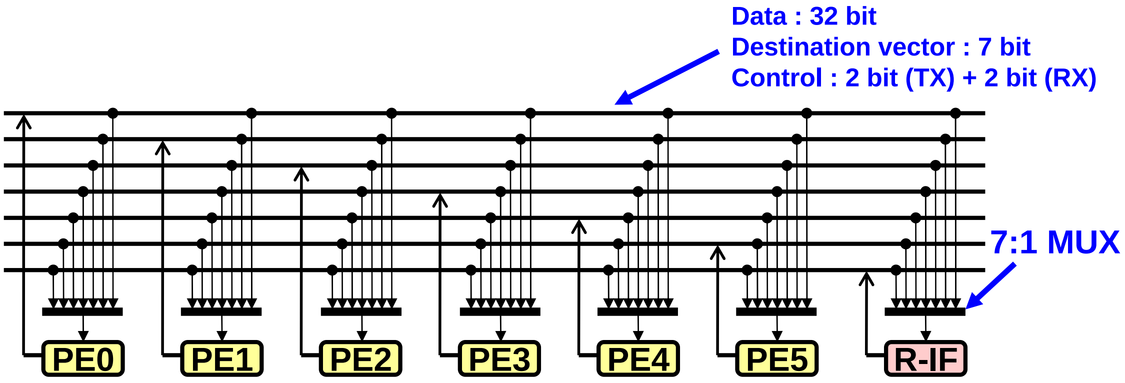
Response

Setup phase:
2 – 6 cycles
(4 cycles typ.)

4 bytes/cycle

TCoP Interconnect Architecture

- Full cross bar interconnect network
- Autonomous decentralized arbitration
- Fast and area efficient (2 ns delay, 1K gates/PE)
- Priority bit for simultaneous requests

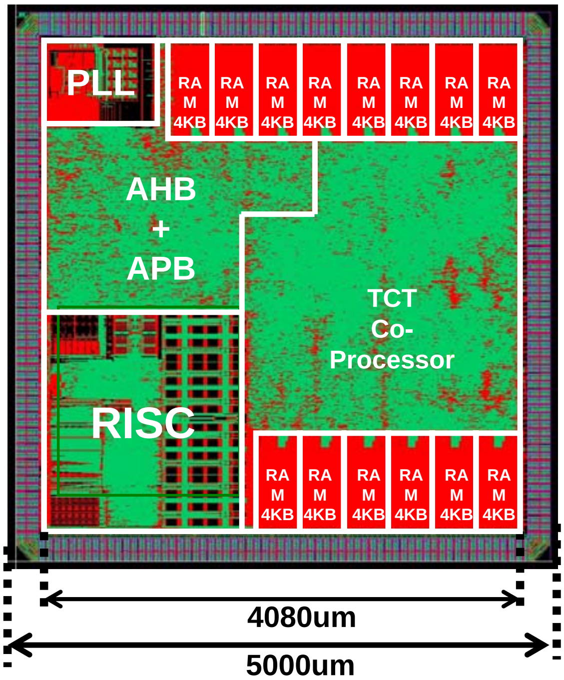


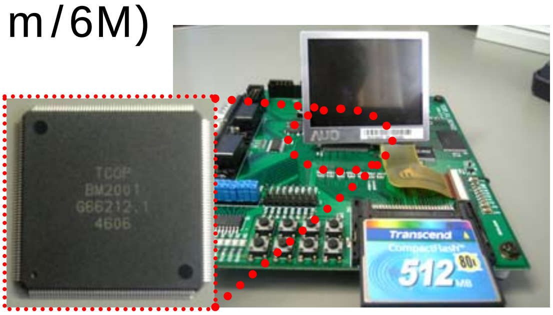
Data : 32 bit
Destination vector : 7 bit
Control : 2 bit (TX) + 2 bit (RX)

7:1 MUX

TCT-MPSoC Chip Implementation

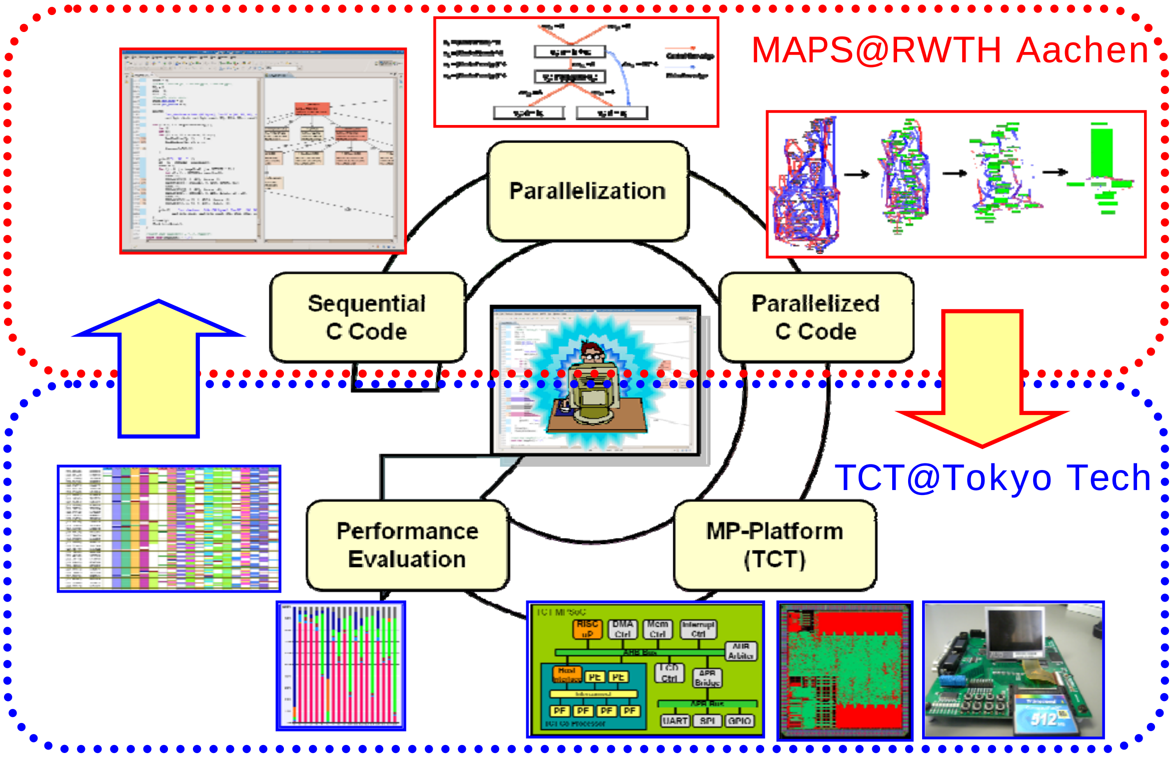
(TSMC 0.18um/6M)





Module	Area (um²)	Gate count (est.)
PE (incl. comm. module)	490,195	37,707
Comm. module	164,048	12,619
Interconnect	90,954	6,996
TCoP (Total)	3,342,090	257,084
SRAM 14x4KB (56KB)	3,934,900	302,685

MAPS-TCT Framework



MAPS@RWTH Aachen

Parallelization

Sequential C Code

Parallelized C Code

TCT@Tokyo Tech

Performance Evaluation

MP-Platform (TCT)

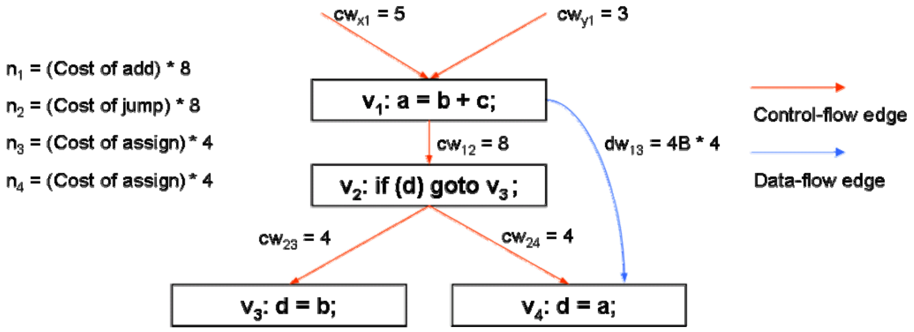
MAPS: MPSoC Application Programming Studio

- A practical MPSoC software development tool suite
  - Sequential C (input) → "threaded C" (output)
  - Powerful analysis tools for providing rich feedback to the programmers
    - Static dependence analysis
    - Dynamic profiling
  - Powerful clustering method for extracting coarse-grain parallelism
    - Weighted Statement Control Data Flow Graph (WSCDFG): annotates dynamic profiling information on CDFG
    - Coupled Block (CB): subgraph of WSCDFG that is schedulable and tightly coupled by data dependence
    - Constrained Agglomerative Hierachical Clustering (CAHC): iterative clustering for building coarser graphs

---

Weighted Statement Control Data Flow Graph (WSCDFG)

- Definition: WSCDFG is a directed graph defined by G = (V,CE,DE,CW,DW,N) :
  - V: IR statement nodes
  - CE: set of control flow edges
  - DE: set of data flow edges
  - CW: weights (count) of control edges
  - DW: weights (amount of data, e.g. bytes) of data edges
  - N: weight of IR statement nodes (execution cost)



Coupled Block (CB)

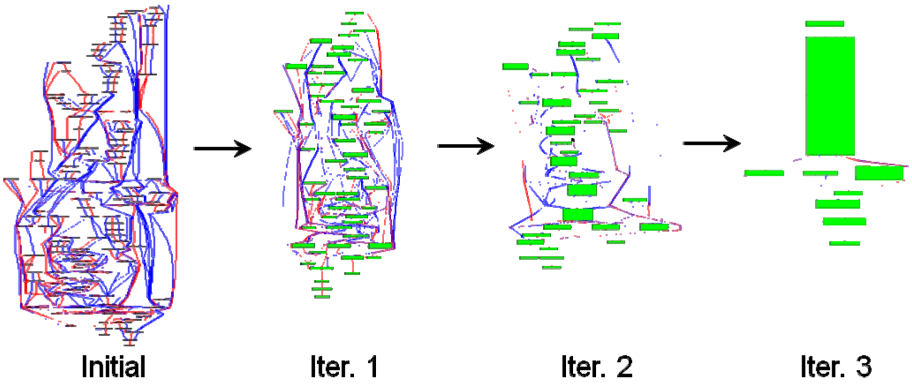
- CBs are sub-graphs in a WSCDFG which fulfills:
  - Schedulability: single-entry single-exit (SESE)
  - Tightly coupled by data-dependence: defined by cost function with tunable parameters

  → A flexible granularity concept driven by cost function as opposed to fixed granularity (i.e. IR-statements, BBs, functions)
- Optimal generation of CB

  →Clustering heuristic for CB generation: CAHC

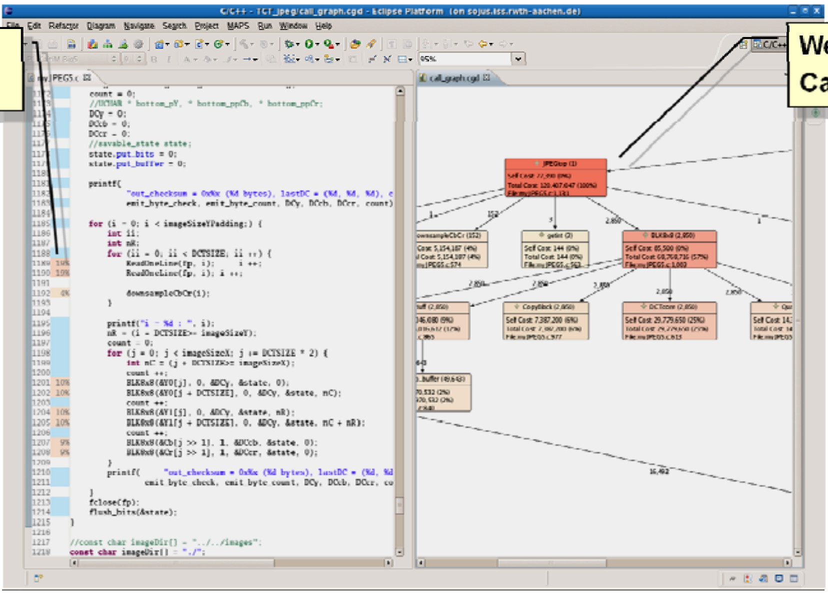
Constrained Agglomerative Hierarchical Clustering (CAHC)

- Based on density-based data clustering algorithm (DBSCAN) :
  - Constrained: comply strictly to CB definitions
  - Hierarchical: several clustering levels with different granularities
  - Agglomerative: build coarser graphs iteratively



JPEG Encoder Case Study (1)

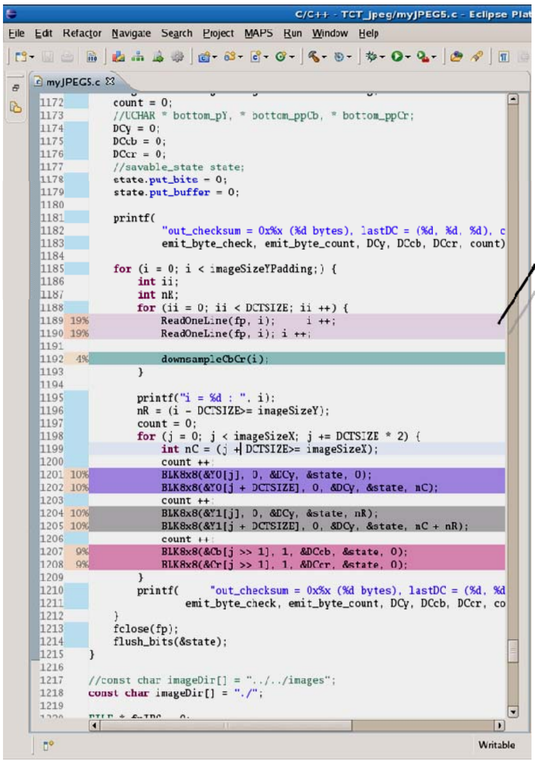
- Analysis result:
  - Functions chosen for task generation: JPEGtop (99%), BLK8x8(57%), ReadOneLine(38%)



Source Level Profile

Weighted Call Graph

JPEG Encoder Case Study (2)



```
        count = 0;
        //UCHAR * bottom_pY, * bottom_ppCb, * bottom_ppCr;
        DCy = 0;
        DCcb = 0;
        DCcr = 0;
        //savable_state state;
        state.put_bits = 0;
        state.put_buffer = 0;

        printf(
                "out_checksum = 0x%x (%d bytes), lastDC = (%d, %d, %d), c
                emit_byte_check, emit_byte_count, DCy, DCcb, DCcr, count)

        for (i = 0; i < imageSizeYPadding;) {
            int ii;
            int nR;
            for (ii = 0; ii < DCTSIZE; ii ++) {
19%             ReadOneLine(fp, i);     i ++;
19%             ReadOneLine(fp, i); i ++;

 4%             downsampleCbCr(i);
            }

            printf("i = %d : ", i);
            nR = (i - DCTSIZE>= imageSizeY);
            count = 0;
            for (j = 0; j < imageSizeX; j += DCTSIZE * 2) {
                int nC = (j + DCTSIZE>= imageSizeX);
                count ++;
10%             BLK8x8(&Y0[j], 0, &DCy, &state, 0);
10%             BLK8x8(&Y0[j + DCTSIZE], 0, &DCy, &state, nC);
                count ++;
10%             BLK8x8(&Y1[j], 0, &DCy, &state, nR);
10%             BLK8x8(&Y1[j + DCTSIZE], 0, &DCy, &state, nC + nR);
                count ++;
 9%             BLK8x8(&Cb[j >> 1], 1, &DCcb, &state, 0);
 9%             BLK8x8(&Cr[j >> 1], 1, &DCcr, &state, 0);
            }
            printf(     "out_checksum = 0x%x (%d bytes), lastDC = (%d, %d
                    emit_byte_check, emit_byte_count, DCy, DCcb, DCcr, co
        }
        fclose(fp);
        flush_bits(&state);
    }

//const char imageDir[] = "../../images";
const char imageDir[] = "./";
```

Task Annotation

- Partitioning result:

Function	No. Iterations	No. Tasks
JPEGtop	1	5
BLK8x8	3	2
ReadOneLine	2	4

JPEG Encoder Case Study (3)

- Speedup & efficiency

Step	Speedup	No. of PEs	Parallel Efficiency
1	3.61x	16	22.58%
2	5.48x	17	32.3%
3	5.48x	16	34.3%
manual	9.43x	19	49.6%

- # tasks in each step

Function	No. Tasks		
	Step 1	Step 2	Step 3
JPEGtop	5	6	6
ReadOneLine	4	4	3
BLK8x8	2	2	2

Summary

- MAPS-TCT Framework
  - Collaboration between RWTH Aachen (ISS) and Tokyo Tech.
  - MPSoC software development framework
- MAPS: MPSoC Application Programming Studio
  - Input: sequential C → output: “threaded C”
  - Analysis tools (static analysis, dynamic profiling)
  - Clustering tool for extracting coarse-grain parallelism
- Tightly-Coupled Thread (TCT) Model
  - Input: “threaded C”
  - Automatic communication insertion
  - Allows parallelism on any granularity
  - Guarantees identical behavior with original sequential C

Ongoing Developments

- MAPS
  - Improvements on partitioning algorithm
  - Heterogeneous platform support
  - Spatial/temporal mapping exploration
  - Multi-application input model: various real-time characteristics, potential concurrencies among applications
- TCT
  - TCT-MPSoC Virtual Platform for heterogenous architecture exploration
  - HW/SW synthesis: behav. synthesis on some threads for dedicated HW generation
  - Multi-tasking: multiple threads per processor
  - Extension of TCT comm. protocol for shared memory support

---

Contacts

- MAPS@RWTH Aachen (ISS)
  - Rainer Leupers (leupers@iss.rwth-aachen.de)
- TCT@Tokyo Institute of Technology
  - Tsuyoshi Isshiki (isshiki@vlsi.ss.titech.ac.jp)