VLSI System Design Part VI : Advanced Topics Oct.2006 - Feb.2007 "C-based Design Framework for Multiprocessor-SoC Synthesis

Lecturer : Tsuyoshi Isshiki

Dept. Communications and Integrated Systems,

Tokyo Institute of Technology

isshiki@vlsi.ss.titech.ac.jp

http://www.vlsi.ss.titech.ac.jp/~isshiki/VLSISystemDesign/top.html

Outline

- Background: Multiprocessor SoC designs
- Tightly-Coupled Thread (TCT) model
 - TCT programming model : a seamless design conversion from a sequential program (in C) to a concurrent execution model
 - TCT concurrent execution model
 - TCT compilation flow
 - TCT verification tools
- Current research status and future works

Multiprocessor System-on-Chip (1)

- Multiprocessor System-on-Chips (MPSoCs) are quickly becoming the next trend in VLSI technology
 - Programmable functional blocks: short design time, easy verification, design changes after chip fab and during product life cycle.
 - Applications: *mobile phones, printers, network routers,* etc.
- Enabling technologies in MPSoC design
 - System-level modeling (System-C, SpecC)
 - HW/SW interface codesign
 - Configurable processors
 - Platform-based designs (CPU+busses+IPs)

Multiprocessor System-on-Chip (2)

- Challenges in MPSoC designs
 - MPSoC design methodology: not yet established
 - Heterogeneous processors, busses, memories
 - → parallelizing compilers for homogeneous multiprocessors cannot be applied to MPSoC designs
 - System partitioning / concurrent behavior modeling:
 - System behavior modeling often start from references in C
 - System partitioning and concurrent behavior modeling : can be facilitated by SystemC or SpecC
 - Crucial early design phase : large impact on the final SoC design
 - *Time consuming manual* process
 - \rightarrow Difficult to evaluate many system partitioning configurations

Challenges of Developing **Complex Embedded Systems**



Current Status of SoC Design

- Increasing size and complexity (both SW and HW)
- System environment
 - Large number of concurrent events, real-time requirements
 - Huge cost in system verification and debugging
- System requirements
 - Functionality, power, performance, development cost, production cost, yield, reliability, portability, flexibility
- Slow growth in design productivity
 - Design methodologies highly dependent on CAD vendor tools
 - Absence of CAD technologies for application-specific design needs
- Design reuse, platform-based designs
 - Fixed SoC architecture
 - High overhead costs for IP qualification and IP integration



Current System-Level SoC Design Flow

7

Design Flow	System specificatior	Environment, Functionality, Performance, Power, Cost	Manual design refinement •Low design productivity •Bug-insertion risks	
	System modeling	Spec. analysis, Environment modeling, Algorithm design	 High design change cost High design optimization cost 	
	System partitioning	Concurrency extraction, Communication insertion	⇒Insufficient cost model at abstract design phases	
	Architecture design	SW/HW binding, Processor, Custom HW core, Bus system, Memory system	of manual designs themselves	
~	System verification	Debugging on HW prototype, Simulate actual environment	Huge debugging cost ⇒Insufficient system- level environment model	

System-Level SoC Design Flow: Current Problems and Future Solutions

- Manual process during design refinement
 - Large description gaps between design levels: bottleneck in design productivity
 - High risks of bug insertion: huge debugging costs
 - High costs for design changes (spec. changes, bug-fix, performance tuning)
- Insufficient design methodology for highly-parallel multiprocessor SoC architectures
 - System parititioning, concurrency extraction \rightarrow all manually designed
 - Limited architecture exploration due to low design productivity
 - Huge system verification cost (need to simulate actual system environment)
 - Absence of effective Real-time OS for MPSoCs
- Insufficient system-level design optimization environment
 - Algorithm optimization: implementation refinement of algorithms and data structures
 - System partitioning optimization: parallel task partitioning, task scheduling
 - Architecture optimization: SW/HW partitioning, processor configuration, HW synthesis, bus architecture, memory architecture
- → System-level design automation tool chain for drastic increase in design productivity
- → Establishing a robust system-level design optimization platform
- → Establishing a systematic design methodology for highly-parallel MPSoCs

Next Generation MPSoC Architecture



- Complex system environment:
 - Large number of peripherals
 - Highly concurrent under tight timing budget
- Highly-parallel scalable architecture:
 - Heterogeneous multiprocessors and customized HW cores
 - High bandwidth interconnect and high speed interface
- Fully optimized at system-level:
 - SW/HW designs (algorithms, processors, cores, architectures, interconnects)
 - System management (low-power, QoS guaranteed multitasking, concurrent event distributions)

Next Generation System-Level SoC Design Flow

System specification

Environment, Functionality, Performance, Power, Cost

System modeling

Spec. analysis, Environment modeling, Algorithm design

System partitioning

Concurrency extraction, Communication insertion

Architecture design

SW/HW binding, Processor, Custom HW core, Bus system, Memory system

System verification

Debugging on HW prototype, Simulate actual environment

System-level optimization platform

- Algorithm design
- Application-specific processor design
- Custom HW core design
- •System partitioning, concurrent modeling
- MPSoC architecture exploration
- Architecture evaluation model generation
- Creation of high quality IPs

System-level SoC synthesis platform

- System-level design automation tool chain
- Drastic increase in design productivity
- Elimination of bug-insertion risks
- ➔ Easy design changes

Ultra-fast system-level simulation platform

- Ultra-fast MPSoC simulation model
- System environment simulation model
- Drastic reduction in verification costs

Next Generation System-Level SoC Design Flow



Tightly-Coupled Thread (TCT) Model

- **TCT model** is a totally new framework which generates a concurrent execution model of *tightly-coupled threads* for functional blocks in MPSoCs.
 - TCT programming model provides a seamless design conversion from sequential C codes.
 - TCT concurrent execution model implements a wide variety of concurrent models such as *pipelining*, *data parallelisms*, *task parallelisms*, as well as their combinations.
 - TCT compiler automatically generates thread-level communication and synchronization instructions via explicit message passing to implement a fully distributed memory system.
 - TCT simulator schedules the sequential execution traces for evaluating the concurrent execution time, communication bandwidth, etc.

TCT Programming Model (1)

- Drastically simple programming model
 - \rightarrow Simply insert thread scope definitions in the C code
 - Syntax:



- Thread scope header introduces a new thread labeled "name" in the program
- Thread scope can have any compound statements in C, and also other thread scopes (thread scopes can be nested)

JPEG Encoder Example



TCT Programming Model (2)

- Compatibility to C standards :
 - By disabling the thread scope header with a simple preprocessor "#define THREAD(n)", it can be compiled with any standard C compiler to generate the computational equivalent sequential executable.
- Thread allocations in function calls:
 - Base-thread encloses the function body (all outermost thread scopes are nested in the base-thread scope)
 - On function calls, *callee's base-thread* is replaced with the *caller thread* (creating a *thread nesting structure through function calls*)

Thread Nesting Through Function Calls



TCT Programming Model (3)

- Global thread slicing tree
 - Expresses the thread scope nesting structure of the entire application
 - Function calls from different threads duplicate local thread slicing trees



TCT Programming Model (4)

- Restrictions in thread scope usage
 - Thread scopes must be a single-entry singleexit (SESE) region
 - no direct jumps into or out of thread scopes
 - Acyclic control dependence between threads
 - No recursive calls inside thread scopes
 - To guarantee thread slicing *tree* structure

TCT Concurrent Execution Model (1)



- Concurrent model *implied* in JPEG example :
 - 3 functional pipelines executing in parallel
 - → Combination of *functional pipelining* and *task parallelism* with complex data flow

TCT Concurrent Execution Model (2)

- Physical allocation of threads
 - Each thread is statically allocated to each processor
 - Each processor executes its own code inside the thread scope
- Thread communication model and memory model
 - External data transferred via message passing
 - Buffered communication channel
 - Finite buffer depth allocated at receiver side
 - Distributed memory model : no remote memory access
 - CM : communication module
 - LM : local memory
 - P : processor



20

TCT Concurrent Execution Model (3)

- Combining data-driven and program-driven principles
 - Program-driven (statically scheduled) thread execution
 - Data-driven thread interaction
- Differences between other hybrid data flow machines
 - Conventional notion of "threads" :
 - Non-blocking (all external data available before thread activation)
 - *Compiler-driven* thread partitioning (thread size tend to be small)
 - Many dynamically scheduled threads to fill the execution pipeline
 - Tightly-coupled threads :
 - Blocking (thread activation is independent of data availability)
 → thread interaction through *fine-grain data synchronization*
 - *Designer-driven* thread partitioning (thread can be of *any size*)
 - Very simple execution model

 \rightarrow allows easy processor customization

TCT Concurrent Execution Model (4)

- Thread activation and deactivation
 - Activated by *control dependent thread* based on control flow ("root-thread" processor activates by itself)
 - Deactivates itself upon completion of thread execution
 - No signaling of thread termination (not a fork-join model)



TCT Concurrent Execution Model (5)

- Data output synchronization:
 - Data transfer instruction sends data directly to the receiver buffer
 - If buffer is full, processor simply stalls until the buffer becomes non-full again.
- Data input synchronization:
 - Availability of external data checked by explicit data synchronization instruction before the first use of the data.
 - Processor simply stalls until data becomes available.



TCT Compilation Flow (1)

1. Front end

- General C parsing + thread scope parsing
- File linkage

2. Internal program representation

- Interprocedural control flow graph (ICFG)
 - CFG for each function (nodes *colored* with thread-IDs)
 - Each caller node in CFGs linked to target function's CFG
- Global thread slicing tree
 - Thread instantiation and duplication
- Thread control dependence tree
 - Similar to global thread slicing tree
 - Expresses the distributed thread activation flow

Interprocedural Control Flow Graph



Thread Control Dependence Tree



TCT Compilation Flow (2)

- 3. Interprocedural data dependence analysis
 - Convert ICFG to Interprocedural Dependence Flow Graph (IDFG)
 - Dependence flow graph (DFG): Generalization of SSA (static single assignment) form integrating *data flow* and *control flow*
 - IDFG captures side effects on globals and argument pointer dereferences during function calls
 - Data dependence extraction of data structures (i.e. arrays) and pointer dereferences
 - Modification to data structures modeled as combination of read-and-write operations
 - Flow-insensitive context-sensitive pointer alias analysis (currently handles single-level pointers only)

Interprocedural Data Dependence Analysis



TCT Compilation Flow (3)

4. Thread communication insertion

- Thread activation instructions
 - Directly derived from thread control dependence tree
- Data transfer instructions
 - Backward search on IDFG to locate the data modifiers in different threads
 - Entire data structure is bulk-transferred to reduce communication setup overhead
 - Loop optimization techniques (loop-invariance, loopprivatization) applied to reduce communications
- Data synchronization instructions
 - Inserted before the *first use instruction* in the thread

Thread Communication Insertion



- **CT**: control token (thread activation)
- DT: data transfer
- **DS**: data synchronization

thread activation flow
control flow
data flow

TCT Verification Tools (1)

- Behavioral verification on abtract processor model
 - Sequential single-thread execution
 - Debug source code
 - Generate program trace for profiling
 - Debug TCT parser (compare against standard C compiler)
 - Sequential execution on distributed memories
 - Debug TCT compiler for inserted communication instructions

TCT Verification Tools (2)

- Trace scheduler for concurrent thread execution modeling
 - Schedules the concurrent thread execution from the sequential execution program trace
 - Estimates concurrent thread execution time
 - Profile-based analysis of the critical path
 - Graphical schedule viewer

Trace Schedule Viewer (1)



Trace Schedule Viewer (2)



functional pipeline flow

Instruction-Level Schedule View

288001(5)	[+@L3.L1@L0]\$740 := \$739 & 255			
288002(5)	[+@L3.L1@L0]\$741 := (unsigned			
	[+@L3.L1@L0]r = \$741			
288003(4)	$[+@L3.L1@L0]DT : r => (L0.RGB_Y)$			
288004(12)	[+@L3.L1@L0]\$742 := rgb >> 8	[+@L3.L1@L0.RGB_Y]DS : {AT1:r}	[+@L3.L1@L0.RGB_Cb]DS : {AT1:r	[+@L3.L:
		[+@L3.L1@L0.RGB_Y]\$747 := (int	[+@L3.L1@L0.RGB_Cb]\$760 := (in	[+@L3.L:
288005(7)	[+@L3.L1@L0]\$743 := \$742 & 255	$[+@L3.L1@L0.RGB_Y]$ \$748 := \$747	[+@L3.L1@L0.RGB_Cb]\$761 := \$76	[+@L3.L:
288006(9)	[+@L3.L1@L0]\$744 := (unsigned	$[+@L3.L1@L0.RGB_Y]$749 := r2y[$	[+@L3.L1@L0.RGB_Cb]\$762 := r2y	[+@L3.L:
000007/ 4)	[+0L3.L10L0]g = 5/44			
288007(4)	[+013.11010]S74E = web C 25E	LIGT 2 TIGTO DCD VIDC . (AMO)	LULA LIALO DCD ChipC . (AMO) -	L. 013 1
288008(10)	[+0L3.L10L0]\$745 := rgb & 255	[+013.11010.RGB_1]DS : {A12:g}	[+012 11010 PCP_Ch1\$762 ([+@L3.L.
288009 (8)	$[+013, 11010]$ \$746 $\cdot = (unsigned)$	[+013, 11010, PCP, Y15751] := 5750	$[+013, 11010, RGB_Cb]$763 := (11)$	[+0L3.L]
200005(0)	[+0.3, 1.10.0]b = \$746			[1613.1
288010(7)	[+@1.3, 1.1@1.0]DT : b =>(1.0, RGB Y)	[+01.3, 1.101.0, RGB, Y1\$752 := r2y]	[+0.3, 0.10.0, RGB, Cb1\$765 := r2v	[+01.3.1.]
288011 (11)	[+@L3, L1@L0]\$786 := pY[1]	[+0L3.L10L0.RGB Y1DS : {AT4:##	[+013.11010.RGB_Cb1DS : {AT4:#	[+0L3.L]
	[second se	[+@L3.L1@L0.RGB Y1\$753 := \$749	[+0L3.L10L0.RGB Cb]\$766 := \$76	[+0L3.L]
288012(11)		[+@L3.L1@L0.RGB Y]DS : {AT3:b}	[+@L3.L1@L0.RGB Cb]DS : {AT3:b	[+@L3.L:
. ,		[+@L3.L1@L0.RGB Y]\$754 := (int	[+@L3.L1@L0.RGB Cb]\$767 := (in	[+@L3.L1
288013(9)		[+@L3.L1@L0.RGB_Y]\$755 := \$754	[+@L3.L1@L0.RGB_Cb]\$768 := \$76	[+@L3.L]
288014(8)		$[+@L3.L1@L0.RGB_Y]$ \$756 := r2y[[+@L3.L1@L0.RGB_Cb]\$769 := r2y	[+@L3.L:
288015(6)		$[+@L3.L1@L0.RGB_Y]$ \$757 := \$753	[+@L3.L1@L0.RGB_Cb]\$770 := \$76	[+@L3.L:
288016(6)		[+@L3.L1@L0.RGB_Y]\$758 := \$757	[+@L3.L1@L0.RGB_Cb]\$771 := \$77	[+@L3.L]
288017(9)		$[+013,11010,RGB_Y]$; = (uns	$[+013.11010.RGB_CD]$ (1)	[+@L3.L.
299019 (6)		$[+013,11010,RGB_{Y}]yy = 759	[+013,11010,RGB,Cb]pb = \$772	
200010(6)		$[+013,11010,RGB_1]DT : YY =>(L$	$[+013,11010,RGB_CD]DT : DD =>($	
200019(0)	[+0.3, 1.10.0] \$786 = ww	['els. higho. Kob_i]Endinread	[,els.hreb.keb_cb]Endfhread	L'ensit.
288020(4)	[+0.3, 1.10.0] \$787 = ppCb[1]			
200020(4)	[,eno.nrenolo.ov bbcp[]]			

Color conversion threads :

Data parallelism on Y, Cb, Cr components

Scheduling Comparison (1)



Design 1:

Single core pipeline: 6 threads Overall speedup: 4.42 Ave. parallelism at "core": 2.93



Design 2: 3x core pipeline: 16 threads Overall speedup: 5.54 Ave. parallelism at "core": 5.38

Scheduling Comparison (2)



Design 2 (from previous page): 3x core pipeline: 16 threads Overall speedup: 5.54 Ave. parallelism at "core": 5.38



Design 3:

3x core pipeline: 19 threads Overall speedup: 5.60 Ave. parallelism at "core": 7.87

Critical Path Analysis View



TCT Verification Tools (3)

- Program visualization
 - Control flow graph (CFG)
 - Dependence flow graph (DFG)
 - Call graph
 - Thread slicing tree
 - Thread interconnection graph

Control Flow Graph View



Dependence Flow Graph View





Thread Slicing Tree / Interconnection Graph



Thread slicing tree

Current Research Status

- Proof-of-concept MPSoC design on TCT model
 - Processor: simple RISC (originally designed)
 - Communication module (buffers implemented on local memory)
 - Interconnect architecture
 - Code generator back-end
 - Instruction-level multiprocessor simulator
 - VLSI implementation

MPSoC Development based on TCT Model



- AMBA system bus + 32-bit RISC core (host processor)
- Symmetric distributed-memory multiprocessor array (6 processors + AHB interface)
- Interconnect architecture
 - Full-crossbar structure with autonomous arbitration scheme
 - Small & fast:1000gates/processor, 2ns@0.18um
- Comm. module: 13K gates (including buffer controller)
 - High-speed burst transfer: 4 bytes/cycle (setup: 1-6 cycles)
 - Shared IO port architecture (1 read port & 1 write port / processor)
- TCT processor: 38K gates (including comm. module)
 - 4-stage pipeline, 32 registers, Harvard

Future Works (1)

- Refinement of automatic thread communication insertion algorithm
 - Instruction rescheduling
 - Thread merging (allocate multiple threads on each processor)
 - Thread spliting, thread boundary adjustment
- TCT programming techniques
 - Thread slicing strategies for different application classes
 - TCT IPs
- Interconnect issues
 - Direct connection, bus-based, router-based, multi-level switches
 - Thread-to-processor mapping schemes

Future Works (2)

- Testing on real applications
 - Video, audio, graphics, communications, networks, etc.
- MPSoC-specific problems
 - Processor customization and dedicated hardware module synthesis
 - Communication interface and interconnect customization
 - Develop MPSoC design methodology by combining existing CAD tools

Summary

- **Tightly-Coupled Thread (TCT) model** is a totally new framework for MPSoC designs where the system designers can use *sequential reference code in C* to directly generate a *concurrent execution model*.
- TCT model realizes complex combination of *functional pipelining*, *data parallelisms* and *task parallelisms*.
- TCT model is largely *orthogonal* to conventional parallelization compilers, VLIW compilers and parallel programming languages: all these technologies can utilize our TCT model to enhance their methodologies.