Systematic Generation of Embedded Software from High-level Models

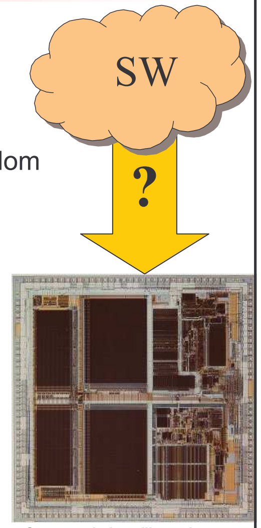
Gunar Schirner

Center for Embedded Computer Systems
University of California, Irvine

UCIrvine
University of California, Irvine

Motivation

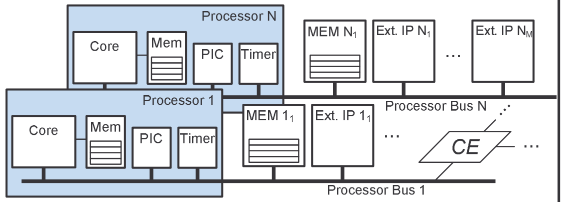
- Increasing complexity of Multi-Processor System-on-Chip (MPSoCs)
  - Feature demands
  - Production capabilities + Implementation freedom
- Increasing software content
  - Flexible solution
  - Addresses complexity
- How to create SW for MPSoC?
  - Avoid break in ESL flow:
    - Synthesize SW from abstract models



Source: simh.trailing-edge.com

Goals

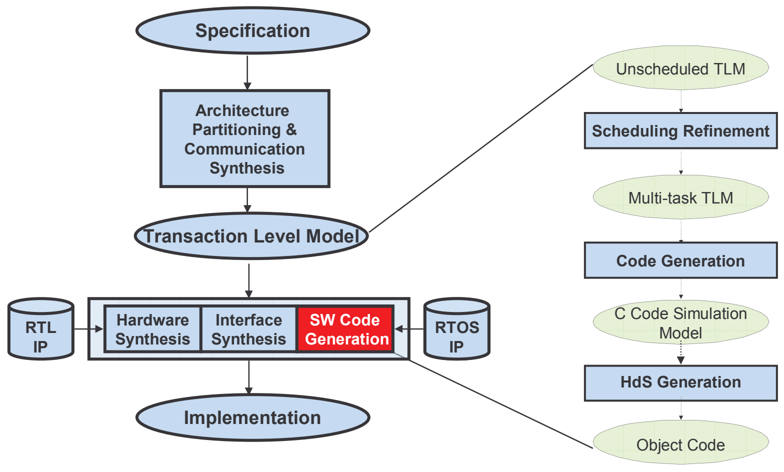
- Generate SW binaries for MPSoC from abstract specification
  - Eliminate tedious, error prone SW coding
  - Rapid exploration
  - High-level application developtment
  - Support wide range of system sizes
    - with RTOS / without RTOS (i.e. interrupt-driven)



Outline

- Introduction
- System Design Flow Overview
- Processor Modeling
- Software Generation
  - Code Generation
  - Hardware-dependent Software Synthesis
    - Communication Synthesis
    - Multi-Tasking
    - Binary Image Creation
- Experimental Results
- Conclusions

System Design Flow Overview



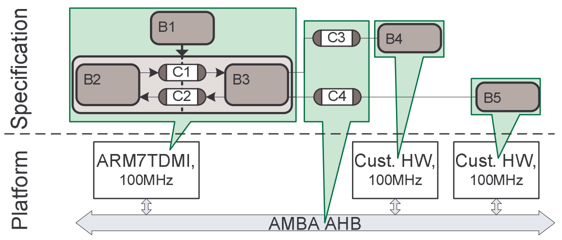
Outline

- Introduction
- System Design Flow Overview
- Processor Modeling
- Software Generation
  - Code Generation
  - Hardware-dependent Software Synthesis
    - Communication Synthesis
    - Multi-Tasking
    - Binary Image Creation
- Experimental Results
- Conclusions

System Design Flow Overview

Input Specification

- Capture Application in C (SpecC SLDL)
  - Computation
    - Organize code in behaviors (processes)
  - Communicate through point-to-point channels
    - Select from feature-rich selection
      - Synchronous / Asynchronous
      - Blocking / Non-Blocking
      - Synchronization only (e.g. semaphore, mutex, barrier)
- Architecture decisions:
  - Processor(s)
  - HW component(s)
  - Busses
  - Mapping
  - …



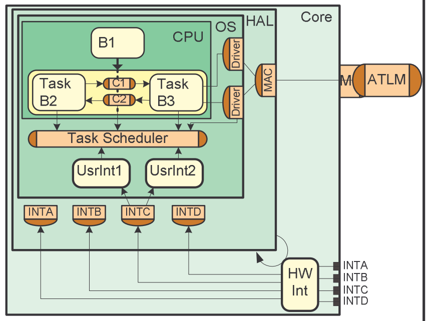
Processor Model: Overview

Automatic TLM Generation

- System Compiler (SCE) generates System TLM

  ➢ Model contains complete implementation information

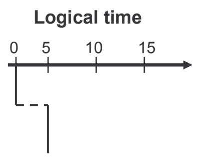
  – TLM Generation, processor modeling described in:
    - [Gerstlauer et. al, TCAD 09/2007], [Shin et. al, CODES+ISSS 2006], [Peng et. al, ASPDAC 2002], [Schirner et. al, ASPDAC 2007]

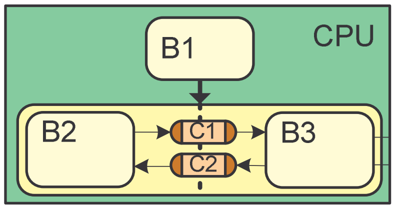


Processor Model: Application (1/5)

- Goal: Timed execution
- **Application Level**
  - User code in SLDL
    - Hierarchical set of behaviors
    - Channels
      - High-level, typed message passing
  - Timed execution
    - Back annotate C code with wait-for-time statements
    - At function level
      - Many other possibilities
  - Execute natively on simulation host
    - Discrete event simulator
    - Fast execution speed

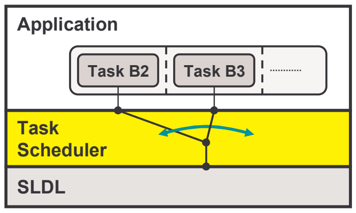
```
...
void f() {
  waitfor(5);
  ...
}
...
```

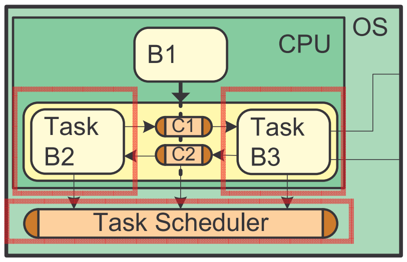




Processor Model: Task (2/5)

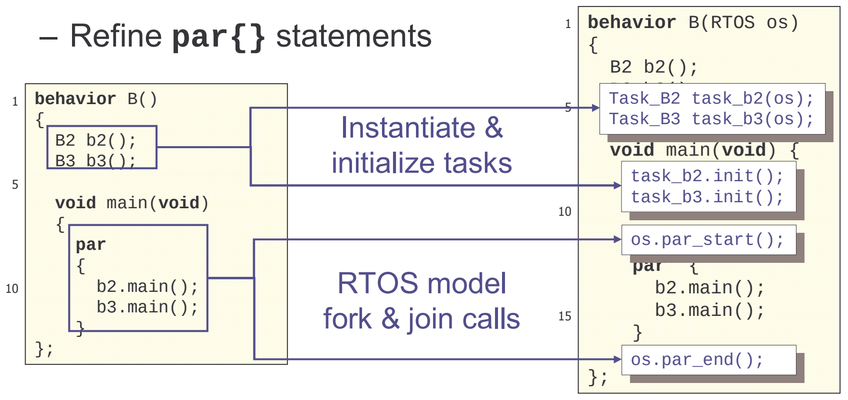
- Goal: Model dynamic scheduling effects
- **Task Level**
  - Group behaviors to tasks
  - Schedule tasks by abstract scheduler
    - Channel in SLDL
  - Wrap primitives that could trigger scheduling
    - Task start
    - Channel communication
    - wait-for-time





Processor Model: Task (2/5)

- Dynamic task creation
  - Refine **par{}** statements



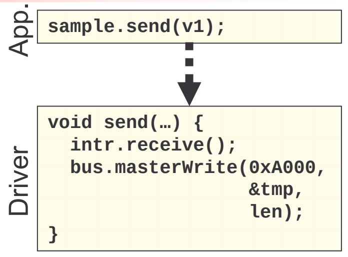
```
behavior B()
{
  B2 b2();
  B3 b3();

  void main(void)
  {
    par
    {
      b2.main();
      b3.main();
    }
};
```

```
behavior B(RTOS os)
{
  B2 b2();
  Task_B2 task_b2(os);
  Task_B3 task_b3(os);
  void main(void) {
    task_b2.init();
    task_b3.init();
    os.par_start();
    par {
      b2.main();
      b3.main();
    }
    os.par_end();
};
```

Source: H. Yu, R Doemer, D.Gajski 2004

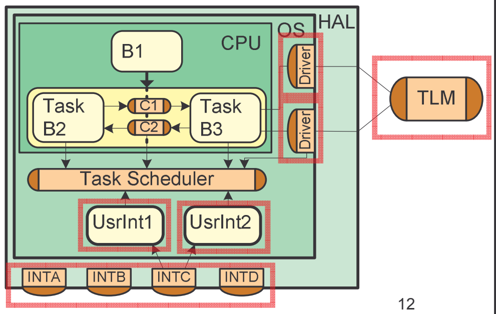
Processor Model: Firmware (3/5)

- Goal: External Communication
- **Firmware Level**
  - Software Drivers
    - Presentation, Session, Packeting
    - Synchronization (e.g. Interrupts)
  - TLM Bus model
    - User transactions
  - However, interrupts are unscheduled



App.
```
sample.send(v1);
```

Driver
```
void send(…) {
  intr.receive();
  bus.masterWrite(0xA000,
                  &tmp,
                  len);
}
```

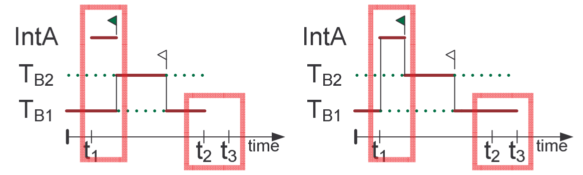


Processor Model: TLM (4/5)

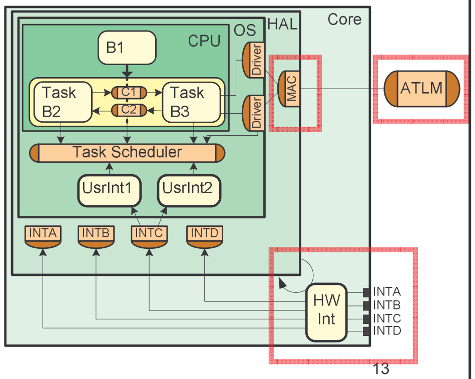
- Goal:
  Interrupt Scheduling

Unscheduled:

Scheduled:

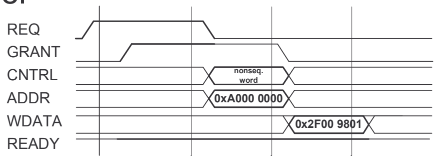


- **Processor TLM**
  - Hardware Interrupt Handling
    - Interrupt Scheduling
      - Suspend user code
      - Priority, Nesting
  - Media Access Control (MAC) for bus interface
    - Split user transaction into bus transaction
  - Arbitrated TLM bus model
  - Complete model!

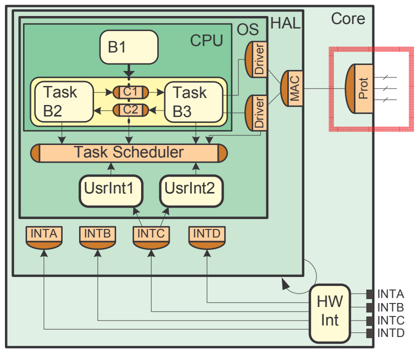


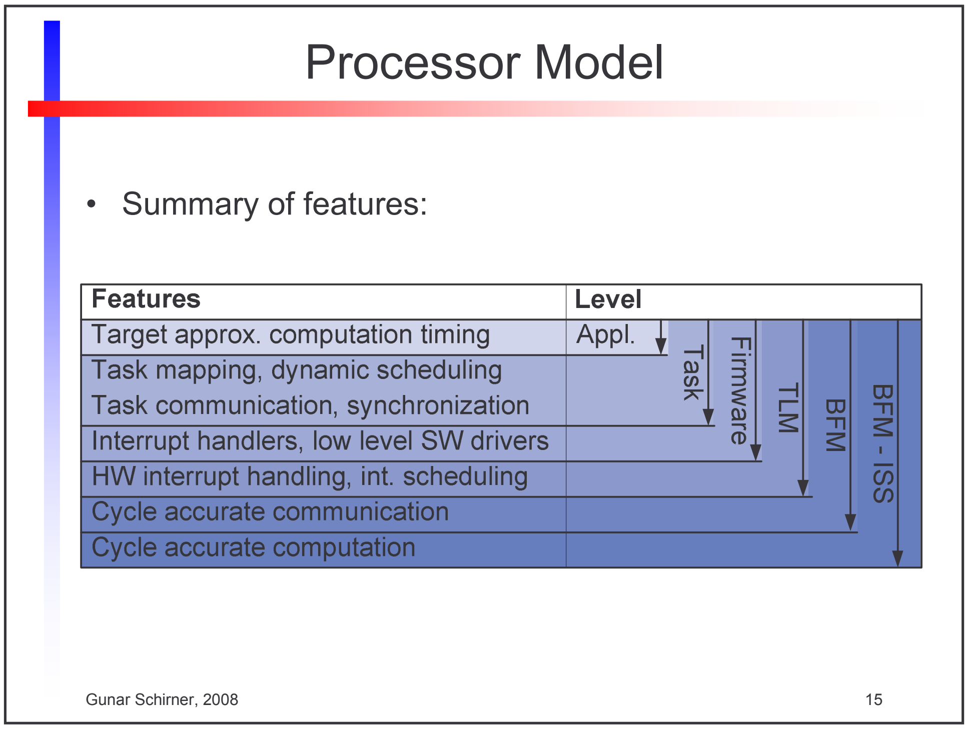
Processor Model: BFM (5/5)

- **Goal**: Accurate Bus Model



- **Processor Bus Functional Model**
  - Pin-accurate model of processor
    - Cycle approximate for SW execution
  - Bus model
    - Pin-accurate
    - Cycle-Accurate



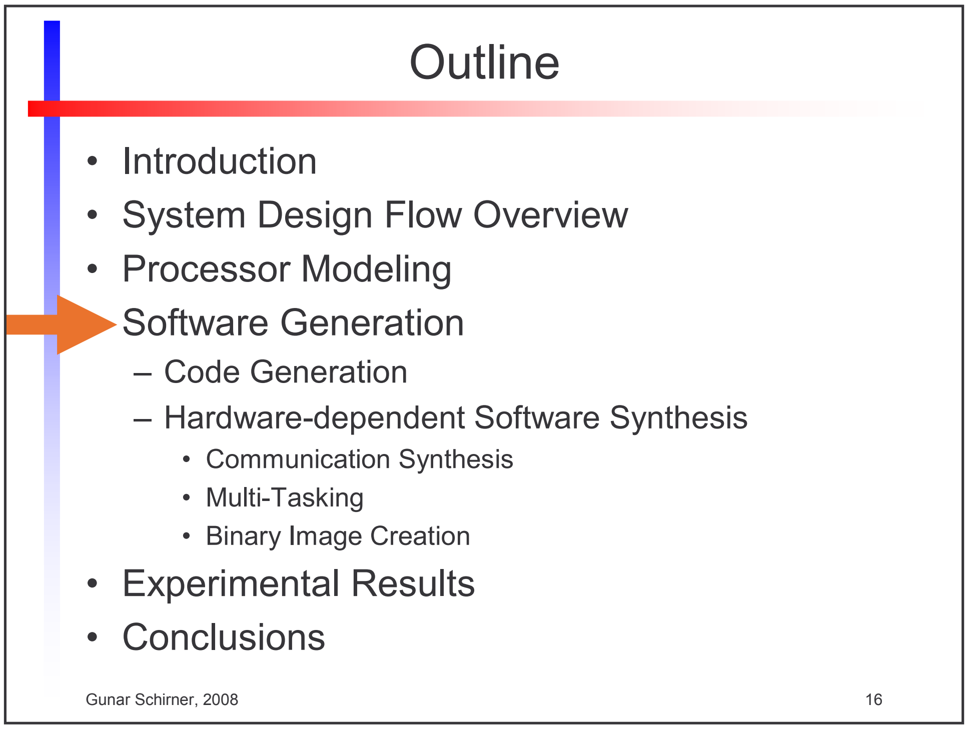


Processor Model

- Summary of features:

Features	Level
Target approx. computation timing	Appl.
Task mapping, dynamic scheduling Task communication, synchronization	Task
Interrupt handlers, low level SW drivers	Firmware
HW interrupt handling, int. scheduling	TLM
Cycle accurate communication	BFM
Cycle accurate computation	BFM - ISS

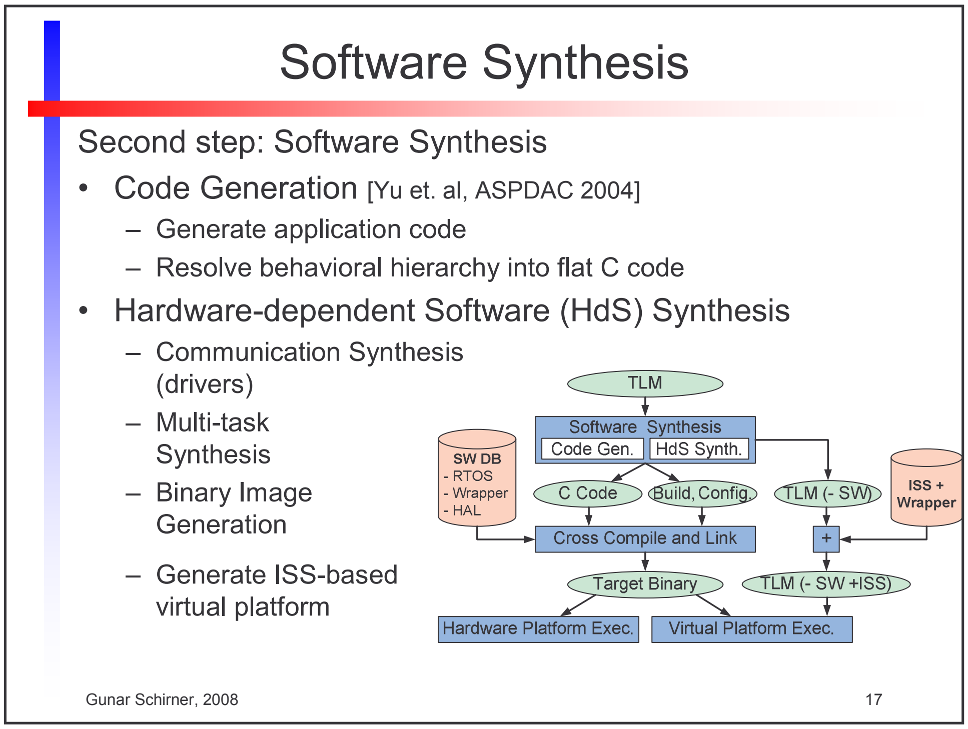
Gunar Schirner, 2008 15



Outline

- Introduction
- System Design Flow Overview
- Processor Modeling
- Software Generation
  - Code Generation
  - Hardware-dependent Software Synthesis
    - Communication Synthesis
    - Multi-Tasking
    - Binary Image Creation
- Experimental Results
- Conclusions

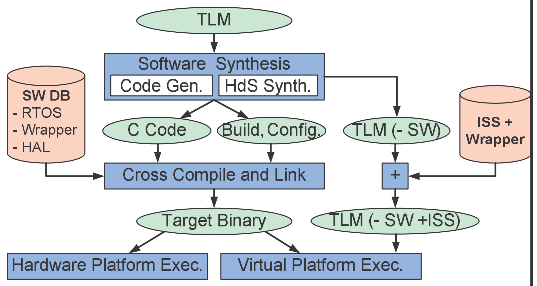
Gunar Schirner, 2008 16



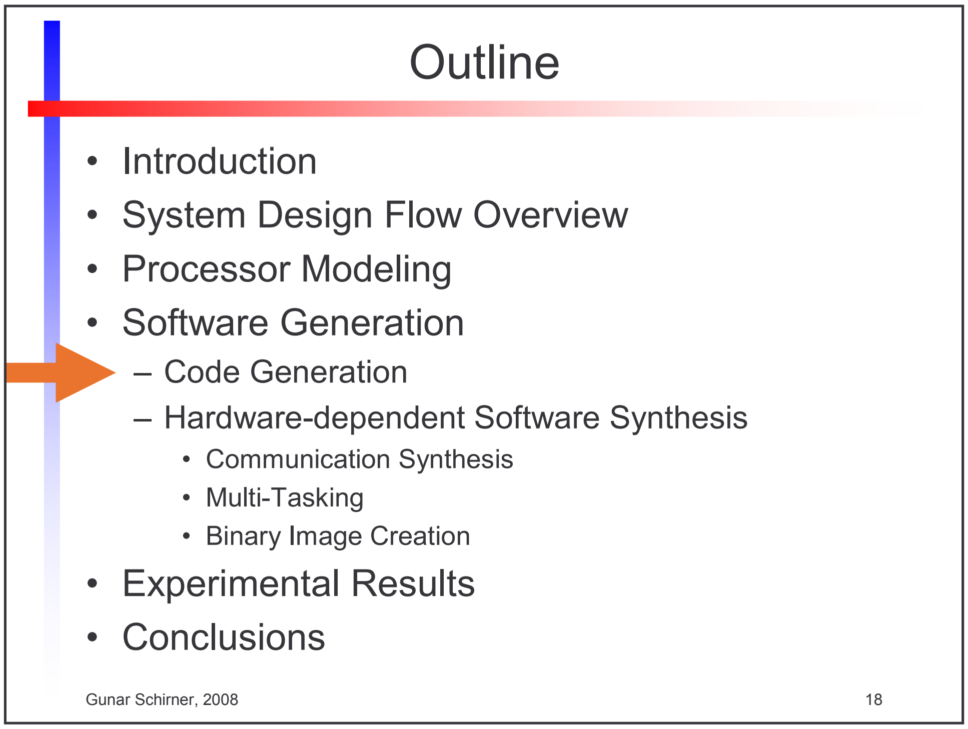
Software Synthesis

Second step: Software Synthesis

- Code Generation [Yu et. al, ASPDAC 2004]
  - Generate application code
  - Resolve behavioral hierarchy into flat C code
- Hardware-dependent Software (HdS) Synthesis
  - Communication Synthesis (drivers)
  - Multi-task Synthesis
  - Binary Image Generation
  - Generate ISS-based virtual platform



Gunar Schirner, 2008 17



Outline

- Introduction
- System Design Flow Overview
- Processor Modeling
- Software Generation
  - Code Generation
  - Hardware-dependent Software Synthesis
    - Communication Synthesis
    - Multi-Tasking
    - Binary Image Creation
- Experimental Results
- Conclusions

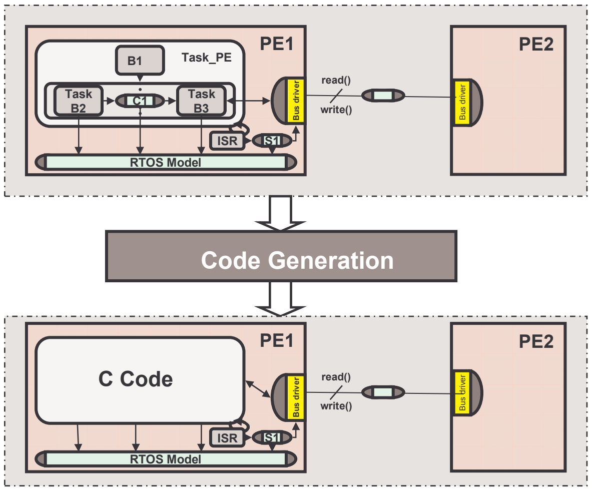
Gunar Schirner, 2008 18

Code Generation (a)

- Rules for C code generation
  1. Behaviors and channels are converted into C *struct*
  2. Child behaviors and channels are instantiated as C *struct* members inside the parent C *struct*
  3. Variables defined inside a behavior or channel are converted into data members of the corresponding C *struct*
  4. Ports of behavior or channel are converted into data members of the corresponding C *struct*
  5. Functions inside a behavior or channel are converted into global functions
  6. A static *struct* instantiation for each PE is added at the end of the output C code to allocate/initialize the data used by SW

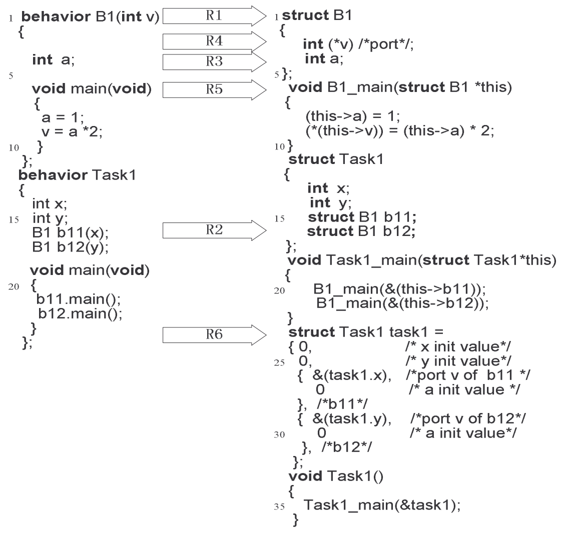
Source: H. Yu, R Doemer, D.Gajski 2004

Code Generation (b)



Source: H. Yu, R Doemer, D.Gajski 2004

C Code Generation Example



```
behavior B1(int v)
{

   int  a;

   void main(void)
   {
    a = 1;
    v = a *2;
   }
 };
behavior Task1
{
   int x;
   int y;
   B1 b11(x);
   B1 b12(y);

   void main(void)
   {
    b11.main();
    b12.main();
   }
 };
```

(a) SpecC Code

```
struct B1
{
    int (*v) /*port*/;
    int a;
};
 void B1_main(struct B1 *this)
 {
    (this->a) = 1;
    (*(this->v)) = (this->a) * 2;
}
 struct Task1
 {
    int  x;
    int  y;
    struct B1 b11;
    struct B1 b12;
 };
 void Task1_main(struct Task1*this)
 {
     B1_main(&(this->b11));
     B1_main(&(this->b12));
 }
 struct Task1 task1 =
 { 0,                  /* x init value*/
   0,                  /* y init value*/
  {  &(task1.x),   /*port v of  b11 */
      0              /* a init value */
  },  /*b11*/
  {  &(task1.y),    /*port v of b12*/
      0              /* a init value*/
  },  /*b12*/
 };
 void Task1()
 {
   Task1_main(&task1);
  }
```

(b) C Code

Gunar Schirner, 2008

Source: H. Yu, R Doemer, D.Gajski 2004

21

Outline

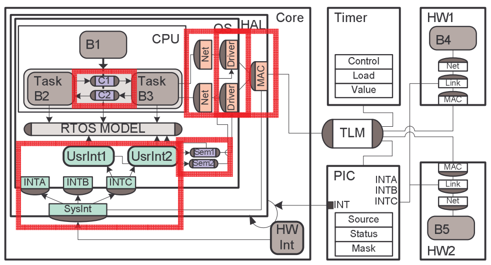
- Introduction
- System Design Flow Overview
- Processor Modeling
- Software Generation
  - Code Generation
  - Hardware-dependent Software Synthesis
    - Communication Synthesis
    - Multi-Tasking
    - Binary Image Creation
- Experimental Results
- Conclusions

Gunar Schirner, 2008

22

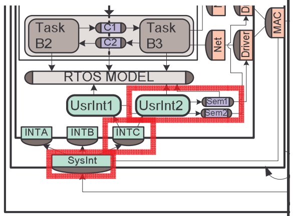
Hardware-dependent Software Synthesis

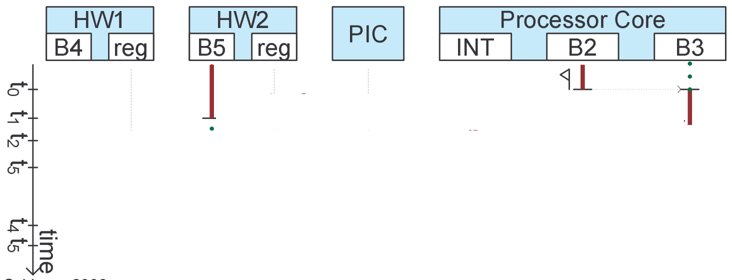
- Communication Synthesis
  - Internal communication
    - Replace with target specific implementation
      - e.g. RTOS semaphore, event, msg. queue
  - External communication
    - ISO / OSI layered to support heterogeneous architectures
    - Contains:
      - Marshalling
      - System addressing
      - Packetizing
      - MAC
  - Synchronization
    - Polling, or
    - Interrupt



Hardware-dependent Software Synthesis

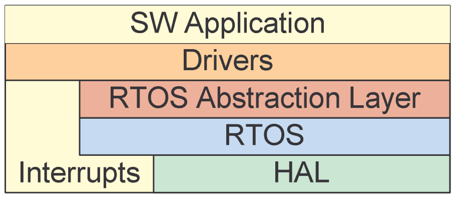
- Example: Synchronization by interrupt
  1) Low level interrupt handler
     - Preempts current task
  2) System interrupt handler
     - Checks PIC
  3) User-specific interrupt handler
     - Handles shared interrupts
  4) Semaphore
     - Releases task





Hardware-dependent Software Synthesis

- Multi-Task Synthesis
  - RTOS-based Multi-Tasking
    - Based on off the shelf RTOS
      - e.g. µC/OS-II, eCos, vxWorks
    - RTOS Abstraction Layer
      - Canonical Interface
        - » limit interdependency RTOS / Synthesis
    - Generate task management code
    - Internal communication



Hardware-dependent Software Synthesis

- Multi-task Synthesis
  - Example

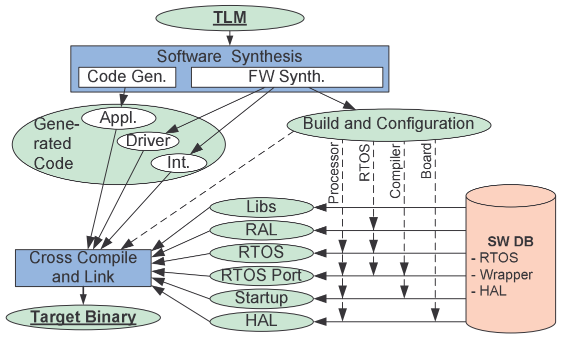
```
behavior B2B3(RTOS os)
{
  Task_B2 task_b2(os);
  Task_B3 task_b3(os);
  void main(void) {
    task_b2.init();
    task_b3.init();
    os.par_start();
    par  {
      b2.main();
      b3.main();
    }
    os.par_end();
};
```

```
struct B2B3
{ struct Task_B2 task_b2;
  struct Task_B3 task_b3;};
void *B2_main(void *arg)
{ struct Task_B2 *this=(struct Task_B2*)arg;
  ...
  pthread_exit(NULL); }
void *B3_main(void *arg)
{ struct Task_B3 *this=(struct Task_B3*)arg;
  ...
  pthread_exit(NULL); }
void *B2B3_main(void *arg)
{  struct B2B3 *this= (struct B2B3*)arg;
   int status; pthread_t *task_b2, *task_b3;
 taskCreate(task_b2, NULL,
            B2_main, &this->task_b2);
 taskCreate(task_b3, NULL,
            B3_main, &this->task_b3);
 taskJoin(*task_b2, (void **)&status);
 taskJoin(*task_b3, (void **)&status);
 taskTerminate(NULL);
}
```

Source: H. Yu, R Doemer, D.Gajski 2004

Binary Image Generation

- Generate binary for each processor
  - Generate build and configuration files
    - Select software database components
    - Configure RTOS
  - Cross compile and link
  - Significant effort in DB design
    - Minimize components
    - Analyze dependencies
    - Goal: flexible composition

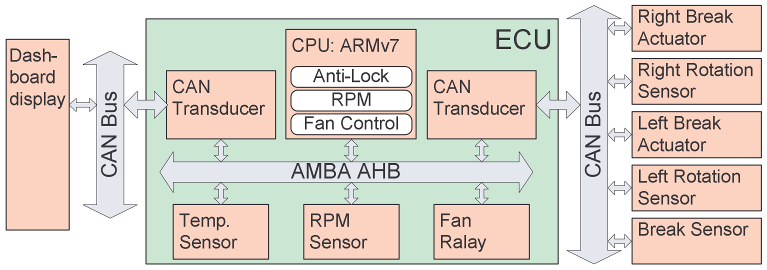


Outline

- Introduction
- System Design Flow Overview
- Processor Modeling
- Software Generation
  - Code Generation
  - Hardware-dependent Software Synthesis
    - Communication Synthesis
    - Multi-Tasking
    - Binary Image Creation
- Experimental Results
- Conclusions

Experimental Results

- Automotive Example
  - ARM7TDMI / 2x CAN / Anti-Lock Brakes, RPM, Fan



  - Generated System's Statistics

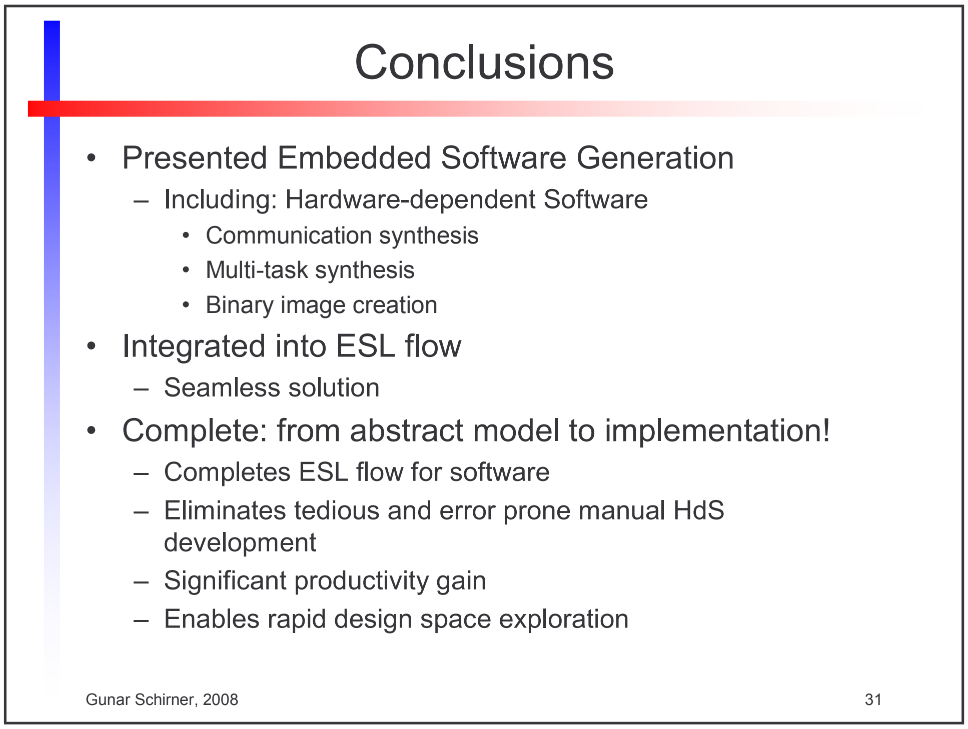
Footprint	36224 Bytes
Alloc. Stacks	4096 Bytes
CPU Busy Cycles	6.706 MCycles
Latency RPM Task	1794 Cycles
# Interrupts	1478

Experimental Results

- Synthesis Results
  - 6 Applications
    - diff. complexities
      - e.g. 2 … 14 ISRs
  - All functionally validated
  - Generated HdS
    - 200 … 1200 lines
    - Within less than 1s
- Manual implementation would take days!

Example	**GSM**	**Car**	**JPEG**	**Mp3 SW**	**Mp3 HW**	**Mp3 HW + JPEG**
Complexity						
IO/HW/Bus	4/1/1	9/2/3	2/0 /1	2/0/1	2/3/4	6/3/4
SW Behaviors	112	10	34	55	54	90
Channels	18	23	11	10	26	47
Tasks/ ISRs	2/ 3	3/ 5	1/ 2	1/ 3	1/ 8	3/ 14
Lines of Code, RTOS-based						
Application	-	153	818	13914	12548	13480
HdS	-	649	210	299	763	1186
Lines of Code, Interrupt-based						
Application	5921	210	797	13558	12218	-
HdS	377	575	187	256	660	-
Execution, RTOS-based						
CPU Cycles	-	6.7M	127.7M	185.8M	44.5M	174.6M
CPU Load	-	0.9%	100.0%	100.0%	30.9%	86.6%
Interrupts	-	1478	805	4195	1144	1914
Execution, Interrupt-based						
CPU Cycles	42.0M	5.1M	126.7M	182.3M	43.3M	-
CPU Load	42.5%	0.7%	100.0%	100.0%	30.5%	-
Interrupts	3451	1027	726	4078	1054	-

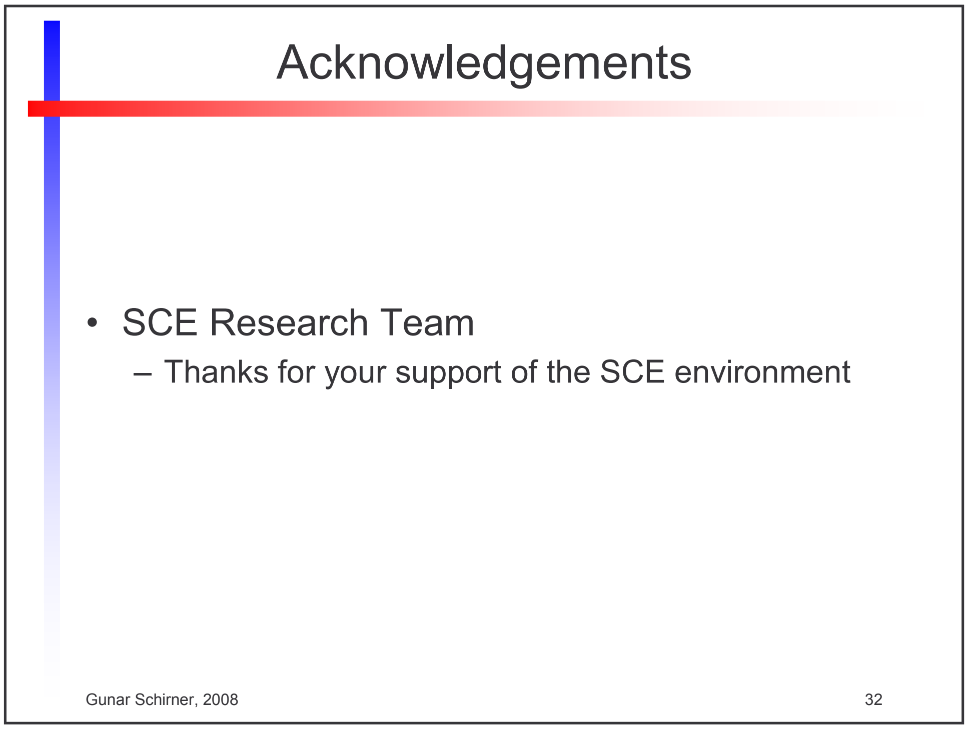
- ➢ Significant productivity gain
- ➢ Fast regeneration after platform modification



Conclusions

- Presented Embedded Software Generation
  - Including: Hardware-dependent Software
    - Communication synthesis
    - Multi-task synthesis
    - Binary image creation
- Integrated into ESL flow
  - Seamless solution
- Complete: from abstract model to implementation!
  - Completes ESL flow for software
  - Eliminates tedious and error prone manual HdS development
  - Significant productivity gain
  - Enables rapid design space exploration

Gunar Schirner, 2008 31



Acknowledgements

- SCE Research Team
  - Thanks for your support of the SCE environment

Gunar Schirner, 2008 32