LIAMA

Sino French Laboratory of Informatics, Automation and Applied Mathematics

Full System Simulation
of Embedded Systems

Vania JOLOBOFF
INRIA
FORMES Project at Tsinghua University Beijing, CHINA

INRIA

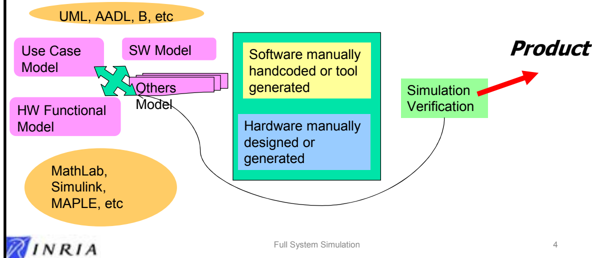
Embedded Systems

Industrial Objectives

- Improve reliability and safety of embedded systems (HW + SW)
- Lower time to market with shorter validation cycle

Model Driven Engineering Chain

- Validation/Verification is one step in the model driven engineering chain, but a critical one for products validation and certification



Different types of Simulation

- Mathematical / Physical Model Simulation
  - Extremely valuable in the design/exploration phase to experiment new methods and algorithms
    - MathLab, Simulink
- Hardware / Software Simulation
  - Different types of simulation according to the goals
    - Verify the hardware design (VHDL) : simulate the hardware at clock/gate and pin level
    - Verify the architecture and the software : simulate the hardware behavior with accuracy with respect to software, which *does not mean simulate every hardware detail*

HW+SW Full System Simulation

- Analogy flight simulator

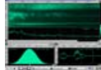
Input Control → Flight Simulator →

- Run the entire embedded application software over simulated hardware
  - CHALLENGE : *With identical behavior and performance, at low cost*

Advantage of Full System Simulation

- Software development can *take place before the hardware is ready*, therefore validation is faster
- Validation is *less costly and faster* because many engineers can run validation tests on a PC instead of sharing a few HW prototypes.
- Some things can be done with simulation that can hardly be done with hardware
  - Verifying correct hardware initialization, simulating defective hardware, internal observations, etc.

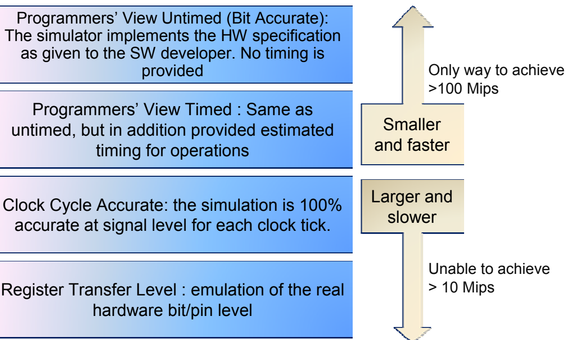
- Simulation tools can be connected with formal methods tools



Simulation Speed

- Example SPEC INT 2000 Test Suite
  - 6 trillions instructions
  - On a 3 GHz PC: 2000 seconds ~ 33 minutes
  - On a 3 Mips simulator 2,000,000 seconds ~ 70 days
- Other examples
  - A program running in 1 second on a 3 GHz host runs in 50 minutes if simulated at 1 Mips.
  - Simulating at 300+ Mips on a 3 GHz host means each target machine instruction is simulated with less than 10 instructions in the simulator host engine.

Simulation Hardware Abstraction Level

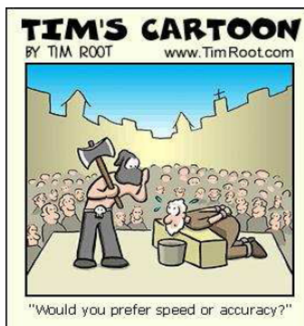


Full System Simulation : What is the good model ?

- From the software point of view
  - Simulation must be fast enough to run the programs in a few minutes, possibly hours for very long sessions ***but not days***…
  - Simulation must be complete, must not validate one piece of software independently from the others
    - Because the problems come from integration…
- From the hardware point of view
  - Simulation must be as accurate as possible
  - Calibration of hardware throughput is important
  - Integration of third party models must be possible

Simulation accuracy and speed

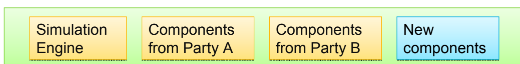
- Ideal: to obtain 100% accuracy with real-time simulation speed. Embedded system real time 300 – 500 Mips
- Cycle accurate HW models (VHDL) are much too slow for software validation…
- Need higher level of abstraction



你是要快还是要准确?

Simulation requires standards

- It is hard to build a complete system simulator from scratch
  - Many components, some of them very complex
- Necessity to re-use existing models
  - From corporate databases and libraries
  - From Third-Parties
- This can only be achieved if there are standard interfaces between the components

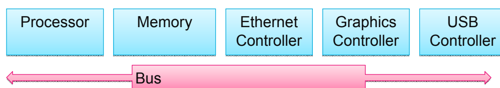


Two standards for interoperability

- SystemC : Simulate a set of hardware components
- TLM : Communication between components

SystemC in one lesson

- Hardware components are elements working simultaneously (parallel) communicating through some wiring

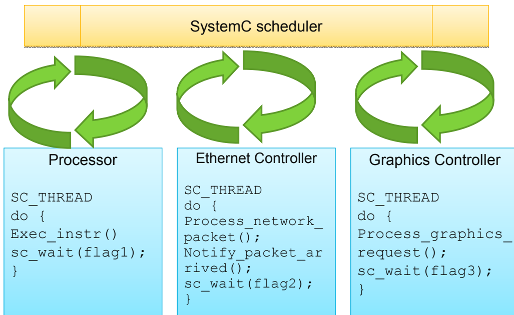


- SystemC (IEEE standard 1666) provides for simulation of parallel hardware components

SystemC in one lesson (2)

- SystemC fundamental concepts
  - Representation
    - MODULE to implement a component (which may contain other MODULEs / components
    - PORT to implement a communication point
  - Control
    - SC_THREAD to simulate parallel processes
    - `sc_start()` to start the simulation after all modules have been constructed
    - `sc_wait()` to wait for something (time, event,…)
    - `sc_notify()` to signal an event
    - A scheduler to schedule threads according to an algorithm described in the standard

SystemC threads and scheduler



SystemC Use Cases

- SystemC can be used for
  - Bit accurate simulation
  - Cycle accurate simulation
  - Cycle accurate simulation with hardware synthesis
    - using a specific subset

SystemC scheduling

- Algorithm steps through so called Delta-cycles that execute in no time
  - 1. Initialization Phase
  - 2. Evaluate Phase: From the set of processes that are ready to run, select a process and resume its execution. The order in which processes are selected for execution from the set of processes that are ready to run is unspecified. The execution of a process may cause immediate event notifications to occur, possibly resulting in additional processes becoming ready to run in the same evaluate phase.
  - 3. Repeat step 2 for any other processes that are ready to run.
  - 4. Update Phase: Execute any pending calls to update() from calls to the request update() function executed in the evaluate phase.
  - 5. If there are pending delta-delay notifications, determine which processes are ready to run and go to step 2.
  - 6. If there are no more timed event notifications, the simulation is finished.
  - 7. Else, advance the current simulation time to the time of the earliest (next) pending timed event notification.
  - 8. Determine which processes become ready to run due to the events that have pending notifications at the current time. Go to tep 2.
- Or the simulation stops by calling stop function.

Communication Interface

Provide

Require

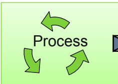
- Java Style Example

```
class Drawable {  … getResolution() … }

interface Printable {
   … print (Drawable d) …
 }

class Rectangle implements Printable {
   print(d) { print a rectangle on drawable d}
class Circle implements Printable {}
   print(d) { print a circle on drawable d}
```


TLM: Transaction Level Modeling



- TLM provides standard interfaces for communication between simulation models
- The communication between an **initiator** and a **target** is abstracted
  - **transactions** are routed from initiators to targets through **sockets** defining **interfaces** for the communications.
- TLM is now a standard supported by Intel, ARM, NXP, Texas Instruments, Infineon, ST Microelectronics, Forte, Mentor Graphics, CoWare, Synopsis, Canon, Nokia, etc.

Transaction Level Modeling

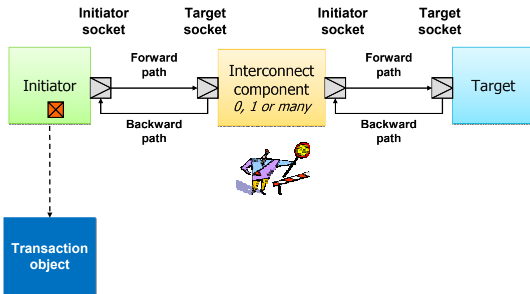
- The initiator and targets have sockets that provide/require an interface, e.g. a set of functions that perform the transaction
- The initiator does not need to be fully aware of the destination details, it just need to know the interface provided by the socket and the address of the target
- Possibility of several intermediate steps in communication

Transaction Level Modeling (1)

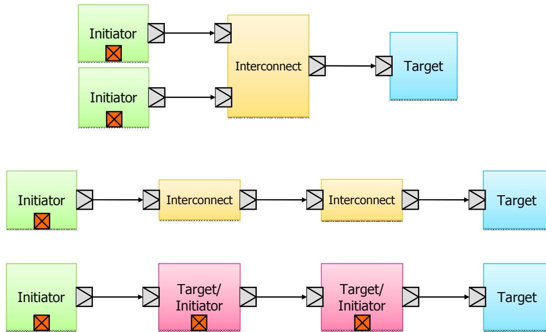
Initiator	Target

MODULE A
Socket S1
 	
THREAD {
While (cond) {
 	
S1.transaction(read, payload, response)
 	
Wait(something)
}
	MODULE B
	
	Socket S2 accepts transaction
	read (payload, response) {
	response = …
	}

TLM transactions



TLM Connectivity



TLM Blocking vs Non-blocking Transport

- Blocking transport interface
  - Includes timing annotation
  - Typically used with loosely-timed coding style
  - Forward path only
- Non-blocking transport interface
  - Includes timing annotation and transaction phases
  - Typically used with approximately-timed coding style
  - Called on forward and backward paths
- Share the same transaction type for interoperability
- Unified interface and sockets – can be mixed

TLM 2.0 Blocking Transport

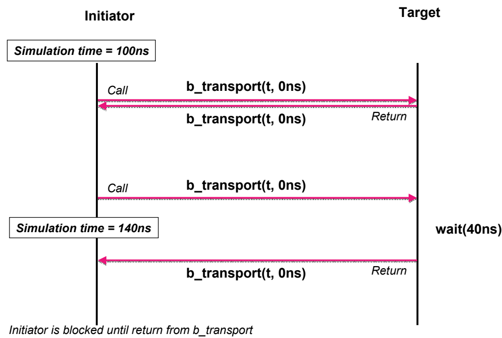
Transaction type

```
template < typename  TRANS  =  tlm_generic_payload >

class tlm_blocking_transport_if : public virtual sc_core::sc_interface {
public:
    virtual   void   b_transport ( TRANS& trans , sc_core::sc_time& t )  =  0;
};
```

Transaction object   *Timing annotation*

TLM 2.0 Blocking Transport



TLM 2.0 Non-blocking Transport

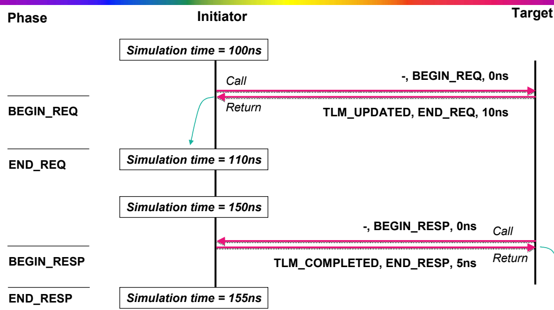
```
enum tlm_sync_enum { TLM_ACCEPTED, TLM_UPDATED, TLM_COMPLETED };


template <  typename TRANS = tlm_generic_payload,
            typename PHASE = tlm_phase>

class tlm_fw_nonblocking_transport_if : public virtual sc_core::sc_interface {
public:
   virtual tlm_sync_enum nb_transport(   TRANS& trans,
                                PHASE& phase,
                                sc_core::sc_time& t ) = 0;
};
```

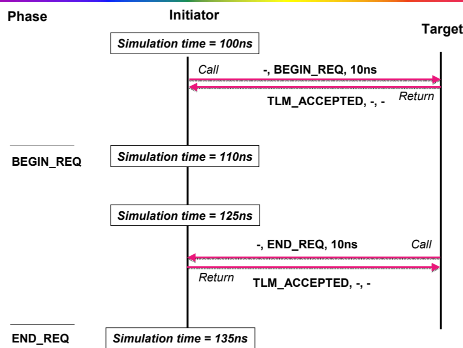
Trans, phase and time arguments set by caller and modified by callee

TLM 2.0 Non Blocking Transport



Callee annotates delay to next transition, caller waits

Timing Annotation



TLM 2.0 Direct Memory Interface

- Memory access in TLM 1.0
  socket.transaction(read, address, ret_value)
  - Goes through the interface forwarding process
  - Slow !!!
- Direct Memory Access in TLM 2.0
  - Special initial transaction :
    status = get_direct_mem_ptr(transaction, dmi_data);
  - *Returns table [range]*
  - Then use value = table[address] (*within the range*)

Full System Simulation Level of Abstraction

- Abstract the hardware to "Bit Accurate Programmer's view", that is, the simulation model behaves exactly like the real hardware from the software programmer's view point
  - The software developers can run the software with the same behavior (but slower)
  - The hardware developers validate that the hardware is functionally correct
  - They can obtain valuable information about the software requirements
    - Bus transfers, FIFO sizes, etc.

FORMES Simulator Goal

- **Build a full simulation environment** simulating the platform as a bit-accurate simulator
- Provide ***base simulation engine*** and off-the-shelf simulators for *commercial off-the-shelf* CPUs
  - ARM, MIPS (Loongson), PowerPC
- ***Use SystemC and TLM*** as the foundation model to standardize interfaces
- Make the simulation environment, **portable** to run on multiple simulations hosts, **open** to multiple architectures
- **Associate formal methods tools** to the simulation framework to prove properties of the simulated models, speed-up the simulation process, and provide better test validation

Computer Architecture Reminder

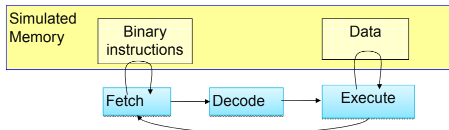
- Processors execute instructions
  - Arithmetic / Logic instructions on integers or floating point
  - Condition and Branch instructions
  - Memory access instructions
  - Peripheral commands instructions (viewed as memory)
- A processor may be interrupted by external devices
  - An interrupt stops the current program and executes another program : the interrupt service routine
  - After interrupt is handled it returns to normal execution
  - On virtually all processors, an instruction is *atomic,* it cannot be interrupted in the middle.
    - interrupts are checked before each instruction

Instruction Set Simulation (ISS)

Early simulation: **Interpreted Simulation**

- Simulate the instruction fetch/decode/execute of the target processor
- Code does essentially

```
do {
        instruction = Fetch (current_pc);
        Decode (instruction);
        Execute (instruction);
} until End Of Program
```



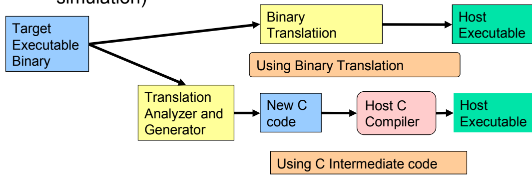
Inefficiency due to decode multiple times the same instructions : speed < 10 Mips

How to do better ?

- Translation:
  - Translate in some way the executable code into another representation run on the simulation host
  - Eliminate most of the decode time, speed up the execute time
  - Cache the translated code for re-use

Static Translation

- Static translation compiles the target program into a host program
  - Fast but not flexible
  - Does not handle all cases, for example dynamically loaded libraries, or self modifying code
  - Bad throughput in development mode (cycle compile + simulation)



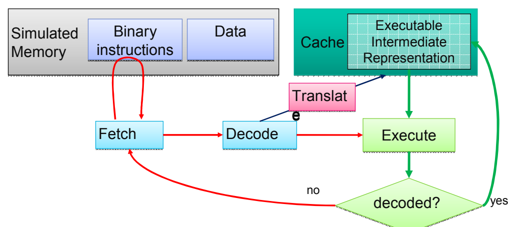
Dynamic (Cached) Translation

- Translation:
  - Eliminate most of the decode time, speed up the execute time
  - Entire compilation step included into simulation run-time.
  - Cache the translated code for re-use
- Advantage
  - Handles all cases, including self modifying code or code generating applications
  - No additional step required before running simulation
  - No problem to mix with other TLM modules
  - Much faster simulation
- Inconvenient
  - The translation time is added to the simulation time
  - However possibilities to decrease translation time with some pre-compile steps…

Dynamic Translation | Simulation time = translation time + execution time

Dynamic (Cached) Translation (2)

- Translation can be done on segment or page basis
- Speed increases significantly > 10 Mips



FORMES Simulator: SimSoC

Embedded Application Software

Processor Model

Dynamic Translation

MMU

DMI

Memory

TLM bus

Translation Cache

HW TLM models

SimSoC simulation platform

Parallell Simulation : SystemC Kernel

- SimSoC develops a simulation framework based on SystemC and TLM with ISS'es for the most common commercial processors
  - ARM (done), PowerPC (half-done), MIPS (started)

SimSoC ISS

- SimSoC implements several types of Instruction Set Simulator in order to make comparison and also to have different degrees of accuracy:
  - P0 mode. Interpreted mode. Interrupts are checked after every instruction (*like on the real hardware*)
  - P1 mode. Simple dynamic translation with no partial evaluation.
  - P2 mode. Dynamic translation with partial evaluation and possibility to check for interrupts at specified intervals.
  - P3 mode. Code generation under research

Partial Evaluation

- Invented in the 70's
  - Yoshihiko Futamura (1971). "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler". Systems, Computers, Controls 2 (5). Reprinted in Higher-Order and Symbolic Computation 12 (4): 381–391, 1999
  - A program P is usually made to operate on any data. A computer program, *prog*, is a mapping of input data into output data:

`prog :` $I_{static} \times I_{dynamic} \rightarrow O$

The *static data* I_{static} is the part of the input data known at compile time. Partial evaluation transforms `prog` into `prog*` : $I_{dynamic} \rightarrow O$ at compile time.

prog * is called the "residual program" and should run more efficiently than the original program.

```
proc(pointer ptr, int v) {
if (ptr == NULL)
   return (error);
if ( v < 100)
   small_number(ptr->f, v)
else
   big_number(ptr->g, v)
}
```

```
p = malloc(size);
if (p == NULL)
        error("out of memory");
else proc(p, 50);
```

Dynamic translation with partial evaluation

- At instruction decoding time, you know which operation on which data
- Hence possible to use partial evaluation compilation techniques to translate
- Uses more memory, but memory is cheap and caches are larger and larger

Partial Evaluation in Translation

- Partial Evaluation Technique can be used in binary translation
- Many instructions to reach the internal switch case on example
- But this information is known at decoding time…
  - Possible to use partial evaluation
- Can be specialized into multiple specialized functions with arguments evaluated at compile time
- Each function uses many less instructions
  - significant performance enhancement

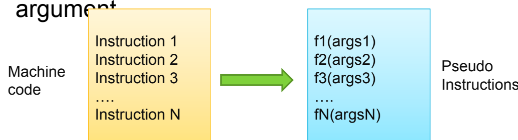
- Example

```
Operation(op, operand1,  operand2)
switch(op){
    case ADD:
      switch(operand1){
          case A:…
          case B: switch (operand2) {
              case X: ..
              case Y:…
                   }
          }
      break;
    case SUB:
      substract code
    case MUL:
      multiplye code
…}
```

- Multiple “specialized” functions
  - ADD_operandA_operandX() {}
  - ADD_operandA_operandY() {}
  - ADD_operandB_operandX() {}
  - SUB_operandA_operandB() {}
  - … etc …

SimSoC partial evaluation

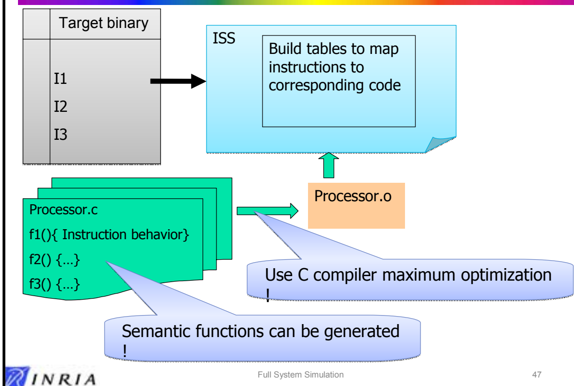
- Translate each machine instructions into a pseudo-instruction that contains a pointer to the partial evaluation residual function *f*, called the semantic function, with the dynamic input as argument



Generating Semantic Functions

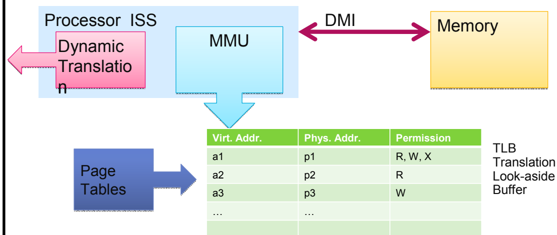
- The number of such semantics function is potentially very large (2^{32} for 32 bits instructions) but finite, and in fact manageable corresponding to computer architecture
- Example ARM
  - 15 condition modes, 2 post-operation mode, 11 operand modes, 3 addressing mode, 4 operations (and, or, eor, not)
  - 4*3*11*2*15 = 3960 functions for boolean instructions
- Therefore semantic functions can be generated and compiled before simulation and loaded at run time.

Partially Evaluated Pre-compiled Code



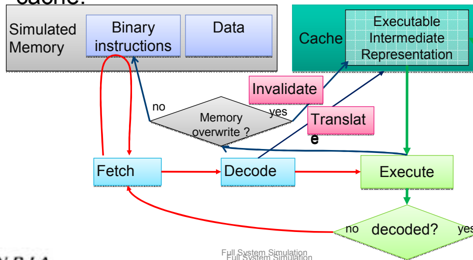
MMU Simulation

- Simulation of MMU Memory Management Unit
- MMU verifies that memory access is permitted



Memory management

- The program may be deleted or modified. The cache must remain coherent. Necessary to keep track of memory access and possibly invalidate cache.



MMU Simulation (2)

- Because MMU associative hardware search is simulated with software table lookup, it is slow.
- Speed up solution:
  - Use a very large table
    - Example : for 32 bits virtual memory with pages of size 4K bytes (12 bits) use a 2^{20} elements table to cache every page. Search done in one memory access.
- Checking memory overwrite is slow if one test for every memory access instruction
  - Use host system memory protection

Simulation Speed Results

	Interpreted	Simple Dynamic Translation	Dynamic Translation with specialization
ARM32 no optimization	6.62 Mips	15.6 Mips	59.9 Mips
ARM32 max optimization	6.84 Mips	15.3 Mips	82.3 Mips
THUMB no optimization	5.01 Mips	17.3 Mips	65.4 Mips
THUMB max optimization	5.40 Mips	17.8 Mips	60.7 Mips

Influence of Direct Memory Access

	No dynamic translation		Dynamic translation	
	no DMI	with DMI	no DMI	with DMI
ARM32 no optimization	7.2 Mips	11.8 Mips	32 Mips	123 Mips
ARM32 max optimization	7.8 Mips	11.1 Mips	75 Mips	140 Mips
THUMB no optimization	5.9 Mips	10.8 Mips	61 Mips	123 Mips
THUMB max optimization	5.9 Mips	10 Mips	75 Mips	110 Mips

FORMES Simulator status as of 2009/01

- Simulation Framework developed for
  - ARM architecture (Arm Version 5)
  - PowerPC under development (2009)
  - MIPS targeted for 2010
  - Compliant with standard IEEE 1666 and TLM

	Interpreted	**Simple Translation**	**Optimized Translation**
ARM32 –O0	6.62 Mips	15.6 Mips	59.9 Mips
ARM32 –O3	6.84 Mips	15.3 Mips	82.3 Mips
THUMB –O0	5.01 Mips	17.3 Mips	65.4 Mips
THUMB –O3	5.40 Mips	17.8 Mips	60.7 Mips

Over-specialization decreases performance…

	Specialized	**Over-specialized**
ARM32 no opt.	59.9 Mips	***58.6* Mips**
ARM32 max optimization	82.3 Mips	***78.3* Mips**

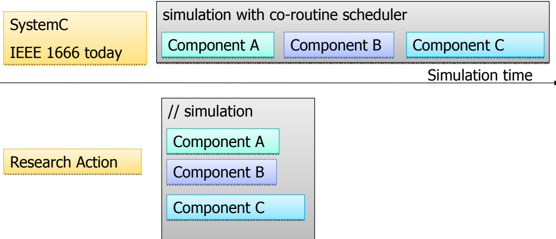
- Reason: over-specialization creates tens of thousands of functions, each of them rarely used.
  - They do not all fit in the host cache….
  - Cache thrashing on the host deteriorates performance.
- Conclusion: specialize until the cache is full...

Research Directions

- Support multi-cores / many-cores platforms
- Improve simulation speed
- Develop tools for ease of use
- Simulate defective hardware

Simulating Multi/Many Cores

- Parallelize simulation for the next generation of many-cores circuits (> 32 processors)

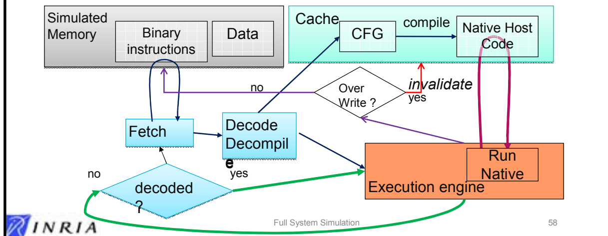


High Speed Simulation

- Dramatically improve simulation speed using most recent compiling technologies
  - Dynamically translate simulated binary code into optimized host code to obtain an order of magnitude speed up
  - Goal : simulate a 300 MHz chip at real speed on a 3 GHz PC.
  - Use sophisticated compiler techniques.
    - Decompile the machine code into an abstract control flow graph CFG as close as possible to original source code
      - Undecidable problem, but heuristics works 80% of time…
    - Recompile this CFG into host code with maximum optimization
- Issue : Accuracy
  - Use of this technique much less effective if interrupts are checked after each instruction

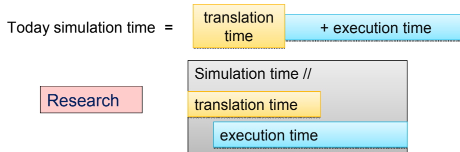
Code Generation

- The machine code is first decompiled into a *Control Flow Graph* then recompiled into host machine code and executed under control of execution engine
- Two existing such simulators : Boston University, Edinburgh University
- Intermediate solution : QEMU builds the CFG and generates a sequence of macro instructions



Research : Parallelize Translation

- On multi-processors simulation hosts, it is possible to translate ***not*** **just-in-time** (when necessary to execute an instruction) but in ***parallel*** **ahead-of-time**
- The translation time does not hurt performance when the process is parallel to the execution process
  - Since it does not hurt performance, the compilation can be made more complex with more optimizations



Ease of Use

- Currently, simulators are build by manually assembling components using SystemC interfaces:
  - Time consuming, errors, little flexibility…
- Research:
  - Generate the simulator(s) from a library of existing industry components models using a higher level tool, generating SystemC code, with some kind of type checking control to detect errors

Conclusion

- Full System Simulation has achieved significant results but we are still far from simulating many-cores circuits at real speed. We have work to do …

谢谢

FORMES:

A joint project between INRIA and Tsinghua and Beihang University