A Low Cost SystemC Acceleration on Multi-Core GNU/Linux Platforms

Cicerone Mihalache
Kotys LLC, Santa Clara, CA
www.kotys.biz
cicerone.mihalache@kotys.biz

Agenda

- Why Speed?
- Simulation Acceleration
- Typical SystemC Simulation
- Parallel SystemC Simulation
- Acceleration Conditions
- Experimental Results
- IPC Adapter
- References
- Conclusions

Why Speed?

- Complex software running on complex hardware
- Simulation time too long (despite the speed gain from SystemC modelling at high level of abstraction)
- Long simulation times have a big impact on the cost and time to market of an SoC, due to the iterative nature of the debugging process Simulate->Debug->Fix problem -> Simulate -> ...

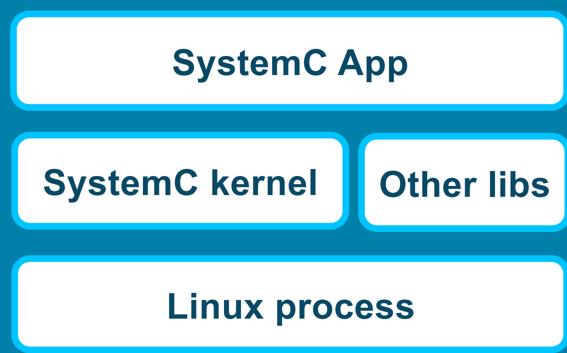
Simulation Acceleration

Related work:

- EZUDHEEN, P., Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines
- NAGUIB, Y. N. Speeding up SystemC simulation through process splitting
- CHOPARDL B. A Conservative Approach to SystemC Parallelization

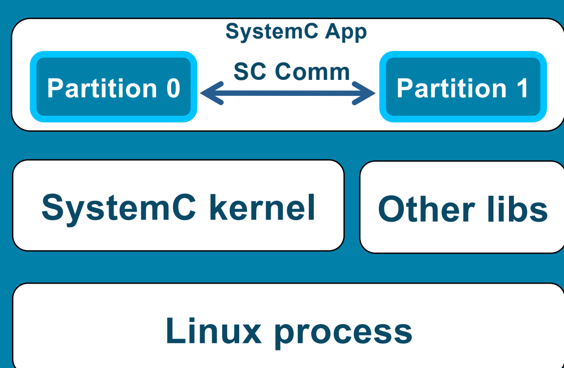
Fast simulation and good debuggability are key factors for reducing the price of SoC development.

Typical SystemC Simulation



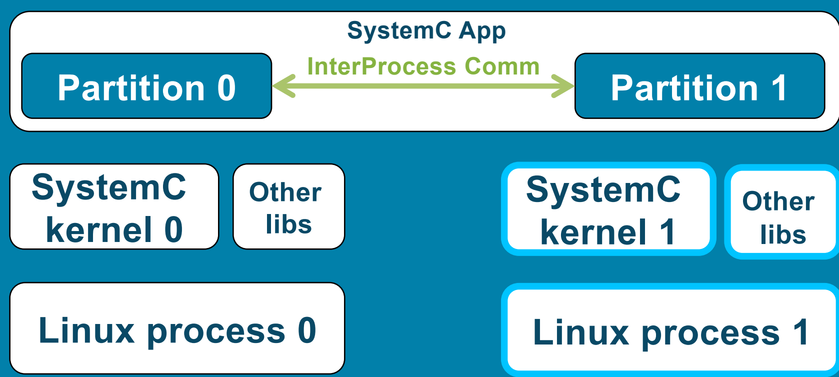
- One Linux process
- Simulation time: T_{one_core}

Typical SystemC Simulation



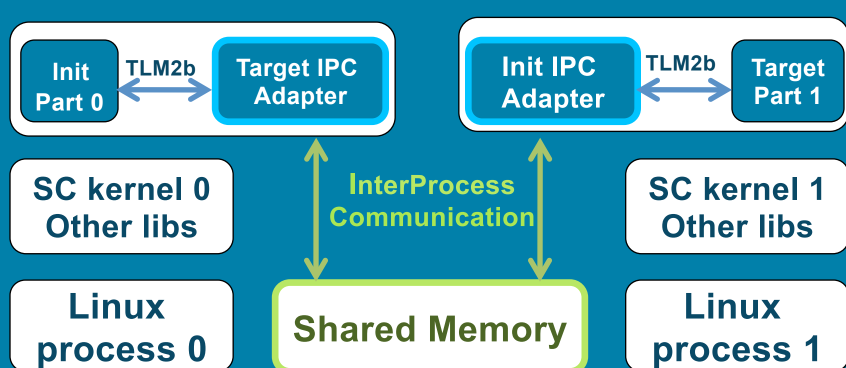
- One Linux process
- Simulation time: $T_{one\_core} = T_{part0} + T_{part1}$

Parallel SystemC Simulation



- Two Linux processes, each running on its own CPU core
- Simulation time: $T_{two\_cores} = \max(T_{part0}, T_{part1}) + T_{IPC}$

Parallel SystemC Simulation



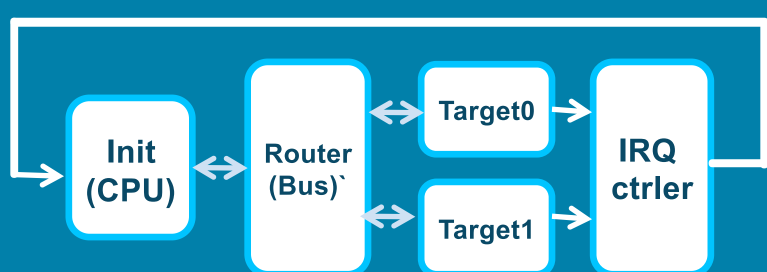
- Extra: SC Communication, InterProcess Communication
- For IPC, deep copy of data and extensions needed

Acceleration Conditions

- Sim time one core: $T_{one\_core} = T_{part0} + T_{part1}$
- Ideal case: $T_{part0} == T_{part1} \longrightarrow T_{one\_core} = 2*T_{part}$
- Sim time two cores: $T_{two\_cores} = max(T_{part0}, T_{part1}) + T_{IPC}$
- Ideal case: $T_{part0} == T_{part1} \longrightarrow T_{two\_cores} = T_{part} + T_{IPC}$
- Acceleration condition:

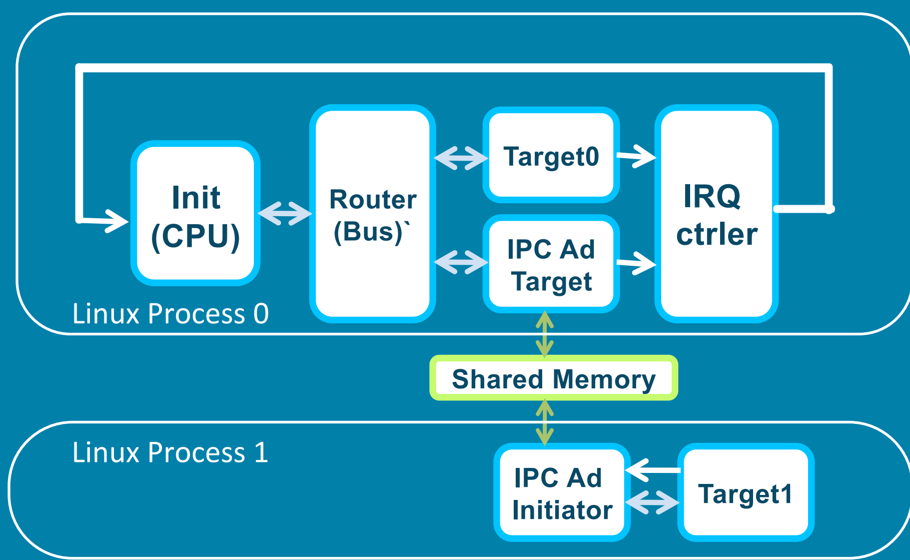
  $T_{two\_cores} < T_{one\_core} \longrightarrow T_{IPC} < T_{part}$

Single Process Simulation



- One Linux process
- TLM2 blocking interface: ⇔
- sc_fifo : →

Dual Process Simulation

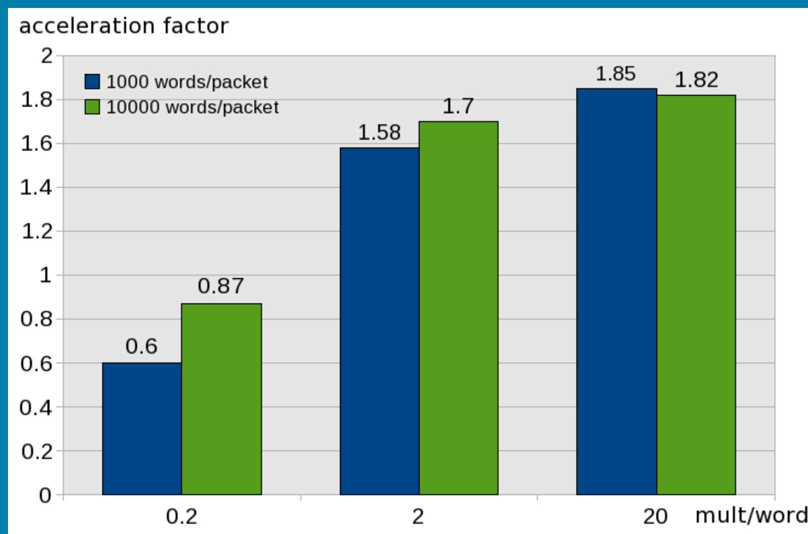


- Two Linux Processes, each running on its own core

Experimental Results

Packet Size (32 bit words)	Multiplications per word (@target)	Packets transferred	Execution time one core (seconds)	Execution time two cores (seconds)
1000	0.2	25000	0.9	1.5
1000	2	25000	6.5	4.1
1000	20	25000	61	33
10000	0.2	2500	0.75	0.65
10000	2	2500	6.3	3.7
10000	20	2500	62	34

Experimental Results



IPC Adapter

class TargetSharedMemory

```
uint32_t irq_vector; // shared memory location where the interrupt vector is stored
bool     is_irq_consumed; // true if the Target Adapter did not consume the irq_vector
boost::interprocess::interprocess_condition  cond_irq_consumed;
boost::interprocess::interprocess_condition  cond_irq_sync;
boost::interprocess::interprocess_mutex      mutex_irq;
tlm::tlm_generic_payload    tlm_payload_rd; //tlm2 payload used for reading
uint32_t buff_rd[BUFF_SIZE]; // the buffer for read data
...
```

IPC Adapter

Building the shared memory structure

mp_shared_mem = new shared_memory_object(create_only, "ipc_adapter_shared_memory", read_write);

mp_shared_mem->truncate(sizeof(TargetSharedMemory)); //set size

mp_mapped_region = new mapped_region(*mp_shared_mem, read_write); //map the whole shared memory in this process

void* addr = mp_mapped_region->get_address(); //address of the mapped region

mp_target_sm = new (addr) TargetSharedMemory; //construct the shared structure in memory

IPC Adapter - Process 1

void InitiatorAdapter::IRQThread()

```
while(1)
    uint32_t irq_vector = m_irq.read(); //blocking read
    scoped_lock<i_m> lock(mp_target_sm-> mutex_irq);
    mp_target_sm->irq_vector = irq_vector;
    mp_target_sm->is_irq_consumed = false;
    mp_target_sm->cond_irq_consumed.wait(lock);
    mp_target_sm->cond_irq_sync.notify_one();
```

i_m = interprocess_mutex

IPC Adapter - Process 0

```
void TargetAdapter::IRQThread()
    while(1)
        wait(1, SC_NS);
        scoped_lock<i_m> lock(mp_target_sm->mutex_irq);
        if (!mp_target_sm->is_irq_consumed)
            mv_irq[0]->write(mp_target_sm->irq_vector);
            mp_target_sm->is_irq_consumed = true;
            mp_target_sm->cond_irq_consumed.notify_one();
            mp_target_sm->cond_irq_sync.wait(lock);
```

i_m = interprocess_mutex

References

- IEEE 1666TM Standard System C Language Reference Manual, 2006
- OSCI TLM-2.0 Language Reference Manual, 2009
- GAZTANAGA, I., Boost.Interprocess, http://www.boost.org/doc/libs/1_43_0/doc/html/interprocess.html
- DREPPER, U., What Every Programmer Should Know About Memory, November 21, 2007
- EZUDHEEN, P., CHANDRAN, P., CHANDRA, J., SIMON, B. AND RAVI, D. 2009, Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines.
- LOVE, R., Linux System Programming, O’Reilly Media, Inc., September 18, 2007

Conclusions

- Parallelizing a SystemC application with IPC Adapters significantly improves the runtime performance
- Boost Interprocess library provides the tools needed for implementing the IPC Adapters
- Only the top netlist of the SystemC application needs modification

Thank you!

Cicerone Mihalache
Kotys LLC, Santa Clara, CA
www.kotys.biz
cicerone.mihalache@kotys.biz