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: $T_{\text{one\_core}}$

Typical SystemC Simulation

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

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

Parallel SystemC Simulation

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

- Sim time one core: \( T_{\text{one\_core}} = T_{\text{part0}} + T_{\text{part1}} \)
- Ideal case: \( T_{\text{part0}} = T_{\text{part1}} \) \( \Rightarrow \) \( T_{\text{one\_core}} = 2 \times T_{\text{part}} \)
- Sim time two cores: \( T_{\text{two\_cores}} = \max(T_{\text{part0}}, T_{\text{part1}}) + T_{\text{IPC}} \)
- Ideal case: \( T_{\text{part0}} = T_{\text{part1}} \) \( \Rightarrow \) \( T_{\text{two\_cores}} = T_{\text{part}} + T_{\text{IPC}} \)
- Acceleration condition:
  \[ T_{\text{two\_cores}} < T_{\text{one\_core}} \quad \Rightarrow \quad T_{\text{IPC}} < T_{\text{part}} \]

Single Process Simulation

- One Linux process
- TLM2 blocking interface:
  - \text{sc\_fifo}:
    - Init (CPU) \leftrightarrow Router (Bus) \leftrightarrow Target0 \leftrightarrow Target1 \leftrightarrow IRQ ctrlr
    - \text{One Linux process}
    - \text{TLM2 blocking interface:} \leftrightarrow
    - \text{sc\_fifo:} \rightarrow
Dual Process Simulation

Linux Process 0
- Init (CPU)
- Router (Bus)
- Target0
- IPC Ad Target
- IRQ ctrlr

Linux Process 1
- IPC Ad Initiator
- Target1

Shared Memory

• Two Linux Processes, each running on its own core

Experimental Results

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

// 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
void TargetAdapter::IRQThread()
    while(1)
        wait(1, SC_NS);
        scoped_lock<interprocess_mutex> 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);

References

• OSCI TLM-2.0 Language Reference Manual, 2009
• DREPPER, U., What Every Programmer Should Know About Memory, November 21, 2007
• 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