A Generic Language for Hardware & Software, Are We There Yet?

An Explorative Case Study Examining the Usage of SystemC for Multicore Programming

Sushil Menon¹ & Suryaprasad Jayadevappa, Ph.D.²

¹University of Pennsylvania (Email: sushil.menon.1988@gmail.com)
²Director, Center for Electronic System Level Design & Verification, PES School of Engineering (Email: surya@pes.edu)
Overview

- Motivating the Multicore Revolution
- Implication to the Software Industry
- Current Software Development Trends
- Problem Definition and Goals
- Experimental Procedure and Analysis
- Conclusion
Motivating the Multicore Revolution

- Although Moore’s law has promoted advances in computing infrastructure….
  - Through the exploration of micro-architectural optimizations that exploit Instruction-Level Parallelism (ILP)
  - Through a reduction in transistor geometries which sustain rising machine clock-speeds

- …designers are constrained by the current energy and performance requirements…
  - Diminishing returns from acceleration of sequential code by exploiting ILP
  - Power-inefficiency of exorbitant machine-clock speeds

- …thereby resulting in the Multicore Revolution
  - Designers pressurized to channelize efforts into exploitation of Thread-Level Parallelism (TLP)
Implication to the Software Industry

- Software developers coerced to think “parallel”
  - Identify valid partitioning of algorithmic tasks…. (not necessarily a trivial decomposition)
  - ….and map each independent sub-task to a concurrent processing element.
- Complicates (an already complicated) software-debugging process
  - Unsynchronized executions of concurrent processes…. 
  - ….generate inconceivable race-conditions…. (sometimes hard to replicate)
  - ….resulting in inexplicable and erroneous program behaviour
Current Software Development Trends

- The software industry has channelized efforts into developing standardizations...
  - Define models for concurrent computation
  - Invention of open-source platforms and tool-chains. Ex: OpenMP
- .... which enable multicore software development over multiple computing platforms.
- However, concurrency is a property inherent to electronic hardware.
- This presents a new possibility....
- ....of utilizing system-level modeling platforms (Ex: SystemC) to develop multicore software.
  - Support for modeling concurrency
SystemC, Concurrency….Really?

- SystemC implements the *co-operative multitasking* compute-model.…
- ….which provides the ability to “model” concurrency (i.e. not truly concurrent) using multiple “cooperative” processes.…
- ….which are processes that voluntarily relinquish simulator control to allow execution of others.
- Concurrency is captured by voluntary relinquishment of a process without advancing simulation time.…
  - ….this permits the execution of other processes in the same simulated time-space.

The Cooperative-Multitasking Compute-Model implemented by the SystemC simulation kernel [10]
Problem Definition

- Now that we know SystemC enables the modeling of concurrency….
- ….”Can we utilize SystemC for multicore programming?”
- Specifically, can we use SystemC to….
  - ....implement concurrent software?
  - ....verify concurrent software?
  - ....analyze software performance?
- And if so, what are the drawbacks….  
  - ....in terms of true performance? (because it is pseudo-concurrent)
  - ....in terms of complexity?
- Methodology of research
  - Performing an explorative case study…. (more on next slide)
Explorative Case Study – Concurrent Quicksort

- Explore the process of modeling and implementing concurrent “Quicksort”....
  - Easy to parallelize
  - Illustrates properties (synchronization requirements) that resemble typical multicore software
- ....using both SystemC and OpenMP.
- Comparison Benchmarks:
  - SystemC concurrent variant
  - OpenMP concurrent variant
  - Uniprocessor-based sequential variant
- Comparison Metrics:
  - Speedup (SystemC simulated timestamps + wall-clock)
  - Code Complexity (# of LoC)
Modeling Concurrent Quicksort – 1

- Comparison based, divide-and-conquer sorting algorithm.
- ....which partitions the parent-list into two smaller sub-lists.
- ....each of which are then recursively partitioned.
- Partitioning is accomplished by selecting a "pivot" element.
- ....and sorting the list such that...
  - if element[i] ≤ element[pivot], then element[i] appears before element[pivot], and
  - if element[i] > element[pivot], then element[i] appears after element[pivot]
- ....thus resulting in appropriate placement of pivot element.
- Observe that there is no interaction between the sub-lists.
- ....hence resulting in efficient algorithmic parallelization (since each sub-list can be partitioned concurrently).

<table>
<thead>
<tr>
<th>Elements &lt; pivot</th>
<th>pivot</th>
<th>Elements &gt; pivot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ..................(p-1)</td>
<td>p</td>
<td>(p+1).................(n-1)</td>
</tr>
</tbody>
</table>

Recursively Sorted in Parallel
Modeling Concurrent Quicksort – 2

- Since neither SystemC nor OpenMP provide constructs for parallelization of recursive algorithms.
- We transform the recursive algorithm into an iterative one using a “work-stack” data-structure.
- ...which stores the left and right array-indices of unsorted sub-lists.
- Since the work-stack is shared amongst multiple concurrent threads.
- ...the access methods are guarded with a “mutex” primitive to prevent race-conditions.

Diagrammatic Representation of Iterative Quicksort

(1) Pop List-Indices

(2) Partition List to produce sub-lists

(3) Push sub-list Indices

Work-Stack

...........

(p+1), (n-1)

0, (p-1)
Modeling Concurrent Quicksort – 3

- Algorithm modeled around “Master-Slave” design pattern
- The master thread:
  - ...retrieves array indices of an unsorted sub-list from the work-stack ...
  - ...signals the slave thread to partition the respective sub-list and waits for it to return...
  - ...posts array indices of the new un-partitioned sub-lists to the work-stack.
- The slave thread:
  - ...on receiving the notification, partitions the specified sub-list...
  - ...on completion, signals the master thread.
SystemC Master Thread Implementation

void MasterCode ( int tid )
{
    while (1)
    {
        /* If work-stack has new work....*/
        if ( stack.empty() == FALSE )
        {
            stack_mutex . lock();
            /* ....get new work from work-stack */
            stack_mutex . unlock();
        }
        else { /* Else, wait for new work */
            wait ( new_work ); continue;
        }
        /* If sub-list indices are valid....*/
        if ( l < r )
        {
            /* ....notify slave to sort and wait */
            begin_sort[tid] . notify ( SC_ZERO_TIME );
            wait ( sort_done[tid] );
        }
        /* Modeling concurrency */
        wait (PROCESS_TIME, SC_NS);
        stack_mutex . lock();
        /* Post work on work-stack */
        new_work . notify ( SC_ZERO_TIME );
        wait ( new_work );
        }  
    }

(1) Compete for mutually-exclusive access to stack and pop sort-indices
(2) Generate “begin sort” event and self-suspend
(3) Self-suspend for arbitrary time to model concurrency
(4) Push unsorted sub-list indices onto work-stack
(5) Generate “new work” event and self-suspend
SystemC Slave Thread Implementation

```c
void SlaveCode ( int tid )
{
   /* Partition the sub-list */
   int pivot_pos = Partition (l, r);

   /* Notify master of sort completion */
   sort_done[tid] . Notify ( SC_ZERO_TIME );
}
```

1. Perform Partition Operation
2. Generate “sort done” event
Experimental Evaluation

- Experiments were performed on a system comprising of:
  - A dual-core, hyper-threaded Intel Core i3 processor clocked at 2.27GHz….
  - 3GB DDR3-RAM clocked at 1067MHz….
  - Running Ubuntu Linux 10.04 which implements Linux Kernel 2.6.32.32-generic

- All programs compiled using gcc-4.4.3 compiler, with flags –O3 and –fopenmp

- Optimized build of OSCI SystemC-2.2.05 library, using “quickthreads” threading library

- Experimental Procedure:
  - 100 simulation-runs of each variant (including multiple threads for concurrent variants)….
  - Logarithmic (log10) array-sizes in the range [10, 100-million]….
  - Arithmetic Mean of wall-clock execution times (for true run-times)
Results – 1. Modeling Concurrency

\[ \text{Speedup}_{N-\text{threads}} = \frac{\text{SystemC Timestamp obtained while using 1 SC_Thread}}{\text{SystemC Timestamp obtained while using N SC_Threads}} \]

Speedup for N-threads saturates to \(|N|\). Indicates successful modeling of concurrency.
void MasterCode ( int tid )
{
    while (1)
    {
        /* If work-stack has new work....*/
        if ( stack.empty() == FALSE )
        {
            stack_mutex . lock();
            /* ....get new work from work-stack */
            stack_mutex . unlock();
        } else
        {
            /* Else, wait for new work */
            wait ( new_work ); continue;
        }

        /* If sub-list indices are valid....*/
        if ( l < r )
        {
            /* ....notify slave to sort and wait */
            begin_sort[tid] . notify ( SC_ZERO_TIME );
            wait ( sort_done[tid] );

            /* Modeling concurrency */
            wait (PROCESS_TIME, SC_NS);

            stack_mutex . lock();
            /* Post work on work-stack */
            stack_mutex . unlock();
        } else { /* Else, wait for new work */
            wait ( new_work ); continue;
        }
    }
}
All threads fail to acquire lock. Thus, able to easily reproduce deadlocks and other debug-scenarios!
Results – 3. True Performance

True Speedup for SystemC is always below 1.0. This is attributed to the sequential nature of the Cooperative-Multitasking compute-model.
Results – 4. Code Complexity

Defined as the # of LoC (excluding comments and debug-messages)

- **Sequential OpenMP SystemC**
  - Number of Lines-of-Code (# of LoC): 110

- **Work-stack + Synchronization code**
  - Number of Lines-of-Code (# of LoC): 213

- **Work-stack + Synchronization code + Simulator code**
  - Number of Lines-of-Code (# of LoC): 269
Conclusion

- Advantages of using SystemC for multicore programming
  - Easy to model concurrency
  - Easy to reproduce multicore verification scenarios
  - Models can be extended to incorporate performance analysis
    - ....through parameterizable wait-times
    - ....think of assigning mathematical distributions to the wait-times

- Disadvantages
  - Performance issues (low true speedup, pseudo-concurrent only!)
  - Learning curve – need to understand simulator architecture

- Future Work
  - Extension of software models for performance analysis
  - Possibly, real-time acceleration of SystemC using multicore hardware (a different problem domain altogether)
  - Reproduce results in works such as [16], [17], [18]
References – 1


References – 2

Thank You! 😊

Questions & Concerns?