Passive TLM

Paul Klein

Zafer Kadi, Ph.D.

Intel Corporation

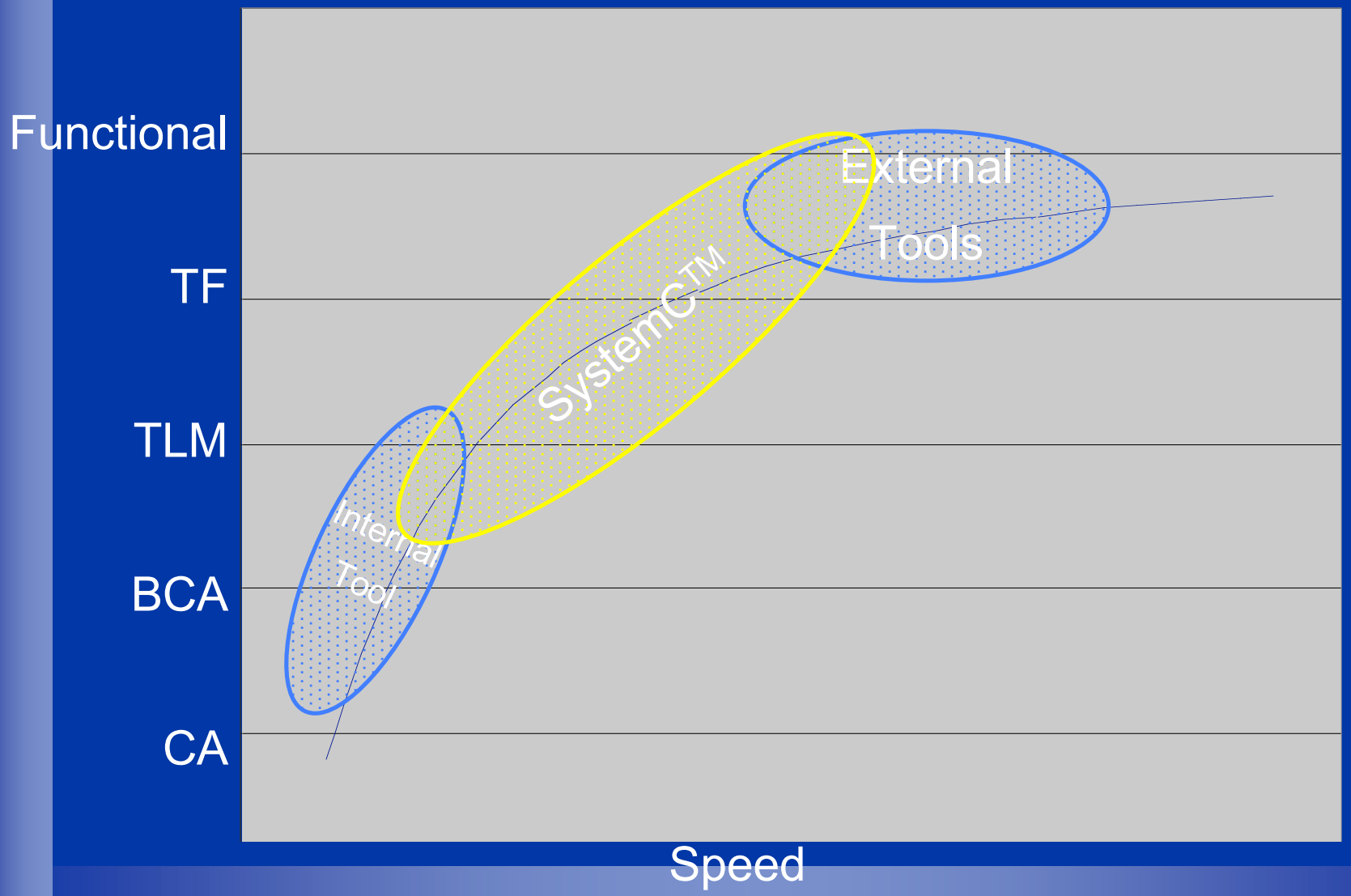
June 7th, 2004

E-mail: Paul.J.Klein@intel.com
Zafer.Kadi@intel.com

Introduction

- Introduction
- Past Tool Experience
- Present Utilization of SystemC
- Passive TLM
  - Studies with SytemC
  - Passive TLM
  - Performance Optimization
- Conclusions

History


Functional
TF
TLM
BCA
CA
External
Tools
SystemC™
Internal
Tool
Speed

SystemC Project(s) Status

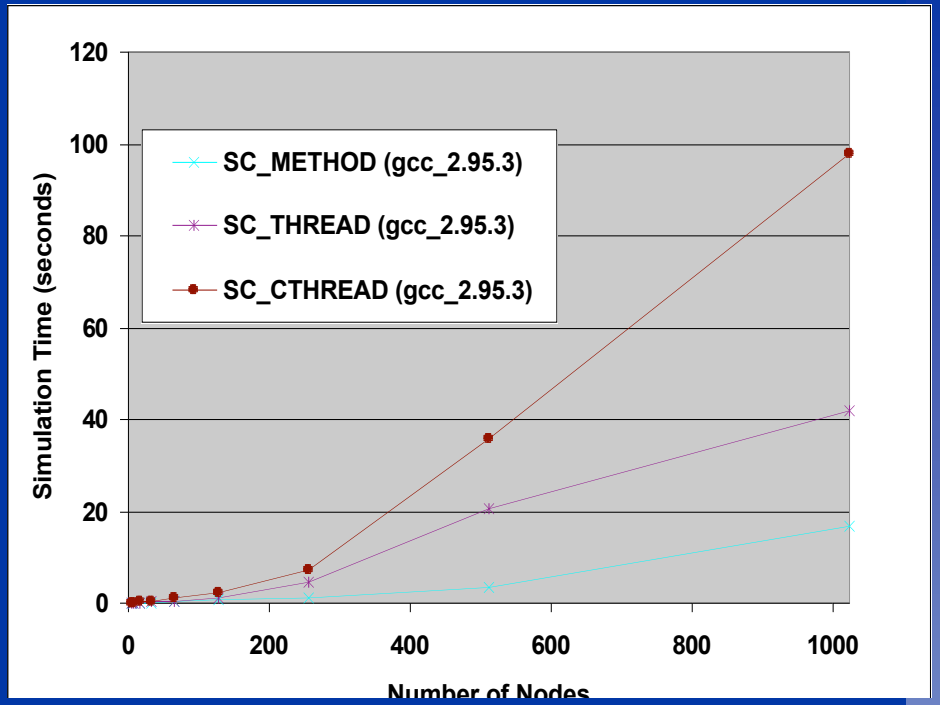
- Utilize SystemC in our SoC efforts
- Model at CCA level
- Internal TLM (in two groups)
- Utilizing SystemC Models
  - Architectural Explorations
  - SW development
- Co Simulation with RTL
  - Verification
  - Tuning

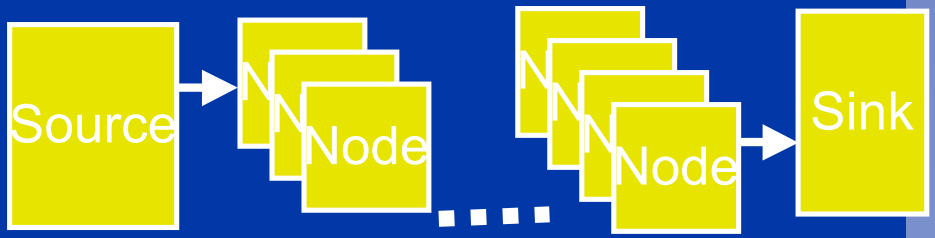
SystemC Experience and Development

- Methodologies
  - TLM or Cycle Counting Accurate
    - "Passive" components
    - "Passive" TLM
  - Modeling SW for performance
    - Memory Mgmt
    - Reduced overhead
    - Transparent usage for different levels of modeling of accuracy or speed
  - Tools
    - Compiler and Profilers
      - Intel C/C++ Compiler
      - VTune
    - Optimized SystemC™ Scheduler

SC_METHOD and Others

- Utilize SC_METHOD
  - Reduced overhead
  - Forces:
    - Initial Planning
    - Modularity of model designs
    - Reduced complexity in larger models
  - Experience
    - Many SC_THREAD based state machines can be converted to SC_METHOD calls





Goals of the TLM we developed

- High-performance TLM and supporting infrastructure for SystemC
  - Passive design
    - Virtual clocking
  - Focus on HW interfaces
    - Although, templates could support SW communications
  - Efficient and automatic memory management
  - Ability to boot real operating systems with detailed timing at high speeds
- Relatively simple API
  - Single SystemC interface (multiple models of operation)
  - No new/delete operations related to TLM API
  - Arbitration interface
    - Custom algorithms can be developed by implementing interface
- Configurable “Protocol” instead of multiple interfaces
- Relocatable and “Bus Portable” Components
  - Target port interface configurable to locate memory-mapped devices
  - Component timing limited to Wait states, all protocol timing handled within channel
- 100% compatible with OSCI reference simulation kernel
  - Easy support for 3rd party simulation environments and their proprietary capabilities and APIs

Passive TLM

- Single configurable channel (bus)
- Passive Design (no clocks)
- Automated Memory Pooling of transaction
- Master, Slave, and Master&Slave ports
- Implicit and Explicit Timing modes
- Point-to-Point and Multi-Drop interconnects
- Protocol Configuration
  - Split Completion
  - Concurrent R/W
  - Arbitration configuration/selection
  - DDR Timing
  - Timing constraints
- No threading overhead

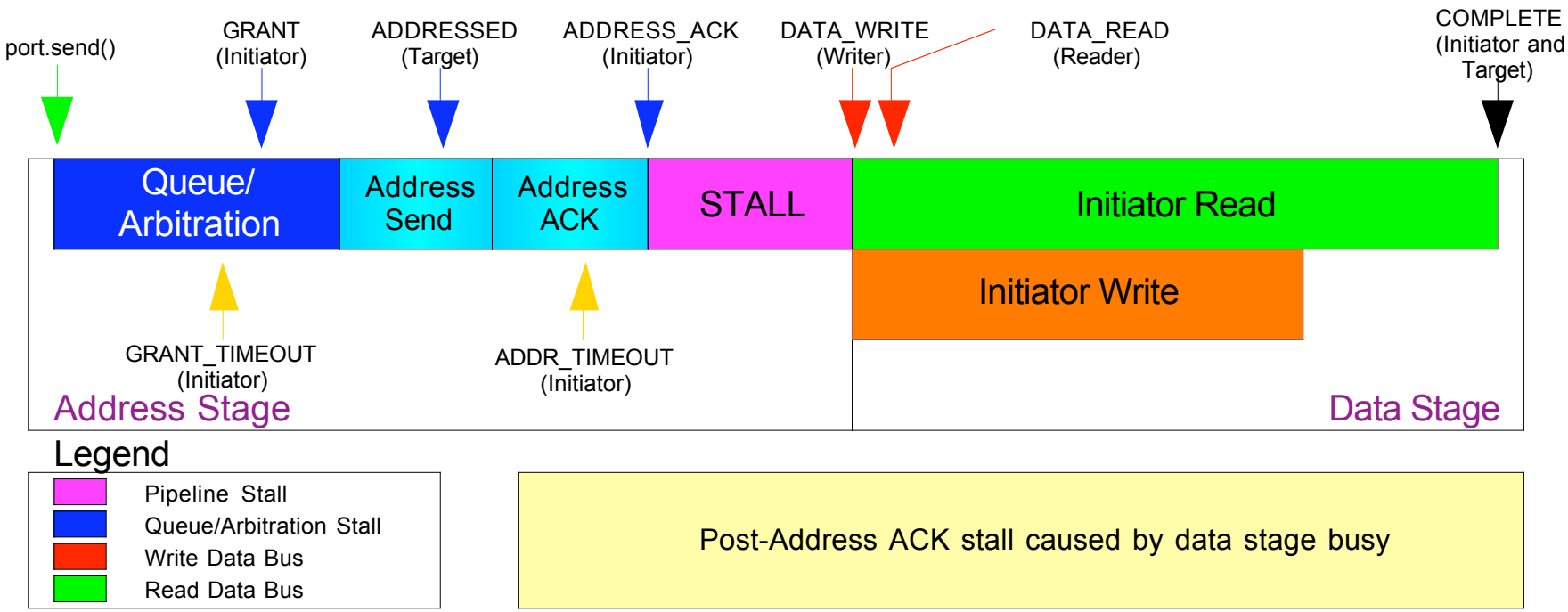
SW Methodologies

- Memory Pooling of Transactions
  - new/delete is virtually eliminated
    - Unless utilized by a component model behavior implementation)
  - Run-time configurable
    - Command line options used to configure memory management kernel
  - Automated
    - The memory pools used are automated via reference counting, therefore there is no need for any component model to “release” objects
  - Pass by Pointer and/or Reference
    - API attempts to be as consistent as possible
- Transparent interfaces to user
- Ability to Integrate Other Tools
  - Integrated Analysis Tools
    - Internal
    - From a 3rd party vendor
- Co – Simulation with RTL

Passive TLM and Components

- Passive:
  - The models are running only when utilized
  - Based on SC_METHOD
  - Event Driven/Triggered
  - Clock cycle accuracy can be implemented with additional logic at components and not the TLM

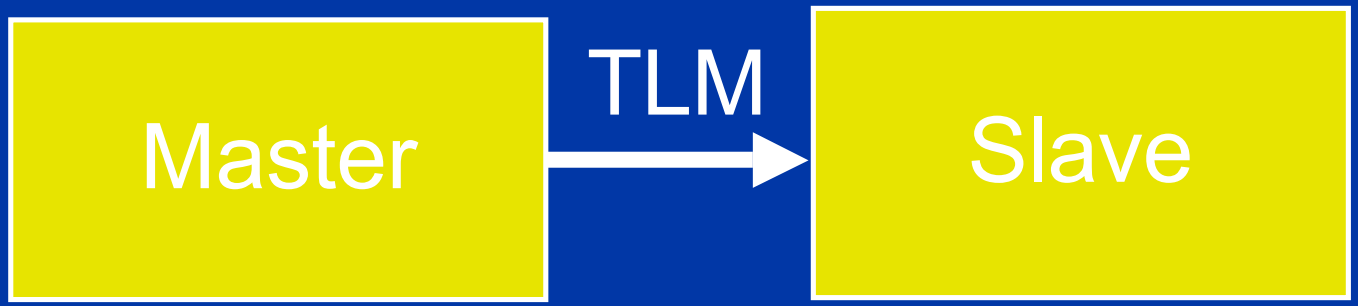
Protocol Timing



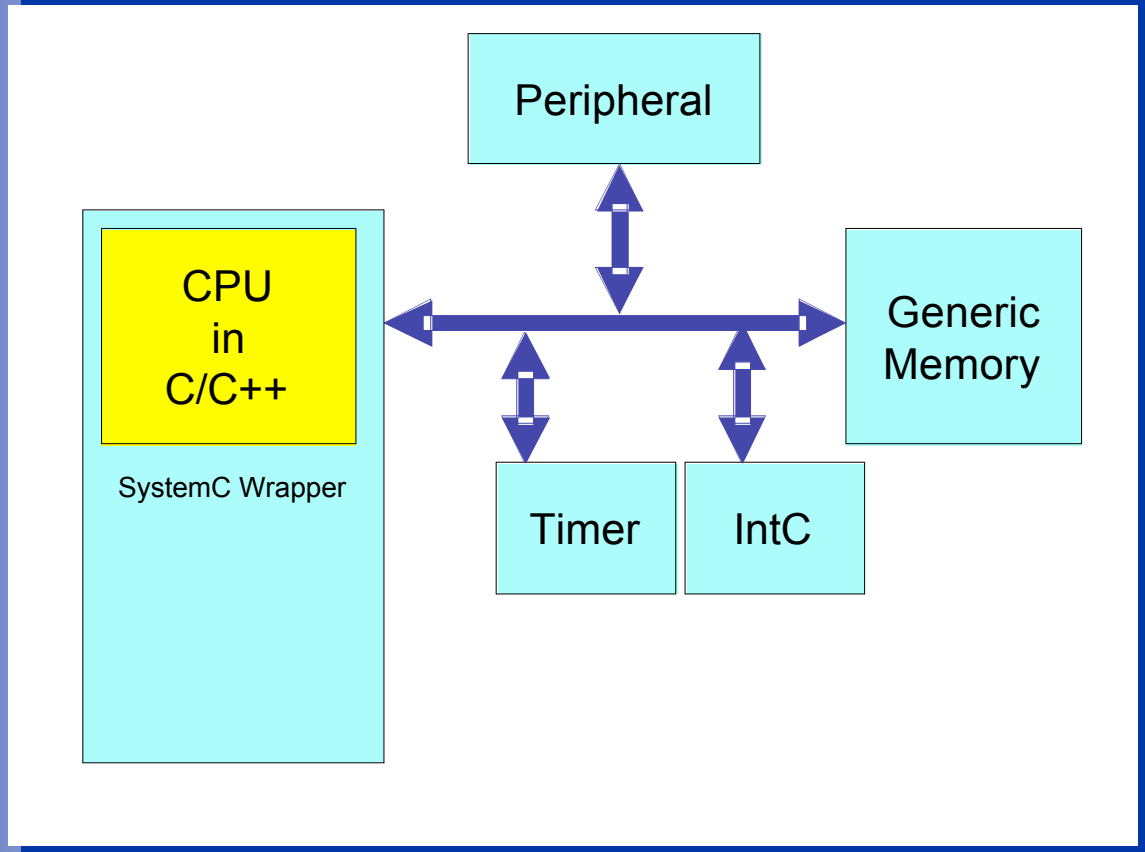
The diagram displays a generic protocol and parameter controlled by the modeler

Results

- The optimized results for the “passive” TLM
- When constantly utilized:
  - ~ 5 MCPS for implicit
  - ~ 2 MCPS for explicit



Example System Model



Ability to

Boot OS

@ 4-12 MCPS

The Passive components are not active, unless utilized, thus the system can run more closely at the speed of the CPU model

Tools

- Intel Compilers ICC™ and VTune™ were utilized to
  - Optimized up to 25%
    - Report with performance improvement numbers published to be published on http://intel.com/cd/ids/developer/asmo-na/eng/97007.htm
  - VTune™
    - Optimization of heavily used areas
- Coherent Build Env. as with any SW development is a must

Conclusion

- For effective and high performance modeling
  - Modeling Methodology play great role
  - Well defined TLM is a must
    - The right trade of speed vs accuracy at the right components
  - A pool of generic and well optimized components
- ICC™ compiler will give boost of performance with no additional overhead
  - It can link with GNU C/C++ builds too.

Q/A