Performance Modeling for Early Analysis of Multi-Core Systems

Reinaldo Bergamaschi[1], Indira Nair[1], Gero Dittmann[1],
Hiren Patel[3], Geert Janssen[1], Nagu Dhanwada[4], Alper Buyuktosunoglu[1],
Emrah Acar[5], Gi-Joon Nam[5], Guoling Han[2], Dorothy Kucar[1], Pradip Bose[1], John Darringer[1]

[1] IBM T. J. Watson Research Center, Yorktown Heights, NY 10598; [2] Univ. of California, Los Angeles, CA 90095;
[3] Virginia Tech, Blacksburg, VA 24060; [4] IBM EDA, East Fishkill, NY 12533, [5] IBM Austin Research, Austin, TX 78758
berga@us.ibm.com

ABSTRACT

Performance analysis of microprocessors is a critical step in defining the microarchitecture, prior to register-transfer-level (RTL) design. In complex chip multiprocessor systems, including multiple cores, caches and busses, this problem is compounded by complex performance interactions between cores, caches and interconnections, as well as by tight interdependencies between performance, power and physical characteristics of the design (i.e., floorplan). Although there are many point tools for the analysis of performance, or power, or floorplan of complex systems-on-chip (SoCs), there are surprisingly few works on an integrated tool that is capable of analyzing these various system characteristics simultaneously and allow the user to explore different design configurations and their effect on performance, power, size and thermal aspects.

This paper describes an integrated tool for early analysis of performance, power, physical and thermal characteristics of multi-core systems. It includes cycle-accurate, transaction-level SystemC-based performance models of POWER processors and system components (i.e., caches, buses). Power models, for power computation, physical models for floorplanning and packaging models for thermal analysis are also included. The tool allows the user to build different systems by selecting components from a library and connecting them together in a visual environment. Using these models, users can simulate and dynamically analyze the performance, power and thermal aspects of multi-core systems.

Categories and Subject Descriptors

C.0 [**General**]: *Modeling of computer architecture, system architectures, systems specification methodology*.

General Terms

Algorithms, Performance, Design, Experimentation.

Keywords

Performance, power and physical analysis, transaction-level modeling, multi-core systems modeling, early analysis.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CODES+ISSS'07, September 30 – October 3, 2007, Salzburg, Austria.
Copyright 2007 ACM 978-1-59593-824-4/07/0009…$5.00.

1. INTRODUCTION

Advanced microprocessor design methodologies rely heavily on early performance and power analysis for microarchitecture trade-offs and tuning. Simulation-based methods using execution-driven or trace-driven models are commonly used. In order to obtain a reasonable degree of accuracy, critical for detailed trade-off analysis, cycle-accurate models of the internal pipelines of the processors, as well as communication delays between components are needed. The communication delays between components include a functional part and a physical part. The functional delay depends on the specific communication protocols used. The physical delay is related to the number of cycles needed to transfer data across the length of the interconnections, which depends on the relative positioning of the components (i.e., floorplan), the technology and buffering capabilities. As components get larger, the physical delays increase and must be taken into account in the models.

Several microprocessor performance analysis tools have been developed over the years for various purposes. These fall in two main types, trace-driven timing simulators [1][2] and execution-driven simulators [3][4]. Both of these have advantages and disadvantages regarding simulation speed and the ability to model certain architectural details, such as branches and speculative execution, and the ability to execute actual software versus instruction traces. Power models and tools, which use statistics generated by performance simulators, have also been developed [5][6].

While these tools have been successfully applied to a variety of processors and systems, they lack the modularity and componentization required for quick design exploration. Moreover, they do not offer an integrated environment for analyzing performance, power, floorplan and thermal aspects.

This paper presents the models and tools supporting an integrated approach to early design analysis for multi-core systems, which were implemented in a tool called SLATE (System-Level Analysis Tool for Early Exploration). This paper gives an overview of the system and a detailed description of the performance models.

This paper is organized in the following way. Section 2 presents an overview of SLATE and the early design methodology it supports. Section 3 describes the SystemC-based performance modeling approach applied to the SLATE components. Section 4 presents the experimental results and Section 5 offers conclusions.

2. SLATE OVERVIEW

The models, tools and environment implemented in SLATE support the following early design analysis methodology:

1. Abstract system design structural specification using a graphical block diagram input of main system components (i.e., cores, caches, buses, memory controller). Blocks are represented with relative sizes, and user controls relative positioning of the blocks.
2. Early floorplan is generated from initial user placement of major blocks and global interconnections estimated. Global delays are estimated based on buffering assumptions and target cycle time.
3. Top-level system performance model is automatically generated by assembling and linking performance models for individual components, stored in a library. Global latencies (from interconnection delays) are passed in as parameters to the model.
4. Trace-driven system performance simulation is executed, producing user-defined graphs and statistics, helping the user analyze bottlenecks and perform architectural trade-offs.
5. Statistics collected from performance simulation are used to drive power computation and analysis for each component in the design. The power model is integrated in the performance model; power is computed on-the-fly during simulation.
6. Power values are back-annotated into the floorplan model as simulation progresses. The power-annotated floorplan is then used to drive a thermal analysis tool to generate thermal plots and analyze hot-spots.
7. Finally, the designer armed with information regarding performance, power, and thermal characteristics, is able to change the design configuration (e.g., topology, parameters) and iterate until an acceptable design point is reached.

The order in which these steps above are performed may change, depending on the type of analysis desired. All steps can be automated in the form of scripts and run in batch mode.

The target user of SLATE is the system architect who typically is an expert on systems analysis and trade-offs, but is not necessarily an expert in programming or physical design. Hence, SLATE was designed to make it very simple to enter, configure and simulate a design, based on pre-defined components stored in a library. This is accomplished either through a visual environment (i.e. a block-diagram editor) or through simple scripts. SLATE library components are designed to have very few pins (i.e., transaction-level ports in the SystemC specification). The number of ports of each component is kept very small by design (usually 1 to 3 ports, except for the bus) to make it easy to connect components. Section 3.1 gives more details about the components ports and interfaces.

The library components available to the user are major design blocks, such as, processor cores, caches, memory controllers and buses. A component in the library is described by an XML file which contains: area, aspect ratio, reference to an executable transaction-level model binary (compiled from its SystemC model), list of transaction-level model ports and locations. If the same component is available with different area, aspect ratio or port positions, a new XML file is created. At an early stage, accurate areas, and aspect ratios for components may not be available, but approximate figures, usually derived from previous designs or designer experience, are sufficient to drive the analysis and provide valuable feedback to the user.

The user composes the system by dragging-and-dropping components onto a block-diagram editor and by connecting them appropriately. The block-diagram editor stores the composed netlist in OpenAccess (OA [7]) which has native support for various physical properties, such as area and pin locations. The image on the block editor displays the components with their relatives sizes and aspect ratios, thus allowing the designer to explore different early floorplans for the system. Different floorplans using the same functional components with different areas and aspect ratios are likely to exhibit different power densities, thermal characteristics (i.e., hot spots), and interconnect delays.

3. PERFORMANCE MODELING BUILDING BLOCKS

Performance models, as opposed to functional models, do not carry out actual computations, but propagate the delays involved in the computation, producing, as a result, metrics such as latency and throughput (but not the final value of the computed data). Although data computation is not taking place, all side effects must be fully taken into account. For example, in a cache model, when a write operation is executed, the corresponding address in the cache is marked as taken, although, there is no specific data value stored. Similarly, queues inside a microprocessor need to be modeled correctly (so that pipelines can be stalled when queues are full), the correct number of read/write ports in register files need to be modeled, etc.

SystemC-based transaction-level models (TLM) were used, for their clear way of separating computation from communication using channels and interfaces.

SLATE provides a hierarchical library of performance models which can be easily connected to form complex components, ranging from simple pipelined execution units, to caches, to complete cores. At the lowest level, there are two basic types of components, namely a *Computation* module and a *Delay* channel. These components are used by the model designer to build more complex cycle-accurate models.

The *Computation* module is a shell for a generic atomic processing element which gets executed every clock cycle, triggered by the positive edge of the clock. This module needs to be filled in with the proper behavior on a case-by-case basis. SLATE provides several such modules for microprocessor building blocks, such as Instruction/Data caches, execution units, issue queues, etc. The interfaces to/from this module are ports connecting to *Delay* channels. All data transfers between *Computation* modules go through *Delay* channels.

Three types of *Delay* channels have been implemented, namely: *Pipeline*, *Delayed_Queue*, and *Async_Fifo*. These channels support the typical TLM interfaces (e.g., write(), can_write(), read(), can_read()), as well as other dedicated functions.

The *Pipeline* channel implements the equivalent of a hardware pipeline with templated data-type and parameterized length and bandwidth. One data element may be written or read per cycle, and the pipeline can be stalled. Stalling prevents new data from

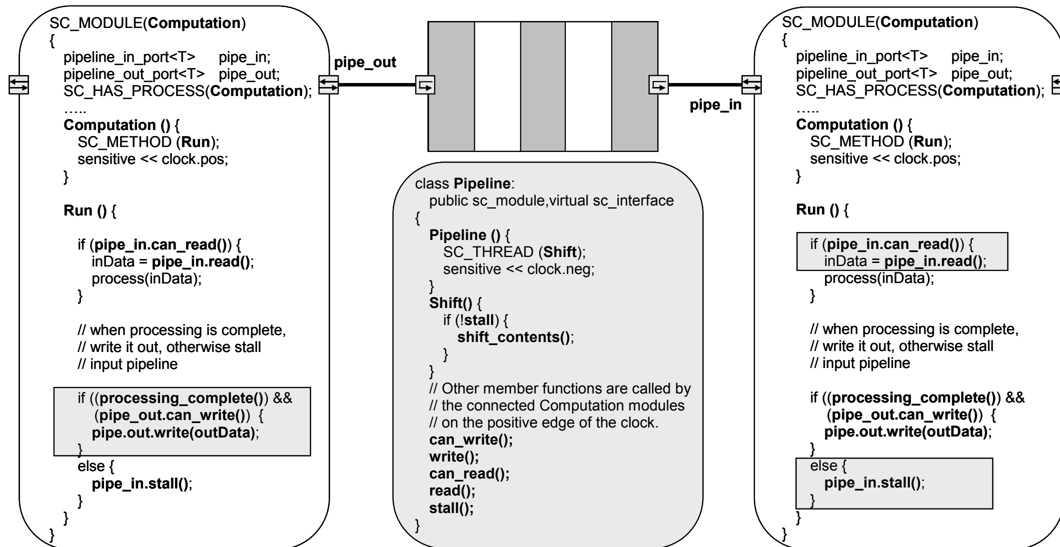

Figure 1: Basic performance modeling components in Slate (Computation module and Pipeline Delay Channel)

being written and valid data (and the end of the pipeline) from being dropped.

The *Delayed_Queue* channel implements a queue with a maximum size and delay parameters. One or more data elements may be written or read per cycle, but once an element is written, it will only be available for reading after a predefined delay. This channel cannot be stalled, but can become full and block new data from being written to. It can also be parameterized to work like a synchronous fifo.

The *Async_Fifo* channel implements fifo communication asynchronously between two components operating at different frequencies. This is usually needed for system modeling when cores connect to caches or buses at different frequencies, and when dynamic frequency scaling is applied on a component by component basis for power management purposes.

Figure 1 shows an example pseudo-code of how two *Computation* modules can be connected using a *Pipeline* delay channel. The *Computation* module has a method **Run()** which is executed every clock cycle. At every cycle, it checks if there is new data in the input pipe (pipe_in.can_read()) to be read and processed. If processing the data is completed and the output pipe is not full or stalled, it then writes the new data into the output pipe (pipe_out.write()). If processing is not completed or it cannot write, then it must prevent the input pipeline from shifting and dropping the current data, by stalling the input pipe (pipe_in.stall()). The model designer must write the code for the data processing only. The *Pipeline* channel provides functions for writing to, reading from, and stalling it, as well as for checking if it can be read/written, which must be called on the positive edge of the clock. It has one internal process, triggered on the negative edge which shifts the pipeline contents if the pipeline is not being stalled. The basic scheme of performing computation on the positive edge of the clock and data shifting on the negative edge, prevents race conditions.

Computation modules and *Delay* channels are the building blocks for constructing cycle-accurate models of complex hardware components.

When connecting larger components, there may be multiple channels required for different types of communications (e.g., different data types, different latencies, etc.). Instead of exporting multiple internal ports to the boundaries of the larger component, SLATE uses a channel container to group several individual channels into one channel object. Each component needs to declare a single port of the type of the container channel, and connect to the other component, thus minimizing the number of ports that need be connected among complex blocks. As mentioned in Section 2, one of the main usability goals of SLATE is to make it very simple for designers to connect up larger blocks to form complex systems. This is achieved by having to connect only very few ports (usually one or two) per component.

A port in a complex component is of type *Channel_Container*, encapsulating one or more *Delay* channels. The number, type and parameters of each Delay channel inside a container are specific to each port. The components use the SystemC *elaboration* callbacks to create and connect the internal channels. During SystemC elaboration, the ports are connected, then during the *before_end_of_elaboration()* callback, the internal channels are created inside the container and connected appropriately. The component accesses the internal channels using channel identifiers passed in as parameters to the access functions. Figure 2 illustrates the channel container approach. The pseudo-code in Figure 2 shows the left component writing an address and data values on two different channels, and reading a control value from a third channel, all inside the channel container and through the same port. The internal channels may be used for communication in different directions. By using this container, only one port-interface-port connection is required between the two components instead of three. The container approach is more flexible and generic than simply a vectored port.

By using *Computation* modules, *Delay* channels and *Channel_Containers*, a range of cycle-accurate models can be built. The actual timing accuracy of the model may depend on the level of detail used, but the infrastructure allows for cycle-accuracy.

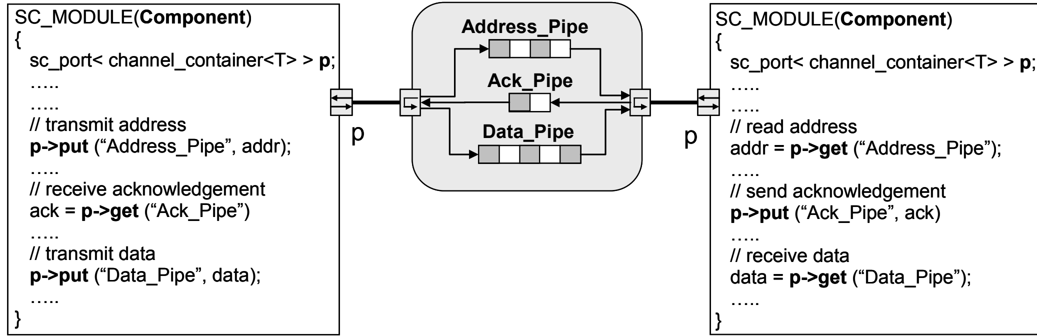

Figure 2: Example of a Channel_Container with 3 internal channels, connecting two complex components.

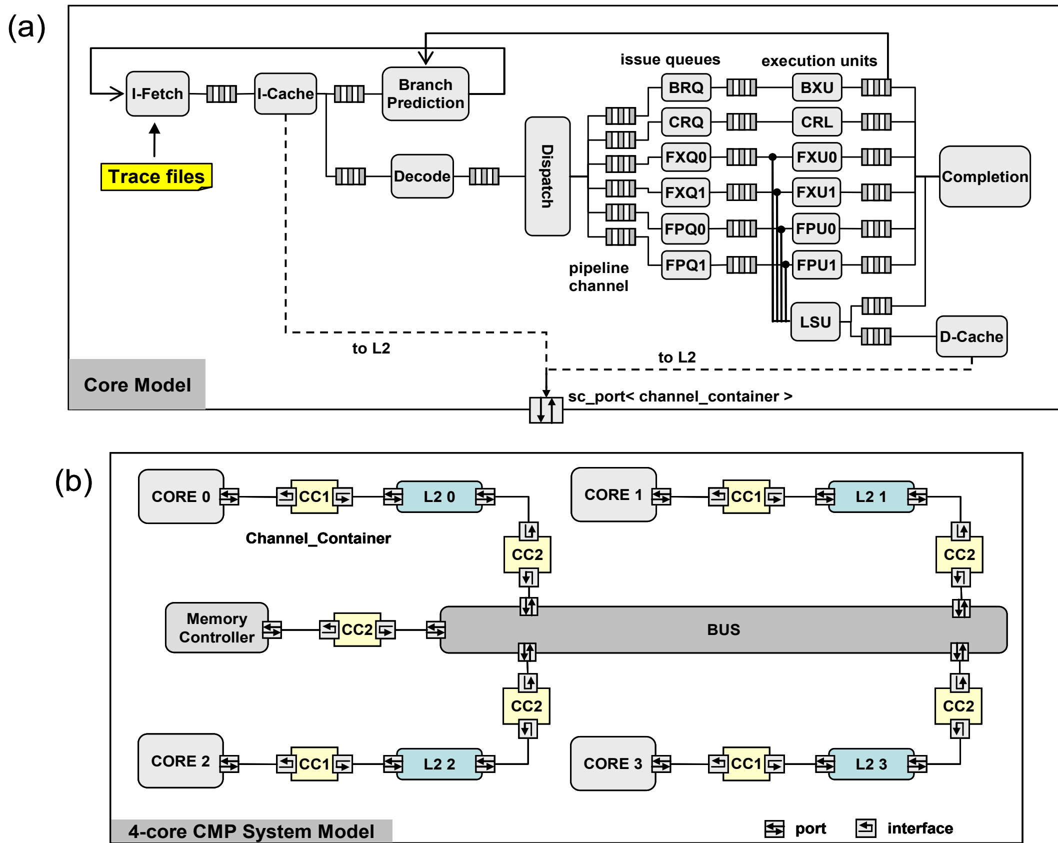

Figure 3: (a) Internal organization of core model (not all connections shown), (b) internal organization of 4-core system model (not all connections shown)

3.1 Component and System Modeling

To validate the approach we built several models based on the POWER family of processors and systems. All models are cycle-accurate performance models running instruction traces. SLATE's current components include: Core processor models, L1 and L2 cache models, bus model and a memory controller model. A wide range of parameters on the core, cache and memory models are supported. This section presents details on one core model and on the multi-core system used for validation and experiments.

3.1.1 Core Model

The core model described in this section is a pipeline-accurate performance model, based on the POWER4 processor which is a single-thread, out-of-order execution, in-order completion micro-architecture [1][8]. The behavior of the units inside the core (e.g., Decode, Dispatch, Issue queues) is modeled at a high level using *Computation* modules which execute every clock cycle. The execution delays of each unit are modeled by cycle accurate *Pipeline* channels, as illustrated in Figure 3(a). The pipeline latencies and bandwidths are parameterized to capture the delays and bit-widths of each unit. The behavior of each unit is described at a high level but it contains all relevant architectural features for

accurate performance modeling. Features such as limited-size queues, instruction grouping, register renaming, load/store queues, pipeline bypassing, correct number of read/write ports on register files, etc., are modeled accurately according to the architectural specification [8].

The core model includes internal L1 Instruction and Data caches, which communicate with an external L2 cache using a single port of *Channel_Container* type. This channel encapsulates six pipelines for read and write transfers between the L1 Instruction and Data caches and the L2 cache.

For a given family of microprocessors, evolving from one processor to the next involves changing certain algorithms (i.e., branch prediction, instruction grouping), changing the parameters to many units (i.e., L1 cache size and associativity, pipeline latencies), and changing the number of units used (i.e., use two floating-point execution units, instead of one). SLATE allows these changes to be made very quickly to a given model in order to explore different micro-architectures. SLATE also includes models for multi-threaded cores, such as POWER5 [9].

3.1.2 Multi-Core System Model

Using the POWER4 core model described in Section 3.1.1, we built a multi-core system consisting of four cores, four private L2 caches connected to a bus and a memory controller, as shown in Figure 3(b). This model is representative of chip multi-processor (CMP) systems built in the last few years.

Each core receives as input an application trace (e.g., bzip2 from SPEC CINT2000 benchmarks [10]) and processes its instructions. Whenever there is an L1 cache miss, the core model initiates a transaction on the L2 model (for accessing a given address for load or store). If the L2 cache has valid data for the given address, it is a hit and it sends a transaction back to the core model. Each of these transactions incurs multi-cycle delays captured in the pipeline channels. If the L2 does not contain valid data at the given address, then it is a miss, and it sends a transaction to the bus requesting the status of the given address. The bus then queries all other L2's in the system for the given address and it determines if it needs to retrieve the contents or invalidate them, depending on whether the operation requested was a load or store. If no L2 owns the given address, the bus passes the request to the memory controller, which then either retrieves the data/instruction or stores it in the memory model. This sequence of transactions may be initiated by any core, thus the bus model must be able to handle and arbitrate correctly among multiple requests. The proprietary coherence protocol in the bus is modeled cycle-accurately. An accurate bus model is important in a multi-core system to simulate coherency and contention to memory precisely, as these factors may affect overall performance significantly.

The bus is connected to the L2 caches and memory controller using *Channel_Container* ports encapsulating *Delay* channels that can be configured to be either synchronous or asynchronous queues/fifos. The ability to use different communication schemes allows additional architectural exploration capabilities.

The core, cache and bus models also compute the dynamic and static power dissipated by the modules as the simulation progresses. The power model used for the core is based on [5]. Performance simulation produces several statistics on the usage of the units inside the core (e.g., how many cycles the fixed point execution unit was active, or the cache miss rate).

These statistics are converted into switching factors which are then inserted into parameterized power formulas associated with each unit, resulting in the unit's average power. The process is repeated for all units inside the cores and for all other models, in order to compute the system power. The power formulas are either estimated or generated by detailed logic and circuit-level simulations of previous versions of the units (from a previous generation, properly scaled for technology changes).

The outputs of the system simulation are several metrics related to the performance and power of the system. For performance, the main metrics are: *Cycles-per-Instruction* (CPI) or its inverse *Instructions-per-Cycle* (IPC) and *Instructions-per-Second* (a measure of throughput), for a single core as well as for the whole system. SLATE also provides detailed metrics and statistics for helping the designer identify performance bottlenecks, such as cache miss rates, pipeline stalls, issue queue occupancy, number of architectural/physical registers used, and many others.

4. EXPERIMENTAL RESULTS

In order to measure the accuracy of SLATE's core model, we created a system including one core (Power4-like, as described in Section 3.1.1), L2 cache, bus and memory controller, and executed the SPEC CINT2000 benchmarks, and measure the core IPC (instructions-per-cycle), which is an accepted measure of micro-architecture performance. We then compared the results against the IPC numbers given by a production-level performance simulator used internally in IBM on all POWER architecture designs [1]. For 8 out of 12 benchmarks, SLATE produced IPC numbers within 11% of the production simulator results. On average for all 12 benchmarks SLATE results were within 16% of the production simulator results, which is acceptable for an early analysis system.

In order to evaluate SLATE's simulation performance, we created 3 systems using 1 core, 2 cores and 4 cores, respectively. In all cases, each core was connected to a private L2 cache, which was connected to a bus and a memory controller. Realistic parameters and delays were used in all cases. We ran the 12 SPEC CINT2000 benchmarks on all 3 systems, using the same benchmark on all cores, and measured IPC, as well as various simulation performance metrics such as peak memory, simulation-cycles per second, and instructions per second.

SLATE's raw simulation performance for the 1-core system, including cycle-accurate performance analysis and power computation, was on average 55k simulation cycles per second for SPEC CINT2000 benchmarks, on a 2.4GHz X86-based Linux workstation. In this version of SLATE, no effort was made to optimize the simulation speed, and the freely available SystemC runtime environment was used with GCC compiler.

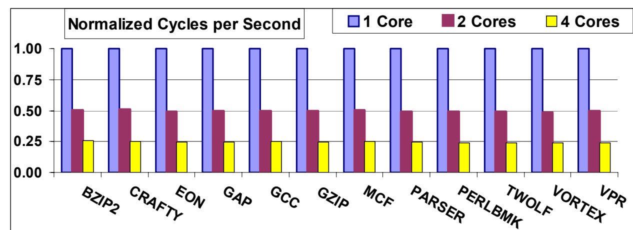

Figure 4: Cycles-per-second comparison: 1-core, 2-core and 4-core systems

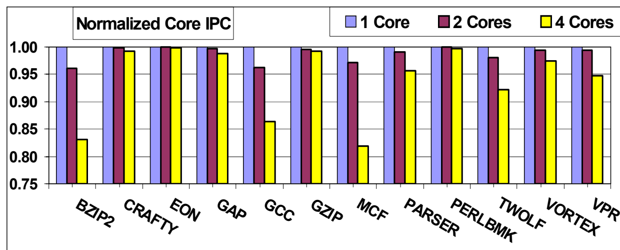

Figure 5: Core IPC comparison: 1-core, 2-core and 4-core systems

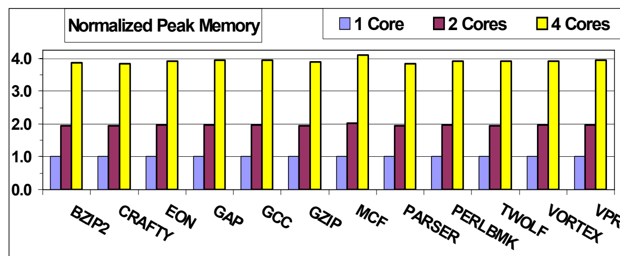

Figure 6: Peak memory comparison: 1-core, 2-core and 4-core systems

To evaluate SLATE's scalability, we compared the peak memory and simulation performance of the 1-core, 2-core and 4-core systems. The complexity of the 2-core and 4-core models is approximately 2x and 4x the complexity of the 1-core model. Thus, it is expected that simulation performance for multi-programmed workloads degrades linearly with the model complexity. This was observed in practice. Figure 4 shows that cycles-per-second for 2-core and 4-core models were roughly ½ and ¼ of the 1-core values. We have simulated systems with up to 8 cores in acceptable run times. This linear behavior holds true in SLATE when simulating multi-programmed workloads on CMP models. It may not be the case for multi-threaded applications where there is significant data sharing.

In the absence of any bus contention to memory and coherency updates, the IPC values on any core on the 3 systems should be the same for the same benchmark. However, due to contention and coherency delays, the IPC values may vary slightly (usually degrading) from the 1-core to 2-core, to 4-core systems. Figure 5 shows the normalized IPCs for the various benchmarks on the 1-core, 2-core and 4-core systems, where it can be seen that small IPC degradation (<10% in most cases) occurs on most benchmarks. Memory-bound benchmarks, such as MCF, are likely to show larger IPC degradation on the 4-core system, due to more memory accesses and more contention on the bus. Note that total instructions-per-second will follow the same variation as IPC and not vary significantly among the 3 systems (when considering the sum of the instructions executed on all cores).

Memory usage is an important consideration in performance models which may simulate for billions of instructions. It is critical that memory allocation depends on the model size and characteristics only and not on the simulation time. On a 1-core system the peak memory consumption in SLATE was about 44MB. Figure 6 shows the normalized memory consumption for the various benchmarks on the 1-core, 2-core and 4-core systems. As expected, peak memory grew linearly with the model complexity (just under 2x and 4x for the 2-core and 4-core systems).

5. CONCLUSIONS

This paper presents an overview of SLATE, a tool for early analysis of performance, power, physical and thermal aspects of multi-core systems. It allows designers to assemble, configure and simulate multi-core systems with memory hierarchy and buses. The components are modeled in SystemC using cycle-accurate transaction-level abstractions and include detailed performance and power models.

SLATE relies on two basic performance modeling building blocks for building complex components, namely, the *Computation* module and *Delay* channels. Different types of *Delay* channels are provided in support of different communication schemes. In order to minimize the number of ports of complex components, a special *Channel_Container* was developed to encapsulate any number of *Delay* channels, which can then be connected using a single component port.

Internally to the tool, the SystemC performance models are linked to a structural netlist representing the block diagram/floorplan of the design which is stored in OpenAccess (OA). This OA netlist, properly annotated with performance and power values, serves as input to integrated physical and thermal analysis tools.

SLATE provides a unique framework and tools supporting early design analysis of multi-core systems.

Acknowledgements

Authors would like to thank Ravi Nair and Kyle Nesbit for writing the original SystemC implementation of the core module.

6. REFERENCES

[1] S.R. Kunkel, R.J. Eickemeyer, M.H. Lipasti, T.J. Mullins, B. O'Krafka, H. Rosenberg, S.P. VanderWiel, P.L. Vitale, and L.D. Whitley, "A performance methodology for commercial servers". IBM Journal of Research & Development, Vol.44, No.6, November, 2000.

[2] M. Moudgill, J.-D. Wellman and J. Moreno, "Environment for PowerPC microarchitecture exploration". IEEE Micro, Vol.19, No.3, pp.15-25, 1999.

[3] M. Reily and J. Edmondson, "Performance simulation of an Alpha microprocessor". IEEE Computer, Vol.31, No.5, pp.50-58, 1998.

[4] T. Austin, E. Larson and D. Ernst, "SimpleScalar: an infrastructure for computer system modeling". IEEE Computer, Vol.35, No.2, pp.59-67, February, 2002.

[5] D. Brooks, P. Bose, V. Srinivasan, M.K. Gschwind, P. Emma, and M. Rosenfield, "New methodology for early-stage microarchitecture-level power-performance analysis of microprocessors". IBM Journal of Research & Development, Vol.47, No.5/6, September/November, 2003.

[6] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimization". In *Proceedings of the 27th annual international symposium on Computer architecture (ISCA 2000)*, Vancouver, 2000.

[7] Open Access Initiative, www.si2.org.

[8] J. Tendler, J.S. Dodson, J.S. Fields Jr., H. Le, and B. Sinharoy, "POWER4 system microarchitecture". IBM Journal of Research & Development, Vol.46, No.1, January, 2002.

[9] B. Sinharoy, R.N. Kalla, J.M. Tendler, R.J. Eickemeyer, and J.B. Joyner, "POWER5 system microarchitecture". IBM Journal of Research & Development, Vol.49, No.4/5, July/September 2005.

[10] SPEC Standard Performance Evaluation Corporation, www.spec.org.