MC-Sim: An Efficient Simulation Tool for MPSoC Designs

Jason Cong, Karthik Gururaj, Guoling Han, Adam Kaplan, Mishali Naik, Glenn Reinman
Computer Science Department, University of California, Los Angeles
Los Angeles, CA 90095, USA

ABSTRACT

The ability to integrate diverse components such as processor cores, memories, custom hardware blocks and complex network-on-chip (NoC) communication frameworks onto a single chip has greatly increased the design space available for system-on-chip (SoC) designers. Efficient and accurate performance estimation tools are needed to assist the designer in making design decisions. In this paper, we present *MC-Sim*, a heterogeneous multi-core simulator framework which is capable of accurately simulating a variety of processor, memory, NoC configurations and application specific coprocessors. We also describe a methodology to automatically generate fast, cycle-true behavioral, C-based simulators for coprocessors using a high-level synthesis tool and integrate them with *MC-Sim*, thus augmenting it with the capacity to simulate coprocessors. Our C-based simulators provide on an average 45x improvement in simulation speed over that of RTL descriptions. We have used this framework to simulate a number of real-life applications such as the MPEG4 decoder and litho-simulation, and experimented with a number of design choices. Our simulator framework is able to accurately model the performance of these applications (only 7% off the actual implementation) and allows us to explore the design space rapidly and achieve interesting design implementations.

1. INTRODUCTION

The costs and technical challenges of designing application-specific integrated circuits (ASIC) have increased substantially as we move into nanometer technologies. At the same time, the exponential increase in silicon capacity has made it possible to integrate multiple processors in a single chip. Aside from processor cores, it has also been possible to integrate a variety of components onto the same chip. These components include on-chip L2 cache banks, scratchpad memories for rapid memory operations, DMA and memory controllers. Chip manufacturers are also incorporating coprocessors and reconfigurable fabric alongside processor cores. The encryption engines in Niagara 2 [9] are examples of coprocessors. FPGA manufacturers such as Xilinx [11] and Altera [25] provide systems with processor cores and reconfigurable fabric on the same chip. Examples are Virtex-4 from Xilinx that includes up to two PowerPC 405 cores and Excalibur from Altera that includes an ARM922T processor. An important component that we have not mentioned is the on-chip communication fabric. While most of the commercial chip-multiprocessor (CMP) designs use bus-based architectures for communication, it is anticipated that CMP designs in the near future will include complex network-on-chip (NoC) architectures. This is because bus-based architectures have been shown to scale poorly in terms of energy consumption and performance as the number of components increase [1].

The ability to integrate such diverse components on a single chip has led to the coining of the term Multi-Processor System-on-Chip (MPSoC). To manage the massive increase in the design space available for system designers, simulation tools, which can provide rapid and accurate estimations of the performance and cost of design instances, are required. Specifically, the tool should possess the capability to rapidly and accurately evaluate the performance of multi-core designs, model a variety of memory systems and NoC configurations, simulate coprocessors efficiently and allow the designer to migrate from one design instance to another easily.

1.1 Previous Work

Single processor simulators

Traditional single-core simulators, such as SimpleScalar [7], were designed to model out-of-order superscalar processor pipelines, and lack the ability to model multiple application workloads or shared-memory parallel applications, let alone the simulation of multiple processor cores. Architecture description languages such as LISA [15] and EXPRESSION [16] allow designers to generate compilers and functional and cycle-accurate simulators for application-specific processors from a high-level specification. While these environments provide designers with the ability to customize the instruction set and processor architecture, considerable effort is needed to come up with descriptions so that efficient compilers and simulators are generated.

Multi-core simulators

Full-system simulation is frequently used to evaluate CMP architectures. Although many full-system simulators [17][28] [33] are capable of executing unmodified binaries, and modeling partial or complete operating system execution, they are often very slow and heavyweight, even when simply modeling the functional execution of an application, unburdened by cycle-accurate models of cores and memories. For instance, Michigan's M5 Simulator [33] supports the booting of entire operating systems, as well as the execution of unmodified application binaries using system-call emulation. However, only a shared-bus model is supported to model interconnection of multiple processor cores, and only four processor cores can be simulated at a time. Virtutech's Simics [17] is a full-system functional simulator that can support the execution of unmodified kernel and driver code. Although Simics does not come prepackaged with a detailed processor model, it has APIs which support plug-in extension with timing or microarchitecture models. One popular Simics plug-in is Wisconsin's GEMS Toolset [32], which contains a separate processor model and memory system simulator. Unfortunately, this framework is slow and heavy (nearly two orders slower than SimpleScalar), and as modeled systems are scaled to even tens of cores, the complete execution of even a small multithreaded application could require weeks of simulation time.

The SESC simulator [6] is a very fast single/multi-processor simulator which functionally simulates instructions on a MIPS

978-1-4244-2820-5/08/$25.00 ©2008 IEEE

emulator [34] before modeling their execution on an out-of-order superscalar processor pipeline. To enable faster simulation, the effects of wrong-path instructions are not modeled, and no address translation is performed. SESC supports the execution of either single-threaded or shared-memory multi-threaded applications (using a POSIX thread-based interface). However, only a single application can be executed in a given workload. Also, although many cores can be simulated by SESC, only snoopy cache-coherence protocols and bus-based interconnects are supported for CMP simulation.

Processor-coprocessor simulation

Researchers investigating hardware-software co-synthesis have also developed tools for performance estimation of heterogeneous systems. In [13], coprocessors are simulated using a VHDL simulator with which the software code communicates through socket calls. Simulations involving RTL descriptions in VHDL are unacceptably slow even for medium sized coprocessors.

C/C++ based system-level description languages such as SystemC [2] and SpecC [3] have gained popularity as environments for specifying both hardware and software components in the design. However, processor simulators described in these languages are significantly slower than traditional instruction set simulators (ISSs) such as SimpleScalar [7] or SESC [6]. The work published in [20][37][21] tries to address this limitation by integrating an ISS with SystemC, which is used to model coprocessors and on-chip interconnect. A set of interfaces are defined to allow the SystemC kernel to interact and synchronize with the ISS. While the software simulation speed is improved greatly over pure SystemC models, the hardware blocks are still described at RT-level, thus slowing down the overall simulation significantly. The MESCAL project [13] uses the Liberty Simulation Environment [10] to construct a simulator for the specified architecture that may include both cores and custom hardware accelerators.

Emulation

An ISS is used to simulate the processor core while a prototype of the coprocessor is synthesized on FPGA in [14]. Communication between the hardware and software components occurs via a PCI bus. In [19], the authors present an emulation platform consisting of multiple VLIW processors, memory blocks and FPGA with a flexible interconnect mechanism on the same board. While emulation allows designers to explore heterogeneous designs rapidly, its setup takes considerable effort. Also, long compilation times for FPGAs prevent fast design iterations.

Multi-core and coprocessor simulators

The MPARM simulator [30] is probably the only existing complete simulator framework for designs with multiple cores and coprocessors. However, the processor cores that are currently being supported in the MPARM framework are relatively simple and automatic construction of performance models for coprocessors from high-level functional specifications is not supported.

In this paper, we present a simulation framework that addresses several of the limitations of the aforementioned tools. We call our simulation framework *MC-Sim*, short for Multi Core Simulator. *MC-Sim* possesses several key capabilities. First, it allows designers to perform rapid and cycle-true simulation of heterogeneous multi-core designs. Second, a variety of configurations for the memory system can be modeled. Third, it has a flexible NoC model, which allows evaluation of different NoC architectures. Fourth, it is able to handle multi-tasking and simulate multi-threaded applications. Finally, it has the important feature of being able to automatically construct fast, cycle-true behavioral performance models for coprocessors from functional models by leveraging a high-level synthesis tool and integrating these performance models with the remainder of the design.

MC-Sim utilizes a hybrid model for simulation:

- A fast, cycle-accurate, functional ISS for modeling the processor cores.
- A detailed structural and cycle-accurate model for the NoC.
- A cycle-true, behavioral, C-based model for coprocessors, which provides more than 45x improvement in simulation speed over SystemC RTL models.

We summarize the features of *MC-Sim* and the advantages over other popularly used simulators in Table 1.

The rest of the paper is organized as follows. Section 2 provides a high level view of the *MC-Sim* simulator framework. The flow for generating and integrating cycle-true C-based coprocessor models from functional specifications is described in Section 3. We also describe how the coprocessor models are integrated with the rest of *MC-Sim*. In Section 4, we present experimental results for some real-world applications. Finally, in Section 5, we conclude the paper and provide directions for future work.

2. *MC-Sim* SIMULATOR FRAMEWORK

In order to investigate the design-space of future multi-core systems, a MPSoC simulator should be able to efficiently model the execution of a many-core CMP. It should be able to support workloads comprised of one or more single or multi-threaded applications. Moreover, support for modeling a multi-banked [35] last-level shared cache, a directory-based cache coherence protocol that is robust in the face of network reordering, and a switched network-on-chip that can be parameterized for a range of topologies is essential.

A top level view of the major components of *MC-Sim* is shown in Figure 1. *MC-Sim* can tractably simulate the execution of 64 or more active superscalar cores, and has a flexible configuration interface that allows fast mapping of any thread from any application to any core in the system. For this framework, we have designed a custom L1 and L2 cache interface, with a robust, directory-based, MSI cache-coherence protocol. Caches, cores, coprocessors, and memory interfaces can be flexibly assigned to different positions in the interconnect topology, and inter-router links can be added or disabled using configuration files, greatly easing the generation of a variety of topologies. These features allow us to quickly evaluate the impact of a variety of design options for many-core chip-multiprocessors and their interconnection networks, directly measuring their impact on application performance. Unlike SESC [6], *MC-Sim* can execute workloads comprised of multiple applications, each either single- or multi-threaded. A thin layer of memory system support, called the Central Page

Table 1: Feature comparison of microarchitecture simulators

	Cycle	Multiple cores	Multi-threading	Speed	Scalable communication	Multiple application	Coprocessor

	accurate		support		models	workloads	Simulation
Simple Scalar	Yes	No	No	fast: ~150 KIPS	n/a	No	No
Simics/ GEMS	Yes	Yes	Yes	slow: ~7.5 KIPS @ 8 cores	Yes	Yes	No
M5	Yes	Up to 4	Yes	slow: comparable to Simics	No	Yes	No
SESC	Yes	Yes	Yes	fast: ~1.5 MIPS	No	No	No
MPARM	Yes	Yes (simple cores)	Yes	-	Yes	Yes	Yes
MC-Sim	Yes	Yes	Yes	fast: ~32 KIPS @ 64 cores (64 threads)	Yes	Yes	Yes

Handler, allows each application to share a physical view of main memory, and allocates physical page frames to each application on-demand. We decided not to support a full fledged OS primarily because we wanted to ensure the tractable simulation of multiple cores.

2.1 Processor Cores

To model the processor cores in our framework, we employ a heavily modified version of the SESC simulator [6], harnessing only its core processor model, and completely rewriting its L1 and L2 cache code, its on-chip interconnection network, and its memory manager. We chose SESC primarily because of its high simulation speed, which is roughly 10 times [6] that of SimpleScalar [7], as shown in Table 1. This allows us to simulate designs with 64+ cores. However, there is no practical restriction on the choice of simulator for processor model, and other simulators could be substituted if desired, or a fully customized model could be designed from scratch. To integrate a pre-existing processor model into *MC-Sim*, it need only be partitioned at its L1 instruction and data-cache access point, and rewritten to call (and be called-back) by the API of the *MC-Sim* memory system.

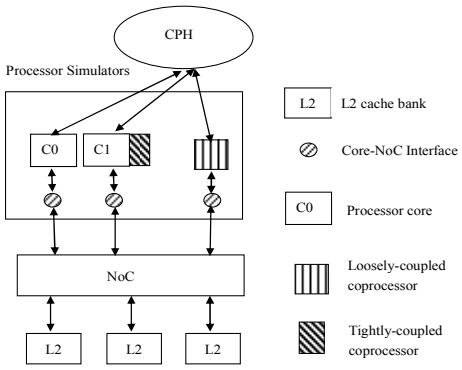

Figure 1: Top level view of *MC-Sim*

In *MC-Sim*, each core has a local L1 instruction and data cache. We have implemented an L1 to L2 cache-communication infrastructure which maintains cache coherence with a scalable directory-based MSI protocol. Additionally, we have defined an interface between the processor core and the NoC that is flexible enough to model all the kinds of interaction between the core and the remainder of the system. For multi-tasking workloads, each application is given its own simulation instance, where simulation instances share memory through the Central Page Handler (CPH) and NoC interfaces. Any core in the system may have a unique configuration, although the instruction set of all processor cores is identical. For example, a simulator instance might consist of two cores, one of which is a 4-issue superscalar processor while the other is a single-issue processor with in-order execution. This allows designers to experiment with configurations which have the same number of cores but different areas by varying the complexity of the cores used.

2.2 L2 Cache and Central Page Handler

The L2 cache is organized as a shared set of NUCA [35] cache banks where each bank has its own cache controller. Each cache controller implements a directory-based cache coherence protocol. Thus, different cores can share data through L2 caches. The number, sizes and positions of the cache banks are configurable. The CPH handles the memory management functions that are normally performed by the OS such as allocating physical pages in memory and keeping track of the virtual address to physical address mapping for each application by creating page tables. This has made modeling multitasking possible without the overhead of full system simulation.

2.3 Network on Chip

MC-Sim supports simulation of a variety of NoC architectures. The designer can perform pure timing simulation using the NoC implemented within *MC-Sim*. Statistics generated from this simulation can be used as input to a detailed router model (example Orion [36]) to generate NoC power numbers. Currently, the default implementation for the NoC in *MC-Sim* is a 2-D mesh with packet switching for on-chip communication. L2 cache banks, processor cores and main-memory interfaces can be assigned flexibly to different locations in the network using a configuration file. Another configuration file can be optionally specified that adds extra links (or Express Channels [31]) to the mesh, each given a separate user specified link bandwidth. Using these capabilities, a user can model additional mesh-based topologies, such as a torus, or a 3D die-stacked mesh network. Each core and L2 cache bank has a pair of message queues, one each for incoming and outgoing messages respectively, through which the core/bank communicates with the network. A cycle accurate simulator model for the NoC enables us to take into account communication latency between different components of the design.

3. COPROCESSOR MODEL

3.1 Automatic Construction of Cycle-true Coprocessor Models

In this section, we describe the methodology to automatically build C-based *cycle-true behavioral* coprocessor models from functional C specification. We call it cycle-true behavioral

models because our model accurately reports the number of cycles elapsed since the invocation of the coprocessor to its termination. Meanwhile, it is functionally identical to the original coprocessor design. However, unlike traditional cycle-accurate RTL models, which accurately simulate the signals and states every clock cycle, our models filter out details of the intermediate states in the microarchitecture of the coprocessor. Thus, we can achieve significant simulation speedup compared to RTL simulation methodology, which could be very valuable to design space exploration at early design stages.

We utilize a behavioral synthesis tool, xPilot [8] from UCLA, which can generate a synthesizable RTL description from a functional C-specification of the desired coprocessor. The xPilot synthesis flow uses a CDFG to represent the coprocessor functional specification and a state transition graph (STG) to represent the scheduling information of the final hardware design. We first formally define these two terms:

DEFINITION 1. A CDFG is a directed graph $G(V_G; E_G)$ where $V_G = V_{bb} \cup V_{op}$ and $E_G = E_c \cup E_d$. V_{bb} is a set of basic blocks (i.e., data flow graphs). V_{op} is the entire set of operations in G, and each operation in V_{op} belongs to exactly one basic block. Data edges in E_d denote the data dependencies between operations. Control edges in E_c represent the control dependencies between the basic blocks. Each control edge e_c is associated with a branching condition $bcond(e_c)$.

Figure 2 shows the CDFG for an implementation of the greatest common divisor (GCD) algorithm. Note that the *phi* node is a special operator in the single static assignment representation (SSA) [22].

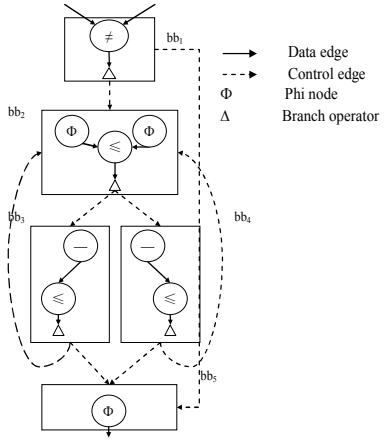

Figure 2: CDFG for GCD implementation

The scheduling information obtained from xPilot is captured by an STG which is described as follows:

DEFINITION 2: An STG is a directed graph $G_s(s_0; V_s; E_s)$. V_s is the set of control states with initial state s_0. Each control state $s \in V_s$ contains a set of operations $OP(s)$ and each operation op is associated with a guard condition $gc(op)$ to guard its execution. Every operation in the CDFG G is assigned to a state and every operation in the same state s is executed in the same clock cycle in the generated RTL description. We say a basic block b belongs to state s if s has at least one operation of b. E_s is the set of transitions edges between the control states and each state transition edge $t_{u\to s}$, between state u and s is associated with a transition condition $t_c(t_{u\to s})$. The transition condition, t_c on a transition edge $t_{u\to s}$ between two states u and s, is the boolean OR of the $bcond(e)$ for the set of control edges, $E_{u,s}$, originating in basic blocks belonging to u and terminating in basic blocks belonging to s i.e. $t_c(t_{u\to s}) = (\vee_{e_{u,s}} bcond(e_{u,s}))$ where $e_{u,s} \in E_{u,s} = \{e \mid e \in E_c,\ e = (b' \to b) \text{ and } b' \in u,\ b \in s\}$. xPilot scheduler will guarantee that one and only one branch condition will be true when the state transition occurs. A control edge e is said to be a *sub-edge* of STG edge e_s if $bcond(e)$ is a term of the the transition condition associated with e_s.

Figure 3 illustrates the xPilot synthesis flow. At the front-end, xPilot uses the LLVM compiler [26] from UIUC to parse the functional description of the coprocessor specified in C/SystemC. The synthesis tool creates a System-level Synthesis Data Model (SSDM), which is the central data model of the synthesis system. The C/SystemC functions in the behavioral description of the desired coprocessor are represented as CDFGs in the SSDM. Once the SSDM is built, the scheduling engine schedules the operations in the CDFGs in such a manner that user-specified timing constraints are met. The back end of xPilot performs resource binding and generates RTL code.

We add a module in the flow of xPilot to automatically construct C-based cycle accurate behavioral models.

DEFINITION 3: A *Cycle-true Behavioral C Model* (CTB-C) for a coprocessor in our simulator framework is defined to be a CDFG *G'* with the following characteristics:

1. It is functionally equivalent to the CDFG G, i.e. it produces the same output as CDFG G for the same input.
2. It has a cycle count variable, CV, to track the number of clock cycles elapsed since the invocation of the coprocessor.
3. After the coprocessor model is executed, the value of CV is identical to the number of cycles elapsed in the simulation of the RTL description (generated by the high-level synthesis tool) for the same set of inputs.

Thus, the coprocessor model generates the same output as the CDFG and determines the exact number of cycles it would take for the simulation of the RTL model to complete computation.

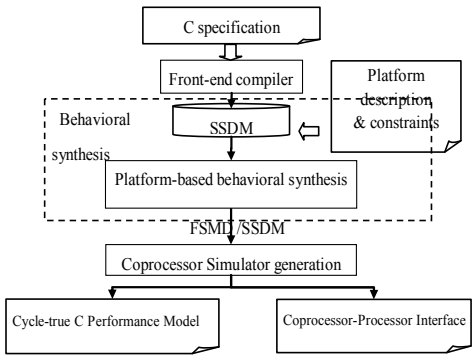

Figure 3: Flow for coprocessor performance model generation

Since CDFGs and STGs capture the behavior of a design and the scheduling information respectively, together they contain all the data needed to generate the cycle-true simulator for the coprocessor. We first make the following observation:

$$Num\_cycles = \sum_{s} Num\_activations(s) \quad s \in V_s \text{ of } STG$$

In our work, an additional customizable function called *performance calculator* is also introduced, which is used to track

the timing of the coprocessor. The default implementation for this calculator function increments the value of *CV* by one whenever it is invoked. The calculator function should be called once and only once when its corresponding state is activated. We use the following notation in the rest of the paper. *G* is a CDFG and *s* is a state in the STG G_s. B_s is the set of basic blocks that belong to state *s*.

For a given state *s*, we consider set D_s that consists of all basic blocks bb_1 such that there is a basic block $bb_2 \in B_s$ and control edge $bb_1 \rightarrow bb_2$ is a sub-edge of an STG edge terminating at state *s*. Thus, set D_s represents all those basic blocks from which the coprocessor can transit to state *s*.

LEMMA 1: If the performance calculator is inserted at the end of all basic blocks bb_1 in set D_s and executed subject to the guard condition on control edge $bb_1 \rightarrow bb_2$ ($bb_2 \in B_s$), then *CV* is incremented if and only if state *s* is activated during the RTL simulation.

Proof: The *if* part – State *s* is activated implies that a state transition was made to *s*. Based on the definition of STG, one and only one term in the state transition edge condition is true. It also implies that the execution path in the original CDFG is from the corresponding sub-edge $bb_1 \rightarrow bb_2$ whose guard condition is true. According to our rule, the inserted performance calculator in bb_1 will be invoked once and *CV* is incremented by one.

The *only-if* part – When basic block bb_1 executes, the condition of control edge $bb_1 \rightarrow bb_2$ ($bb_2 \in B_s$) is satisfied, the performance calculator is invoked. This also implies that in the STG a state transition to *s* will be made.

Figure 4(a) shows a scheduling result for the GCD function shown in Figure 2. Figure 4(b) shows the positions in the CDFG where the performance calculator function should be invoked for state transitions $S_1 \rightarrow S_2$, $S_1 \rightarrow S_4$ and $S_2 \rightarrow S_3$. For state transitions from S_1, the performance calculators PC_1 and PC_2 are inserted as shown with the guard conditions associated with control edges $bb_1 \rightarrow bb_2$, and $bb1_1 \rightarrow bb_5$ respectively. Similarly, for state transitions from S_2, the performance calculators PC_3 and PC_4 are inserted as shown with the guard conditions associated with control edges $bb_2 \rightarrow bb_3$, and $bb1_2 \rightarrow bb_4$ respectively. We do not show the performance calculators for the other states because of lack of space.

Theorem 1: For any coprocessor, when the CDFG annotated with performance calculators, *G'*, completes execution, the value of *CV* will be identical to the cycles elapsed in the simulation of the RTL of the coprocessor for the same inputs.

The proof follows from Lemma 1.

3.2 Coprocessor Integration

3.2.1 Communication Architecture

We model the communication overhead between the cores and coprocessors. We allow the coprocessor to interface with the cores in two different ways.

1. *Tightly-coupled coprocessor*: Only a single core can access the coprocessor. This mechanism is similar to that used for the encryption engines in the Niagara 2 processor [9]. The coprocessor has direct access to the L1 data cache of the core from which it can read input data and write out its output. The communication between the core and coprocessor takes place through a high-speed interconnect such as the *HyperTransport*™ [38] that provides high bandwidth between the core and coprocessor. In our experiments, we set the bandwidth to be 2 Gbps, which is a conservative estimate of the achievable bandwidth on the HyperTransport. The coprocessor CP_1 in Figure 5 shows the architecture associated with a tightly coupled coprocessor. The communication delay is modeled as the total number of cycles to bring the data to the L1 cache and to transfer the data over the high-speed link to the coprocessor. Similarly, we model the delay associated with transferring the output data back to the L1 cache.

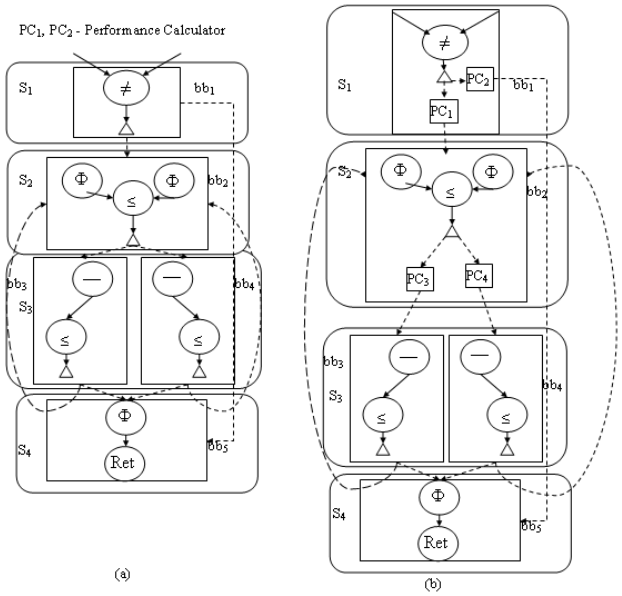

Figure 4: An example of performance calculator insertion

2. *Loosely-coupled coprocessor*: The coprocessor is attached to a router in the NoC just like a processor core. Such a coprocessor can be shared by multiple cores in the design. Similar to a core, it possesses a local L1 data cache. Communication with such a coprocessor takes place over the NoC, identical to how a core communicates. Data can be transferred to/from the coprocessor through the shared L2 cache banks. Figure 5 shows the architecture associated with a loosely coupled coprocessor. The communication delay is modeled as the total number of cycles to bring the data over the NoC to the L1 cache of the coprocessor from the L2 banks. Similarly, we model the delay associated with transferring the output data back to the shared L2 cache banks.

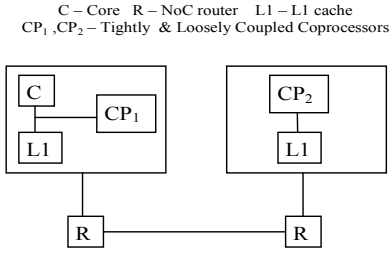

Figure 5: Coprocessor-core communication architecture

Tight coupling of coprocessors to cores greatly reduces the communication time; however, tightly coupled coprocessors cannot be shared by multiple cores. The designer can choose the mechanism that best suits the characteristics of the application.

3.2.2 Coprocessor API, Invocation and Execution

We provide a configuration file to *MC-Sim* that describes the location and the types of arguments of every coprocessor in the design. Each coprocessor instance in the design is given a unique

id. The configuration file is generated automatically during our high-level synthesis process.

We define a system call named *coprocessor()* to invoke a coprocessor from the application code. This system call takes as input the *id* of the coprocessor to be invoked and a list of addresses of the arguments for the coprocessor. The API is sufficiently generic to invoke any coprocessor irrespective of the number of arguments and the data types of the arguments. While this API is simple, it is adequate for our simulation purposes.

To invoke the coprocessor, the designer needs to modify the application by replacing the function call with a coprocessor system call. The arguments to this system call are simply the *id* of the coprocessor to invoke and the same arguments passed in the original function call; *MC-Sim* automatically performs data fetch and transfer to/from the coprocessor depending on the type of each argument.

We describe briefly the steps involved in the invocation and execution of the coprocessor module. Firstly, *MC-Sim* "traps" the coprocessor system call from the application binary. The argument *id* of the system call is used to obtain the location of the coprocessor, number of arguments and the sizes of the arguments. Using the starting address and size of each read and read/write argument, a series of read requests are issued to the memory hierarchy. Once all the reads are complete, the cycle-true coprocessor model is invoked to determine the values of the outputs and the number of cycles required for this computation. After computation is complete, output data is written by issuing a series of write requests to the memory hierarchy. During the read and write steps, the timing overhead associated with communication is modeled based on the architecture described in Section 3.2.1. Finally, the coprocessor signals to the invoking core that it has completed execution.

4. EXPERIMENTAL RESULTS

4.1 Evaluating the Coprocessor Simulators

We first compare the accuracy and simulation speed of our cycle-true behavioral C (CTB-C) model with RT-level SystemC model generated by our synthesis tool.

Table 2 shows the experimental results for five small-sized computation kernels used in media processing and encryption applictions. The generated C performance models can provide exact performance numbers while the simulation speed is much faster than RTL simulation. On an average, we can achieve 45.3X speedup for our benchmarks.

Table 2: #Cycles and speed of coprocessor simulator

Benchmark	Simulation Speed(sec)			#Cycles
	RTL-SystemC	CTB-C	Speedup	
Dct	0.128	0.004	32.8	147
Idct	0.219	0.004	56.2	283
motion compensation(MC)	4.170	0.086	48.5	873
pipelined MC	1.230	0.030	41.0	303
Sha	0.240	0.005	48.0	396

4.2 Lithography Simulation

In order to evaluate the complete simulation framework after integrating the coprocessor simulators with *MC-Sim*, we use a lithography simulation application developed in [24] as a test driver. Litho-simulation is a computation intensive application, which simulates the lithography process used for printing circuit patterns onto wafers. In [24], the computation intensive kernel of litho simulation is synthesized on FPGA as a coprocessor. A number of implementations based on different kernel and memory partitioning strategies were experimented with in [24]. We validate our simulation results against a design with 5x5 partition [24] implemented on XtremeData's XD1000 [27] development system, which consists an AMD Opteron 248 CPU and Altera's Stratix II FPGA. The implementation uses 25,042 ALUTs and 2,972,876 memory bits. A 15X speedup compared with the pure software implementation on this system was reported in [24]. We configure a similar system in *MC-Sim* by integrating the generated coprocessor simulator with a 4-issue, superscalar core. Table 3 shows the estimated speedup for different implementations over the purely software implementation. Comparing the result in the third column with the implementation in [24], we find that our simulation result is off by a mere 7%. Moreover, RTL simulation of the design used in [24] would not complete even after 1 week whereas our simulation finished within 90 minutes.

Table 3. Speedup for different coprocessor implementations

	3x3	4x4	5x5	6x6
Speedup	7.58	11.61	14.49	19.59

4.3 MPEG4 Decoder

Finally, we simulate a variety of design instances for the MPEG4 decoder provided by Xilinx [23]. The block diagram for the

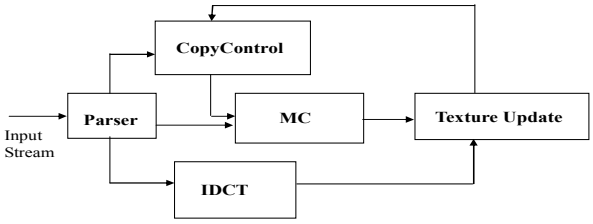

Figure 6: Block diagram for MPEG-4 decoder

MPEG4 decoder is shown in Figure 6. The frame data is divided into 16x16 macroblocks each containing six 8x8 blocks for YCbCr 4:2:0 formatted data. The goal was to achieve a throughput of 30 frames/second for the decoder. We first simulate the decoder on a 2GHz, 4-issue superscalar processor with 64KB L1 cache and 256KB L2 cache. We consider this system to be the base system with which we compare all other implementations. The profiling results for the decoder when running on the base system is shown in Table 4.

We generate all coprocessors using xPilot [8] and obtain area numbers for coprocessors from Magma synthesis tool [40]. For processors, we obtain area numbers from the ARM website [39]. We obtain the approximate area of our superscalar processor from the area of the Cortex A8 processor by ARM. A single issue processor in our experiments is assumed to have approximately the same area as a processor in the ARM 11 family. We report area numbers for 65nm TSMC technology. For the cases where the area numbers for the required technology are not available, we use quadratic scaling formulas for area [41]. The default NoC architecture for our case studies is a 2-D mesh.

Based on the block diagram, we first construct a multi-threaded implementation of the decoder with the *Parser, MotionCompensation (MC), TextureIDCT (IDCT)* and *Texture Update* modules running on four different cores in a pipelined

fashion. The *Copy Control* module is executed on the same core as the *Parser* module. However, as seen in the second row of Table 5, the pipelined version achieves modest speedup – about 1.47 times - but at a huge increase in area (speedup for a design is defined as the ratio of the frames per second for the design to the frames per second for the base system). This is because of the large disparities in the running times of the different modules, which caused the cores running the *Parser* and *TextureUpdate* to be idle for a large portion of the time. To obtain a better design, we synthesized a pipelined coprocessor for the *MC* module running at 200MHz and pipelined its execution with the software components in the system, which now run on a single 4-issue core. This implementation achieves 1.6X speedup.

Table 4. Profiling information for MPEG4 decoder

Modules	Percentage of total execution time	Avg. #Cycles/Call
Parser	21.55	2913
Copy Control	3.85	1476
MC	35.43	15,934
IDCT	19.08	24,810
TextureUpdate	5.66	2546

Further investigation reveals that the *IDCT* module became the bottleneck consuming a large portion of the time on the processor. We implement both the *IDCT* and *MC* modules as coprocessors, running alongside a single core. By pipelining the execution of the coprocessors with the software components of the design, we are able to achieve significant speedup – up to 2.75 times. All our results are listed in Table 5. In a further attempt to reduce system area, we simulate the coprocessors with a simple, single-issue in-order core with just 32KB L1 cache. As shown in the last row of Table 5, even this simple system meets the desired throughput of 30 frames/second. Thus, we obtained a low area design for the decoder. We repeated all the experiments using a 1 GHz processor. However, we did not find any configuration that achieves the required throughput.

5. CONCLUSION

In this paper, we have described the *MC-Sim* simulator framework that is capable of tractably simulating a MPSoC design with several heterogeneous components. Our framework is designed with the ability to simulate workloads comprised of multiple applications (i.e. multi-tasking), each of which can be single- or multi-threaded (i.e. co-operatively multithreaded). Additionally, *MC-Sim* supports a flexible memory system, which includes L1 caches that are local to processor cores, L2 caches that are shared between cores and software controlled scratchpad memories. Our framework has a flexible and accurate structural model for the NoC for on-chip communication. We also present a methodology to automatically generate and interface coprocessor simulators with processor cores. Experiments on real-life applications indicate that not only is our simulator accurate, but simulation times are well within acceptable limits. Note that our simulator framework can be integrated into platforms such as Metropolis [4] and Artemis [5] to provide accurate timing information. Future work will take into account power modeling for the cores and coprocessors as well.

6. ACKNOWLEDGEMENTS

This work is partially supported by the MARCO GSRC center, the SRC contract 2005-TJ-1317, and the NSF grant CNS-0725354.

7. REFERENCES

[1] H. G. Lee, N. Chang, U. Y. Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. In ACM Trans. on Design Automation of Electronic Systems (TODAES), 12(3), August 2007.

[2] Open SystemC Initiative. http://www.systemc.org.

[3] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao. SpecC: Specification Language and Methodology. Kluwer Academic Publishers, 2000.

[4] Abhijit Davare, Douglas Densmore, Trevor Meyerowitz, Alessandro Pinto, Alberto Sangiovanni-Vincentelli, Guang Yang, Haibo Zeng, Qi Zhu. A Next-Generation Design Framework for Platform-Based Design. Conference on Using Hardware Design and Verification Languages (DVCon), February, 2007.

[5] Andy Pimentel, Pieter van der Wolf, Bob Hertzberger, Ed Deprettere, Jos T.J. van Eijndhoven, and Stamatis Vassiliadis. The Artemis architecture workbench. In Progress Workshop 2000, Oct. 2000.

[6] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, Jan. 2005, http://sesc.sourceforge.net.

[7] D.C. Burger and T.M. Austin. The SimpleScalar Tool Set. Technical Report CS-TR-1997-1342, University of Wisconsin, Jun. 1997.

[8] J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang. Platform-Based Behavior-Level and System-Level Synthesis. In Proceedings of IEEE International SOC Conference, pp. 199-202, 2006.

[9] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. IEEE Micro, Vol. 25, No. 2, pp. 21-29, 2005.

[10] M. Vachharajani, N. Vachharajani, D. A. Penry, J.A. Blome, and D.I. August. Microarchitectural Exploration with Liberty. In Proceedings of the 35th International Symposium on Microarchitecture (MICRO), Nov. 2002.

[11] Xilinx Inc., http://www.xilinx.com.

[12] J.Liu, M. Lajolo, and A. Sangiovanni-Vincentelli. Software Timing Analysis Using HW/SW Cosimulation and Instruction Set Simulator. In International Workshop on Hardware/Software Codesign - CODES/CASHE 1998.

[13] D.I August, K. Keutzer, S. Malik, and R. Newton. A Disciplined Approach to the Development of Platform Architectures. SASIMI, Jan., 2001.

[14] M. Kudlugi, S. Hassoun, C. Selvidge, and D. Pryor. A Transaction-based Unified Simulation/Emulation Architecture for Functional Verification. In Proceedings of the 38th Conference on Design Automation, pp. 623-628, 2001.

[15] A. Hoffmann, H. Meyr, and R. Leupers. Architecture Exploration for Embedded Processors with LISA. Kluwer Academic Publishers, Dec. 2002.

[16] A. Halambi, P. Grun, et al. EXPRESSION: A Language for Architecture Exploration Through Compiler/Simulator Retargetability. In Proceedings of the European Conference on Design, Automation and Test, Mar. 1999.

[17] P.S. Magnusson, et al. Simics: A Full System Simulation Platform. Computer, Vol. 35, No.2, pp. 50-58, Feb. 2002.

[18] Y. Nakamura, et al. A Fast Hardware/Software Co-Verification Method for System-On-a-Chip by Using a C/C++ Simulator and FPGA Emulator with Shared Register Communication. In Proceedings of the 41st Design Automation Conference, pp. 299-304, 2004.

[19] M.D. Nava, el al. An Open Platform for Developing Multiprocessor SoCs. Computer, Vol.38, No.7, pp. 60-67, Jul. 2005.

[20] L. Formaggio, F. Fummi, G. Pravadelli. A Timing-Accurate HW/SW Co-simulation of an ISS with SystemC. In Proceedings of

the 2nd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, Sep. 2004.

[21] D. Bertozzi , D. Bruni. Legacy SystemC Co-Simulation of Multi-Processor Systems-on-Chip. In Proceedings of the 2002 IEEE International Conference on Computer Design, pp. 494, Sep. 2002.

[22] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. ACM Transactions on Programming Languages and Systems, Vol. 13, Issue 4, pp. 451-490, 1991.

[23] P. Schumacher and W. Chung. FPGA-based MPEG-4 codec. DSP Magazine, pp. 8–9, 2005.

[24] J. Cong and Y. Zou. Lithographic Aerial Image Simulation with FPGA-Based Hardware Acceleration. In Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, 2008.

[25] Altera Corp., http://www.altera.com.

[26] The LLVM Compiler Infrastructure., http://llvm.org.

[27] XtremeData, Inc., http://www.xtremedatainc.com/

[28] M. Rosenblum, S.A. Herrod, E. Witchel, and A. Gupta. Complete Computer Simulation: The SimOS Approach. IEEE Parallel and Distributed Technology, Vol. 3, No. 4, 1995.

[29] L. Cai, and D. Gajski. Transaction Level Modeling: An Overview. In Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 19-24, 2003.

[30] MPARM, http://www-micrel.deis.unibo.it/sitonew/research/mparm.html

[31] W.J. Dally. Express Cubes: Improving the Performance of K-ary N-cube Interconnection Networks. IEEE Transactions on Computers, Vol. 40, No. 9, Sep. 1991.

[32] M.M.K. Martin, et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. SIGARCH Computer Architecture News, Vol. 33, No. 4, Sep. 2005.

[33] N.L. Binkert, R.G. Dreslinski, L.R. Hsu, K.T. Lim, A.G. Saidi, and S.K. Reinhardt. The M5 Simulator: Modeling Networked Systems. IEEE Micro, Vol. 26, No. 4, 2006.

[34] J.E. Veenstra and R.J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Jan. 1994.

[35] C. Kim, D. Burger, and S.W. Keckler. An Adaptive, Non-uniform Cache Structure for Wire-Delay Dominated On-chip Caches. In Proceedings of ASPLOS-X, Oct. 2002.

[36] H. Wang, X. Zhu, L-S. Peh, and S. Malik. Orion: A Power-Performance Simulator for Interconnection Networks. In Proceedings of MICRO 35, Nov. 2002.

[37] F. Fummi, S. Martini, G. Perbellini, and M. Poncino. Native ISS-SystemC Integration for the Co-simulation of Multi-processor SoC. In Proceedings of the conference on Design, Automation and Test in Europe, pp. 10564, Feb. 2004.

[38] J Trodden and D. Anderson. HyperTransport System Architecture. Addison-Wesley Developer's Press, 2003.

[39] ARM website http://www.arm.com/products/CPUs/application.html

[40] Magma, Inc http://www.magma-da.com.

[41] S. Borkar. Design Challenges of Technology Scaling. IEEE Micro, Vol. 19, No. 4, pp. 23-29, 1999.

Table 5. MPEG-4 decoder performance for different configurations

	L1 Cache	Configuration	Frames/second	Speedup	Area (mm^2) (cores+coprocessors)
2GHz	64KB	4-issue, single core 2GHz	19.21	Base System	4
	64KB	4-core, pipeline	28.23	1.47	16
	64KB	4-issue core+*MC* coprocessor	31.09	1.62	4.021
	64KB	4-issue core+*MC*+ *IDCT* coprocessor	53.01	2.75	4.064
	32KB	1-issue core+*MC*+*IDCT* coprocessor	34.24	1.78	2.56
1GHz	64KB	4-issue core+*MC*+*IDCT* coprocessor	27.83	1.44	4.064
	64KB	1-issue core+*MC*+*IDCT* coprocessor	21.43	1.11	2.56