Multiprocessor simulation using communicating sequential processes

Pranav S. Vaidya* and Jaehwan John Lee

Department of Electrical and Computer Engineering,
Indiana University – Purdue University Indianapolis,
723 West Michigan Street, SL 160,
Indianapolis, IN 46202-5132, USA
Fax: 1(317) 274-4493
E-mail: psvaidya@iupui.edu
E-mail: johnlee@iupui.edu
*Corresponding author

Abstract: To tackle the increasing complexity of capturing the design and subsequent simulation of multicore/multiprocessor systems, it may be beneficial to consider formal methods of high-level design. However, current formal methods of high-level multiprocessor system design and simulation often use sequential/parallel discrete event simulation with cooperative multithreading and hence do not exploit the true multiprocessing capabilities of simulation hosts now available even at the desktop computing level. Hence in this article, we present our ongoing research work on a multiprocessor simulator based on the formal method of communicating sequential processes (CSP). Here, we show how a multiprocessor system can be described using the operators of CSP. Furthermore, we show how this formal description of a multiprocessor system can be mapped to the functions provided by the Kent C++CSP multithreading library for creating a multithreaded multiprocessor simulator. Finally, we present the results of the simulator performance obtained using two applications.

Keywords: simulation; computer architectures; multiprocessor systems; communicating sequential processes; CSP; models of computation.

Reference to this paper should be made as follows: Vaidya, P.S. and Lee, J.J. (2010) 'Multiprocessor simulation using communicating sequential processes', *Int. J. Computer Aided Engineering and Technology*, Vol. 2, No. 1, pp.94–111.

Biographical notes: Pranav S. Vaidya received his BE in Computer Engineering from Pune University, India in 2004. Between 2004 and 2005, he worked as a member of technical staff with Persistent Systems Pvt. Ltd., India. He received his MS in Electrical and Computer Engineering from Indiana University – Purdue University Indianapolis (IUPUI) in 2007. Currently, he is a PhD student at IUPUI; his research interests include modelling and multithreaded simulation of traditional as well as hybrid (reconfigurable) computing systems. His other research interest includes investigation of FPGA-assisted data stream management systems for real-time multimedia sensor networks. He is a member of the ACM.

Jaehwan John Lee received his BS from Kyungpook National University, Korea. Between 1986 and 2000, he was a Senior Researcher for the Agency for Defense Development in Korea, where he was involved in the development of guided missiles. He received his MS and PhD in Electrical and Computer Engineering from the Georgia Institute of Technology in 2003 and in 2004, respectively. He joined the faculty of the Department of Electrical and Computer Engineering, IUPUI in 2005. His research interests include development of novel parallel hardware-oriented algorithms, hardware/software codesign, multi-processor system-on-a-chip design issues, and multithreaded simulation of computer architecture and systems.

1 Introduction

Detailed quantitative evaluation of computing systems using simulation is the most commonly used approach in computer architecture research (Skadron et al., 2003). Researchers in both industry and academia often leverage simulation to evaluate novel ideas and characterise the nature of design space. Nevertheless, increasing complexity of multiprocessor systems is making high-level design and creation of simulators more and more tedious and time-consuming. To tackle this increasing complexity, formal methods can be used to capture the high-level design and create/synthesise corresponding multiprocessor system simulators. However, there are two major issues in using formal methods for multiprocessor system design and simulation. First, there are limited case studies that use formal methods other than sequential/parallel discrete event simulation (SDES/PDES) for multiprocessor system simulation. Most multiprocessor simulators and simulation frameworks use either SDES (Vachharajani et al., 2002; Emer et al., 2002; Perez et al., 2004) or cooperative multithreading (Ranganathan et al., 1997; SystemC, 2009) as their simulation methodology. This also represents the second issue, where such simulators and simulation frameworks do not exploit the true multiprocessing capabilities now available even at the desktop computing level with multicore/multiprocessor systems. Hence, there is a need to explore formal methods (other than SDES/PDES) for multiprocessor system simulation. Moreover, any such description should ideally be used to create/synthesise parallel multithreaded simulators in order to utilise current multiprocessor simulation hosts.

In this article, we present our ongoing work on the development of a modular, object oriented, cycle-accurate and multithreaded multiprocessor simulator based on the formal method of communicating sequential processes (CSP) (Hoare, 1978). CSP is a process calculus with well-defined formal semantics of concurrency and communication. CSP and CSP-based languages have been previously used to describe and model asynchronous computing architectures (Theodoropoulos, 1997; Chien et al., 1995). As opposed to these approaches, our contribution in this work is to use the CSP process calculus to model a synchronous processor pipeline and a synchronous multiprocessor system. Furthermore, we are able to model processes at various levels of thread granularity, namely, functional level, nano-thread-level and kernel-level threads using a simulation library called Kent C++CSP (Brown and Welch, 2003; Brown, 2004). As a result, we are not only able to

describe the microarchitecture of an in-order processor and the system architecture of a multiprocessor system in terms of the CSP primitives, but we are also able to translate our description into an actual parallel, multithreaded simulator for multiprocessor systems.[1]

This article is organised as follows. Section 2 describes prior work in multiprocessor simulation. Sections 3 and 4 introduce definitions and relations eventually leading to the basis of the mathematical intuition behind using CSP for cycle-accurate simulation of multiprocessors. Section 5 briefly describes the process calculus of CSP, its mathematical terminology and corresponding language primitives available in Kent C++CSP library (Brown and Welch, 2003). Section 6 describes our microarchitectural simulator and the multiprocessor simulator. Section 7 describes the experimentation methodology and applications. Section 8 outlines the results and discusses the results of the simulation while Section 9 summarises the article and presents the conclusions. The results indicate that our simulator's performance is dependent on the number of processor cores on the simulation host. Thus, our simulator running on a simulation host with 'n + 1' cores shows a performance improvement of 30–40% over the simulator running on an 'n' core simulation host.

2 Survey of existing multiprocessor simulators

Much research already exists in the field of multiprocessor simulation using SDES/PDES. The RSIM (Ranganathan et al., 1997) simulation environment provides great flexibility in the configuration of the individual processors in a simulated multiprocessor. However, RSIM uses user-level threads – an artefact derived from its process-oriented, discrete-event simulator YACSIM (Jump, 1993). As a result, the simulation itself runs on a uniprocessor. The MINT multiprocessor simulator (Veenstra and Fowler, 1994) uses a similar approach to RSIM and was subsequently modified to simulate a chip multiprocessor (CMP) (Krishnan and Torellas, 1998). Simics (Magnusson et al., 2002) is a popular commercial multiprocessor simulator currently used in the academia and industry. Its methodology involves simulating a multiprocessor system by simulating each processor in a round-robin fashion. Each processor is simulated for a given number of cycles specified by a variable called 'cpu-switch-time'. This variable allows the coarseness in thread interleaving to be scaled. However, setting the 'cpu-switch-time' to a large value can have significant effect when simulating multithreaded applications with contended locks (Wallin et al., 2005). As a result, derived simulators such as VASA (Wallin et al., 2005) typically set the value of this variable to one. Other simulators used in the academia follow a similar round-robin simulation methodology (GxEmul, 2007) while simulating a multiprocessor system. None of the simulators mentioned above use parallel simulation approaches.

On the other hand, Wisconsin Wind Tunnel (WWT) (Reinhardt et al., 1993) and its successors (Mukherjee et al., 2000; Hill et al., 1996) report significant to almost linear speed-up by parallelising the simulation. Similar results are also reported by George and Cook (2002), George et al. (1999) and Chidester and George (2002). The WWT II simulates a parallel, ccNUMA system on various parallel systems connected using Myrinet (2009). It uses synchronised active messages (SAM) (Hill et al., 1996) to communicate between the host nodes for parallel simulation. WWT II also uses direct

execution to run orders of magnitude faster than pure interpreted software simulation (Mukherjee et al., 2000; Reinhardt et al., 1993). However, WWT II does not allow changes in the processor models and other architectural parameters such as issue widths, speculative memory accesses and out-of-order execution (Chidester and George, 2002). Most parallel simulators have been built using specialised programming models for distributed computing such as SAM and MPI. While SAM is not portable to non-SPARC simulation hosts, MPI suffers serious performance degradation on multicore shared memory architectures as it maps each node of computation to an OS process.

There have also been a very few multiprocessor simulators based on formal methods other than SDES/PDES. Zhu et al. (2004) used operation state machine (OSM) to represent concurrency in the system architecture of multiprocessor systems as well as the microarchitecture of individual processors. Furthermore, they showed how the OSM specification can be used to synthesise cycle-accurate simulators for multiprocessor System-on-a-Chip (MPSoC). Gonvindarajan et al. (1997) used timed-Petri nets for the simulation of multithreaded multiprocessor systems. Jalabert et al. (2004) introduced a SystemC (2009) based description tool for the synthesis of a multiprocessor SoC simulator. This tool also encouraged the reuse of simulation models by providing a library of parametrised on-chip communication architecture components. However, it is important to note that each of these prior approaches eventually synthesised a sequential or cooperatively multithreaded simulator of multiprocessor systems or MPSoC.

One can derive two deductions based on these prior approaches. First, parallel/multithreaded simulation can be used for fast and scalable simulation of multiprocessor systems. Second, multiprocessor system design can be made tractable with formal methods of design capture. While most prior approaches have used SDES/PDES with cooperative multithreading, relatively few have explored other formal methods such as Petri nets and OSM to formally describe multiprocessor systems. Additionally, prior approaches using formal methods have eventually synthesised sequential or cooperatively multithreaded simulators for multiprocessor systems or MPSoC. As opposed to the prior research, we not only describe a multiprocessor system using the formal method of CSP but also show how this description can be used to create a parallel, multithreaded simulator of multiprocessor systems. Moreover, though CSP has been traditionally used to describe asynchronous processor architectures, our contribution in this work is to demonstrate how CSP can be used for high-level design of synchronous multiprocessor systems and the creation of corresponding multiprocessor simulator.

3 Associating time-driven simulation methodology with CSP

In this section, we present the denotational framework of the tagged signal model (Lee and Vincentelli, 1996). We use this meta-model to describe how untimed-CSP can be used for time-driven (cycle-accurate) simulation of multiprocessor systems.

Let T be a set of *Tags* and V be a set of *Values*. Based on these, the following definitions are provided (Lee and Vincentelli, 1996).

- Event (e): e is a pair of a tag and a value, i.e.,
  $\{e=(x,y) | x \in T \wedge y \in V\} \Rightarrow e \in (T \times V)$.

- Signal (s): s is a set of events, i.e., $s \subseteq (T \times V)$. s can also be a partial function from T to V and if it is a partial function then for any $[e_x = (t, v_x)] \in s$ and $[e_y = (t, v_y)] \in s \Rightarrow (v_x = v_y)$.
- Trace (S): A set of signals processed by the system is called a trace, i.e., $S = 2^{(T \times V)}$. A set of N signals is denoted by S^N.
- Empty signal: A signal with no events is called an empty signal and is denoted by λ. This signal is just like any other signal such that $\lambda \cup s = s$. Here, the $\cup$ means a pointwise union of sets. A set of empty signals is represented by Λ such that $\Lambda \cup S = S$.
- Bottom ($\perp$): In some models of computation, there is a need to indicate a lack of value in an event. This is represented by $\perp$.
- Process (P): In the tagged signal model, a process is defined as an entity responsible for producing a subset of the trace (S^N), i.e., $P \subseteq S^N$.
- System (Q): A system Q is a collection of processes such that

  $$Q = \left\{ P_i \mid P_i \subseteq S^N \right\}.$$

  where, i is a suffix uniquely identifying a process in the system Q.

Based on these definitions, we now define the following relations.

- $<$: This is an irreflexive, antisymmetric and transitive relation between every two members of a set of tags (T). This relation imposes total ordering on the set of tags in the system.
- $\leq$: This is a reflexive, antisymmetric and transitive relation between any two members of a set of tags (T). This relation imposes partial ordering on the set of tags in the system.

We now define timed system as well as untimed system and explain the model of time-driven simulation and CSP. These definitions will be used to describe the basis of using untimed-CSP for time-driven (cycle-accurate) simulation in Section 4.

Definition 3.1: Timed system: Any tagged system Q where the set of tags (T) is totally ordered is called a timed system. Thus, for any two t_x and t_y, either $t_x < t_y$ or $t_y < t_x$. This condition ensures causality in a timed system.

Definition 3.2: Synchronous events: Two events $e_x = (t_x, v_x)$ and $e_y = (t_y, v_y)$ are said to be synchronous iff $t_x = t_y$.

Definition 3.3: Synchronous signals: Two signals s_x and s_y are said to be synchronous iff all events in s_x are synchronous with all events in s_y.

Definition 3.4: Time-driven system (or synchronous system): A system Q producing a trace S^N is called synchronous iff $\forall s_x$ and $\forall s_y \in S^N$ and s_x is synchronous with s_y.

Definition 3.5 Untimed system: Any tagged system Q where the set of tags (T) is partially ordered is called an untimed system. Thus, for any two t_x and t_y, either $t_x \le t_y$ or $t_y \le t_x$. This relationship introduces non-determinism in the system and the system cannot ensure causality. Hence, a similar concept to causality provided by the monotonicity condition (Lee and Vincentelli, 1996) is used to describe an untimed system of CSP.

Definition 3.6: CSP-based system: Any CSP-based system Q is defined as having the following properties:

1 the events in each signal are totally ordered, but

2 ordering between events in different signals can only be ensured between a group of processes synchronising at the rendezvous point. Each rendezvous point will be represented by a set of conditions, such that

$$T(s_x) = T(s_y) = T(s_z) = \ldots$$

where $T(s_x)$, $T(s_y)$ and $T(s_z)$ represent the tags of the signals s_x, s_y and s_z, respectively.

Thus, a CSP-based system without completely matched rendezvous points is an untimed system. In the following section, we describe the basis of using untimed-CSP for time-driven (cycle-accurate) simulation.

4 Basis of using untimed-CSP for time-driven (cycle-accurate) simulation

As we can see from Definition 3.6, any two signals s_x and s_y that are not associated with a rendezvous point are partially ordered. Hence, by Definition 3.5, process P_x described by s_x and process P_y described by s_y are partially ordered in execution time. This implies that any two such processes P_x and P_y have no causal relationship between them, and hence the system is an untimed system. However, if we are to have a timed system, from Definition 3.1, it is necessary for all signals in the time-driven system to have total ordering among them at each clock 'tick'. This implies that two communicating processes P_x and P_y should synchronise at every clock cycle. Thus, even if process P_x does not wish to communicate its results at any given clock cycle, it should produce an event with the value of '⊥' to indicate an absence of value. This ensures that our system (explained in detail in Section 6) is a timed-system as per Definitions 3.1, 3.2, 3.3 and 3.4. A similar concept is found in all CSP-oriented languages, where the absence of an event at a particular clock is well-defined (Lee and Vincentelli, 1996).

5 Overview of CSP and C++CSP library

The previous section demonstrated the mathematical basis required for using CSP for time-driven (cycle-accurate) simulation. This section explains the process algebra of CSP and the corresponding primitives that represent the operators of CSP (e.g., assignment, sequential and parallel composition of processes, and selection) in the C++CSP library.

In CSP, concurrent activities are modelled as processes that communicate and coordinate their activities using channels. These channels represent mediums through which messages are sent and received.

5.1 Processes

Processes represent concurrent modules that do not share variables but communicate by sending and receiving messages. In C++CSP library, processes are modelled by inheriting from the CSProcess or the ThreadCSProcess class. Processes derived from the CSProcess class run in the process space of the process that created them. As a result, they represent user-level threads. Processes derived from ThreadCSProcess are kernel-level threads and are scheduled for execution by the OS. Although they are scheduled by the OS, they are synchronised with respect to each other by the C++CSP kernel. Additionally, if required, processes derived from CSProcess can also be made to run in separate kernel-level threads by executing them in parallel using the RunInParallel (processes) helper function.

5.2 Channels and ports

A port on process X is connected to a port on process Y using a channel. Each process can read and write to the channels using the reading or writing end, respectively. The reader end is obtained via a call to the channels' '*reader()*' method while the writer end is obtained via a call to the channels' '*writer()*' method. C++CSP provides the following types of channels:

- One2OneChannels: These are channels between a pair of processes.
- Any2OneChannels: These are channels from a given set of processes to an individual process.
- Any2AnyChannels: These are channels between two groups of processes.

The channels described above are not 'buffered'. This means that the writer is blocked until the reader has read the channel. All of the channels described above also have 'buffered' counterparts, where the writer can continue writing to the channel until the buffer becomes full. After that, the writer is blocked until the reader has read from the buffer.

5.3 Assignment

CSP overloads redirect operators to facilitate communication via channels. '$Chan >> a$' represents reading from the channel into variable a. Similarly, '$chan << a$' represents writing to the channel from a variable a.

5.4 Sequential and parallel composition of processes

CSP provides sequential as well as parallel composition of processes. The operator ';' represents the 'in sequence' operator. Thus, the notation $[P_i; P_j]$ is used to represent sequential composition. To represent parallel composition of processes, CSP provides the '||' operator. Hence, $[P_i \| P_j]$ indicates that processes P_i and P_j are to be run in parallel. In C++CSP library, these operators are modelled as RunInSequence(processes) and RunInParallel(processes) helper functions, respectively.

5.5 Selection

In CSP, selection is represented as $[G_1 \to P_1 \| G_2 \to P_2 \| \ldots \| G_n \to P_n]$ where, $G_1, G_2, \ldots, G_n$ denote the predicates or 'guards' and $P_1, P_2, \ldots P_n$ denote concurrent processes. Thus, the above statement can be interpreted as 'wait (represented by '$\to$') until guard G_i are true and then execute corresponding processes P_i in parallel'. Repetition is represented in CSP as '$*[P_i]$'. This statement is interpreted as 'execute P_i forever'. In C++CSP library, selection, wait and repetition are modelled using the alternative class, guard class and simple 'while' loops in the '*run()*' function, respectively.

6 Detailed methodology of a formal description of a multiprocessor system

CSP Description 1 is the formal description of a microarchitecture of individual processors as well as system architecture of a multiprocessor system. For each processor, we have modelled a five-stage in-order pipeline. Here, each stage of the in-order pipeline is mapped to a single process in CSP. These processes then communicate using the buffered channels described in Section 5.

Line 1 of this description shows that the multiprocessor system is timed with a timer input tm. After receiving the timer signal, the multiprocessor system executes the processes that model processors ($Proc_1 \ldots Proc_n$), crossbar (X_{bar}), memory controller (Mem_{Cntr}), devices (in this case, a single device) (D_1) and the console ($consl$). These processes are described to run in parallel as shown in line 1. Line 2 shows that each processor waits on the timer input and runs the instruction fetch stage (IF), decode stage (ID), execute stage (EX), memory stage (MEM) and the write-back stage (WB) in parallel.

CSP Description 1: a multiprocessor system in CSP

1	MP_{sys}	:=	$*\left[tm \rightarrow \left(Proc_1 \| .. \| Proc_4 \| X_{bar} \| Mem_{Cntr} \| D_1 \| consl\right)\right]$
2	$Proc_1,...,Proc_n$	:=	$*\left[tm \rightarrow \left(IF \| ID \| EX \| MEM \| WB\right)\right]$
3	$Pipe_{stage}$	:=	$*\left[RdInpSig; LoanInst; FuncRun; RetInst; WrOutSig\right]$
4	$FuncRun_{IFstage}$	:=	$PCWrite \rightarrow MemBranch$
5	$MemBranch$	:=	$\left[PCStep \rightarrow Mem\left(PC+4\right) \right\|$ $PCExp \rightarrow Mem\left(ExpVector\right) \|$ $PCBranch \rightarrow Mem\left(BranchAdd\right)\right]$
		....	
6	Mem_{Cntr}	:=	$*\left[RdInpSig; LoanInst; FuncRun; RetInst; WrOutSig\right]$
7	$FuncRun_{Mem_{Cntr}}$	:=	$\left[ServiceRequest\right]$
8	D_1	:=	$*\left[tm \rightarrow \left(RdInpSig; LoanInst; FuncRun; RetInst; WrOutSig\right)\right]$
9	$FuncRun_{D_1}$	:=	$\left[Access\right]$

Line 3 shows that when a pipeline stage is run, it results in the sequential execution of five processes (simulated using functions, hence called functional processes). These functional processes are ReadInputSignals ($RdInpSig$), LoanInstances ($LoanInst$), FunctionalRun ($FuncRun$), ReturnInstances ($RetInst$) and WriteOutputSignals ($WrOutSig$), which are explained in more detail in Section 6.1. Line 4 describes how the functionality of the instruction fetch stage can be represented in terms of CSP primitives. It indicates that when the PCWrite signal is received, the instruction fetch stage is run. Line 5 shows that after receiving the PCWrite signal, the instruction fetch stage requests an instruction at one of the following locations depending on additional input signals from the execute stage (EX):

- Program counter (PC) + 4, if the additional input signal is PCStep.
- Exception vector routine, if the additional input signal is PCExp.
- Branch address, if the additional input signal is PCBranch.

Lines 6 and 7 describe the functioning of the memory controller where the memory controller waits for input read/write signals from the processors. After requests are received from all processors, the memory controller model executes a functional process called '*ServiceRequest()*' to process the read/write requests. Lines 8 and 9 show that devices are activated by the timer signal. Each device model then executes a functional process called '*Access()*', which simulates the operation of the device.

6.1 Description of functions

Following is a more detailed description of the functions executed by each pipeline stage corresponding to the formal description outlined in CSP Description 1. Program 1 shows an example of the corresponding interface of the instruction fetch pipeline stage.

1 ReadInputSignals: This function reads input data from input channels.

2 LoanInstances: To reduce the overhead of instantiating channel data objects at every clock cycle, this method obtains the objects from a global object pool of channel data objects. Furthermore, the object reference is obtained and passed through the channels to reduce communication overhead of copying data between channel ends.

3 FunctionalRun: In C++CSP, all processes have to implement a virtual run function. This is the function where the behavior of any pipeline stage component is defined.

4 ReturnInstances: Channel objects that are loaned from the object pool by the writers are returned by the readers using this function. A call to this function ensures that the object pool is replenished and also avoids memory leaks.

5 WriteOutputSignals: This function is used to write the outputs to the required stages.

6 Run: This function is called once by the C++CSP simulation runtime when it launches the modelled component (such as the IF stage described in Program 1) as separate CSProcess or ThreadCSProcess.

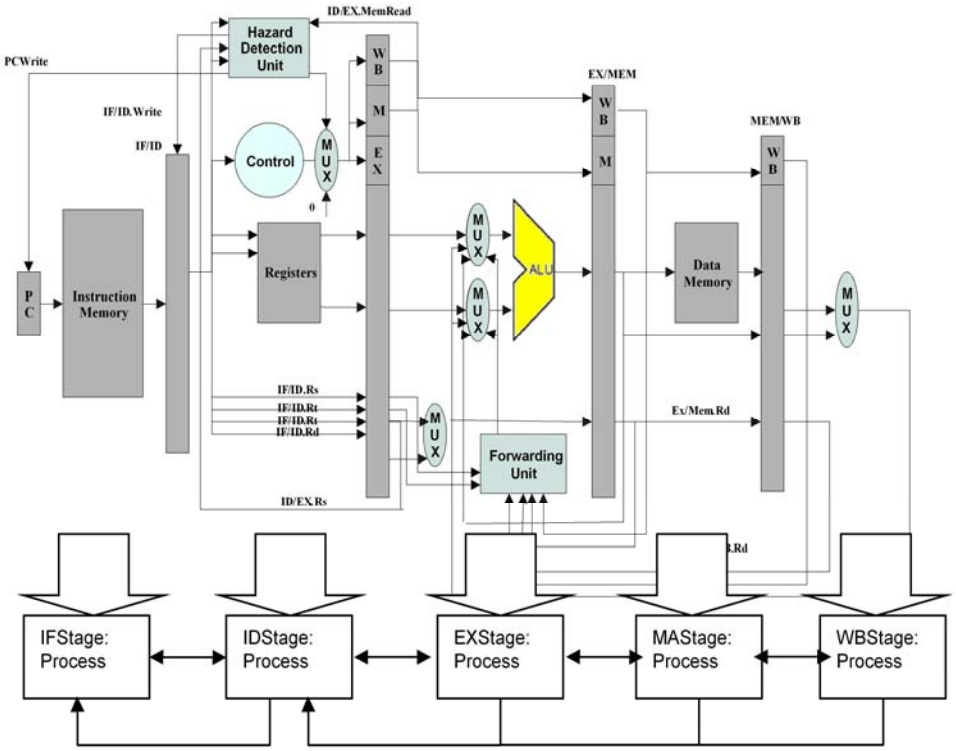
Program 1: class representing the instruction fetch stage

```
class TIFetch : public CSProcess
{
    public:
        TIFetch
        (Chanin<unsigned long int>& cinClock,
         AltChanin<T_ID2IF_ChannelData>& cinID2IFChannel,
         AltChanin<bool>& cinWB2IFChannel,
         Chanout<T_IF2ID_ChannelData>& coutIF2IDChannel,
        );
        ~TIFetch ( );
    protected:
        void ReadInputSignals ( );
        void LoanInstances ( );
        void FunctionalRun ( );
        void ReturnInstances ( );
        void WriteOutputSignals ( );
        void Run ( );
};
```

6.2 Microarchitectural components

Our microarchitectural simulator accurately models each of the microarchitectural components of a five-stage in-order pipeline shown in Figure 1. Our simulator contains the modular components of the following stages: instruction fetch stage, decode stage, execute stage, memory access stage and write-back stage.

Figure 1 Microarchitecture model of processors implemented in the simulator (see online version for colours)



We now briefly describe each of the pipeline stages of our simulator.

1 Instruction fetch stage: The components modelled include instruction memory, PC multiplexer and branch target buffer (BTB). The instruction memory and BTB are of customisable size. An example interface of the instruction fetch stage is shown in Program 1.

2 Decode stage: The decode stage contains an instruction decoder, a control unit, a hazard detection unit, an n-bit branch predictor, a sign extender, integer registers, as well as floating point registers.

3 Execute stage: Our simulator models the execute stage in great detail, which contains ALU control units, a floating point ALU, an integer ALU and a forwarding unit. Since we model an in-order processor, multicycle instructions such as floating point

add/divide will stall the pipeline for the given number of cycles. These instructions will produce results only when the period of time specified by their latency has been simulated in the pipeline.

4 Memory access stage: The memory access stage issues read/write requests to the memory controller which in turn controls the memory.

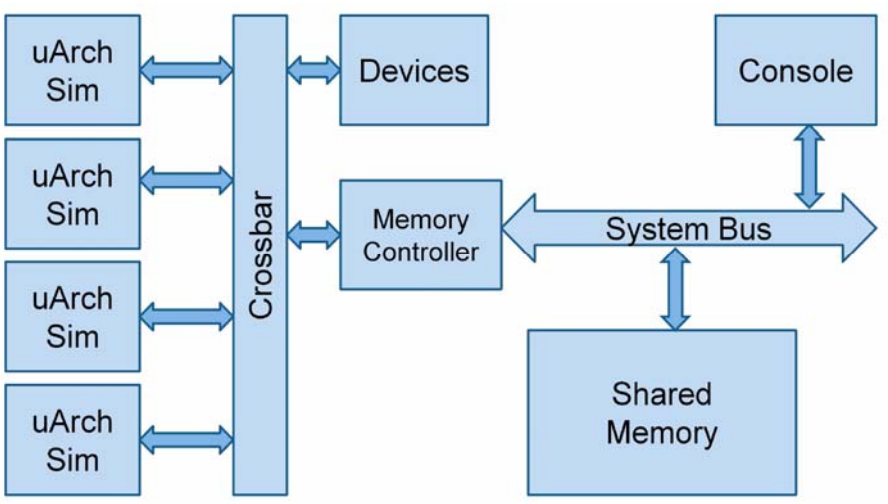
5 Write-back stage: This stage is responsible for writing the data back to the integer as well as floating point registers.

In our experiments (described later in Section 8), processes that model individual pipeline stages are derived from the CSProcess class. To run modelled components as user-level threads running within a single kernel-level thread, we use the RunInParallelOneThread (processes) helper function provided by the C++CSP library (Brown and Welch, 2003).

6.3 System architecture of a multiprocessor simulator

Figure 2 shows the system architecture modelled in our multiprocessor simulator. In this multiprocessor simulator, we have reused the microarchitectural processor model described in Section 6.2. Additionally, we constructed device models for the memory, console and the memory controller. To ensure consistency of memory in the multiprocessor model, individual processor models synchronise by rendezvous at the crossbar at each clock cycle. This means that a memory read/write request by a processor at time '*t*' is assumed to be an implicit rendezvous by the processor model at the crossbar. Furthermore, if the processor model is not making a memory read/write request to the crossbar at time '*t*', it should rendezvous at the crossbar by emitting a '⊥' signal at time '*t*'. Note that, the crossbar and memory controller are modelled functionally and do not represent any real crossbar designs.

Figure 2 Multiprocessor system architecture modelled in the simulator (see online version for colours)



7 Experimentation methodology

7.1 Experimentation environment

We have performed our experiments on two different simulation hosts. Our first simulation host is an AMD Athlon 64-bit machine with 2.21 GHz dual-core 4200+ processor. It has 4 GB of RAM and runs the 64-bit Windows XP SP2 operating system. Our second simulation host is a 2.4 GHz Intel Core 2 quad-core with 4 GB of RAM running the 64-bit Windows XP SP2 operating system.

7.2 Applications

For our experiments, we have considered two applications. Our first application is an Insertion Sort algorithm (IS in Figures 3, 4 and Tables 1, 2 and 3) that sorts 1000 64-bit integers. For the microarchitectural simulation, we simulated a single processor executing 365,671 instructions corresponding to an insertion sort program. In case of multiprocessor simulation, each processor executed 365,671 instructions corresponding to an independent insertion sort program. Our second application is an FIR filtering (FIR in Figures 3, 4 and Tables 1, 2 and 3) in a multiband processing system used commonly in hi-fi systems. In the case of the microarchitectural simulation, we simulated a single processor executing 654,072 instructions corresponding to a single filter program. In case of the multiprocessor simulation, the system used four different filters each running on a separate processor (executing 654,072 instructions) to process a 4-tone input signal into four distinct frequency signals. This filtering application processes 512 input samples. In Figures 3 and 4, 'IS on 2' represents the results of running insertion sort on a dual-core host machine. Similar terminology is used to represent the results for IS/FIR filtering on a dual-core and a quad-core host in Figures 3 and 4.

8 Experiments and results

8.1 Microarchitectural simulation

The goal of this experiment was to test the simulation speed obtained by allowing individual pipeline stages to run as user-level threads versus kernel-level threads. In a multiprocessor machine, using kernel-level threads for modelling the pipeline should produce faster simulation speeds because the OS can concurrently run simulated pipeline stages in kernel-level threads on multiple processor cores. We evaluated this hypothesis using the applications described in Section 7.2.

The results of this experiment are presented in Table 1 and Figure 3. Table 1 shows the number of instructions executed by the CPU and the corresponding time taken when five pipeline stages and four stage registers are simulated as user-level threads versus kernel-level threads. Our results indicate that simulating individual pipeline stages as kernel-level threads in a microarchitectural simulation is faster than simulating them using user-level threads, giving 30–40% speedup per each additional core.

Table 1 Microarchitectural simulation results of one processor using user-level vs. kernel-level threads

Appl.	*Num. of instr.*	*Proc. cores*	*User-level threads (sec)*	*Instr/sec*	*Kernel-level threads (sec)*	*Instr/sec*	*Speedup (%)*
IS	365,671	2	112	3264.9	78	4688.1	30.3
		4	122	2997.3	49	7462.7	59.8
FIR	654,072	2	200	3270.4	127	5150.2	36.5
		4	213	3070.8	79	8279.4	62.9

Table 2 Execution time of the multiprocessor simulator when simulating an 'n' processor model

Appl.	*Total number of instr. simulated*	*Number of processor cores available on host*	*Execution time of simulator (in seconds) for*			
			1-proc. model	*2-proc. model*	*4-proc. model*	*8-proc. model*
IS	# of processors modelled × 365, 671	2	112	157	312	639
		4	121	161	195	398
FIR	# of processors modelled × 654,072	2	200	237	451	927
		4	217	250	317	701

Table 3 Performance of multiprocessor simulator (in terms of IPS) when simulating an 'n' processor model

Appl.	$NProc_h$*	*Aggregate instructions executed per second (IPS)*			
		1-proc. model		*2-proc. model*	
		$NSIM^{**}_{Instr}$	*IPS*	$NSIM^{**}_{Instr}$	*IPS*
IS	2	365,671	3264.9	731,342	4658.2
	4		3022.1		4542.5
FIR	2	654,072	3270.4	1,308,144	5519.6
	4		3014.2		5232.6
Appl.	$NProc_h$*	*4-proc. model*		*8-proc. model*	
		$NSIM^{**}_{Instr}$	*IPS*	$NSIM^{**}_{Instr}$	*IPS*
IS	2	1,462,684	4688.1	2,925,368	4578.0
	4		7500.9		7350.2
FIR	2	2,616,288	5801.1	5,232,576	5644.6
	4		8253.3		7464.4

Notes: *Number of processor cores available on the host.
**Aggregate (or total) number of instructions executed during the simulation of the application.

Figure 3 Performance of microarchitectural simulator (in IPS) using user-level vs. kernel level threads (see online version for colours)

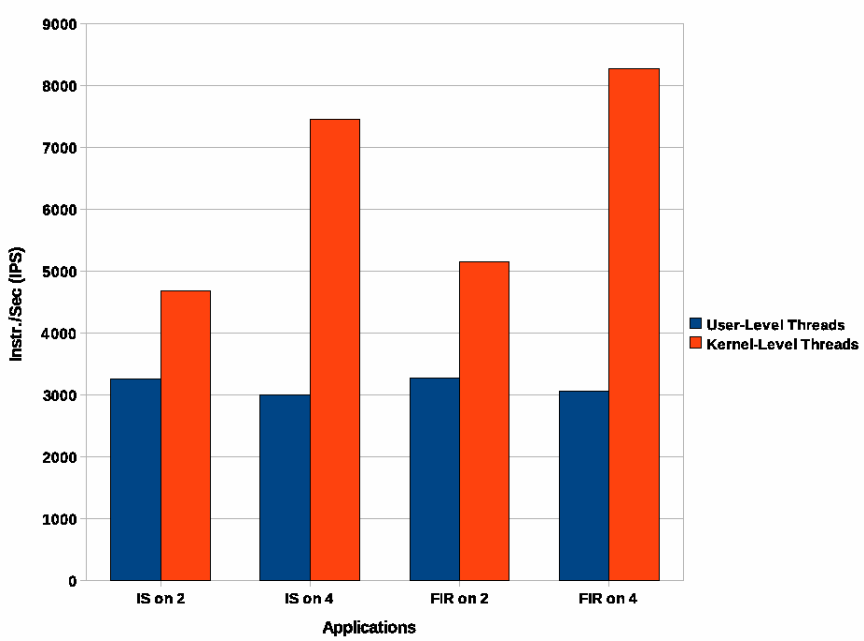
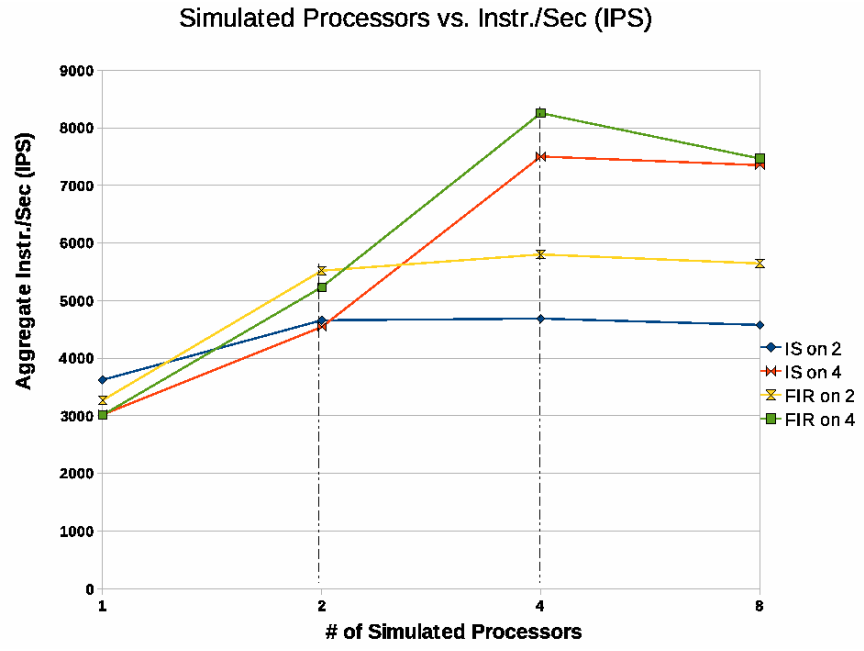


Figure 4 Performance of multiprocessor simulator (in IPS) for an 'n' processor model (see online version for colours)



8.2 Multiprocessor simulation

The goal of this experiment was to measure the simulation speed obtained by simulating each processor as a kernel-level thread while individual pipeline stages within each processor are simulated as user-level threads. We expected that the simulation should show speedup when the number of processors in the multiprocessor system being simulated is less than or equal to the number of processor cores available on the simulation host. We also expected that when the number of processors being simulated exceeds the number of processor cores available on the host, the total number of instructions executed per second by the machine (i.e., the sum of number of instructions executed by each simulated processor) remain constant. We evaluated our hypothesis using the applications described in Section 7.2.

Table 2 shows the time taken by the simulator to execute the applications described in Section 7.2. The second column of this table shows the total number of instructions executed by the applications in an '*n*' processor simulated system. The third column of this table shows the actual number of processor cores available on the host. The remaining columns show the execution times of the simulator when it is simulating an '*n*' processor model.

Table 3 shows the performance of our simulator in terms of number of simulated instructions executed per second. The second column of this table shows the number of processor cores available on the host ($NProc_h$). The remaining columns show the performance of the simulator in terms of the aggregate (or total) number of simulated instructions executed per second (IPS) for the corresponding '*n*' processor model. The aggregate number of instructions ($NSIM_{Instr}$) executed in the simulator is obtained by multiplying the number of instructions executed by a single modelled processor with the total number of modelled processors ('*n*').

Figure 4 summarises the results of Tables 2 and 3, and shows the aggregate number of instructions executed per second by the simulated processors when the experiments are run on a dual-core host (IS/FIR on 2) and when they are run on a quad-core host (IS/FIR on 4) machine. In Table 3 and Figure 4, the aggregate number of instructions executed per second by the simulator is obtained by dividing '$NSIM_{Instr}$' by the execution time of the simulator (as shown in columns 4–7 of Table 2), respectively.

From Tables 2 and 3 and Figure 4, we can see that as the number of kernel-level threads containing the processor models approaches the number of processor cores on the simulation hosts, speedup of up to 30–40% can be obtained. However, if the number of simulated processors goes beyond the number of processor cores available on the simulation host, the total number of instructions executed per second by the simulator does not increase but remains more or less constant.

9 Conclusions and summary

In this article, we showed how the formal method of CSP can be used to describe a multiprocessor system and create its corresponding multithreaded multiprocessor simulator. The experimental results indicate that the simulator shows scalability of 30–40% up to the number of processor cores available on the simulation host. However, an interesting observation is that the simulator executes more or less a constant number of

instructions per second as the number of kernel-level threads modelling pipeline stages (in microarchitectural simulation) or individual processors (in multiprocessor simulation) exceeds the number of processor cores available. Generally, parallel simulations show degradation in performance when the number of processes surpasses the total number of processor cores available on the simulation host. This indicates that designing simulators with a combination of user-level threads and kernel-level threads using CSP may provide a scalable speedup for multiprocessor simulation on multicore or multiprocessor simulation hosts.

References

Brown, N. (2004) 'C++CSP networked', *Communicating Process Architectures*.

Brown, N. and Welch, P. (2003) 'An introduction to the Kent C++CSP Library', *Communicating Process Architectures*.

Chidester, M. and George, A. (2002) 'Parallel simulation of chip-multiprocessor architectures', *Simulation*, ACM Trans. Model. Comput., pp.176–200.

Chien, C., Franklin, M., Pan, T. and Prabhu, P. (1995) 'ARAS: asynchronous RISC architecture simulator', Paper presented at *2nd Working Conference on Asynchronous Design Methodologies*, pp.210–219.

Emer, J., Ahuja, P., Borch, E., Klauser, A., Luk, C., Manne, S., Patil, H. Wallace, S., Binkert, N., Espasa, R. and Juan, T. (2002) 'Asim: a performance model framework', *IEEE Computer*, Vol. 35, No. 2, pp.68–76.

George, A. and Cook, S. (2002) 'Distributed simulation of parallel DSP architectures on workstation clusters', *Simulation*, Vol. 12, No.3, pp.94–105.

George, A., Fogarty, R., Markwell, J. and Miars, M. (1999) 'An integrated simulation environment for parallel and distributed system prototyping', *Simulation*, Vol. 75, No. 5, pp.283–294.

Govindarajan, R., Suciu, F. and Zuberek, W. (1997) 'Timed Petri net models of multithreaded multiprocessor architectures', Paper presented at the *Int. Workshop on Petri Nets and Performance Models*, pp.153–162.

GxEmul Instruction Set Simulator (2007) Available at http://gavare.se/gxemul/ (accessed July 2007).

Hill, M., Larus, J. and Wood, D. (1996) 'Parallel computer research in the Wisconsin Wind Tunnel project', Paper presented at *NSF Conference on Experimental Research in Computer Systems*.

Hoare, C. (1978) 'Communicating sequential processes', *Communications ACM*, Vol. 21, pp.666–677.

Jalabert, A., Murali, S., Benini, L. and Micheli, G. (2004) 'xpipescompiler: a tool for instantiating application specific networks on chip', Paper presented at the *2004 Design Automation and Test in Europe*, pp.884–889.

Jump, J. (1993) *YACSIM Reference Manual, Version 2.1*, Rice University.

Krishnan, V. and Torellas, J. (1998) 'Hardware and software support for speculative execution of sequential binaries on a chip multiprocessor', Paper presented at *Int'l Conf. on Supercomputing*, pp.85–92.

Lee, E. and Vincentelli, A. (1996) 'Comparing models of computation', Paper presented at the *Int'l Conf. on Computer-Aided Design*, pp.234–241.

Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A. and Werner, B. (2002) 'Simics: a full system simulation platform', *IEEE Computer*, Vol. 35, No. 2, pp.50–58.

Mukherjee, S., Reinhardt, S., Falsafi, B., Litzkow, M., Hill, M., Wood, D., Huss-Lederman, S. and Larus, J. (2000) 'Wisconsin Wind Tunnel II: a fast and portable parallel architecture simulator', *IEEE Concurrency*, Vol. 8, No. 4, pp.12–20.

Myricom Page for Myrinet (2009) Available at http://www.myri.com/myrinet/overview/ (accessed July 2009).

Perez, D., Mouchard, D. and Temam, O. (2004) 'MicroLib: a case for the quantitative comparison of microarchitecture mechanisms', Paper presented at the *37th Int'l Symp. on Microarchitecture*, pp.43–54, IEEE Computer Society.

Ranganathan, P., Pai, V. and Adve, S. (1997) 'RSIM: an execution-driven simulator for ILP-based shared memory multiprocessors and uniprocessors', *IEEE Tech. Comm. On Computer Architecture (TCCA) Newsletter*.

Reinhardt, S., Hill, M., Larus, J., LeBeck, A., Lewise, J. and Wood, D. (1993) 'The Wisconsin Wind Tunnel: virtual prototyping of parallel computers', Paper presented at the *ACM Sigmetrics Conf. on Measurement and Modeling of Computer Systems*, pp.48–60.

Skadron, K., Martonosi, M., August, D., Hill, M., Lilja, D. and Pai, V. (2003) 'Challenges in computer architecture evaluation', *IEEE Computer*, Vol. 36, No. 8, pp.30–36.

SystemC 2.0 (2009) Available at http://www.systemc.org (accessed January 2009).

Theodoropoulos, G. (1997) 'OCCARM: an occam parallel simulation model of the AMULET1 asynchronous microprocessor (OCCARM)', Paper presented at *23rd IEEE Euromicro Conference on New Frontiers of Information Technology*.

Vachharajani, M., Vachharajani, N., Penry, D., Blome, J. and August, D. (2002) 'Microarchitectural exploration with Liberty', Paper presented at the *35th Ann. Int'l Symp. on Microarchitecture*, IEEE CS Press, pp.271–282.

Veenstra, J. and Fowler, R. (1994) 'MINT tutorial and user manual', *Technical Report 452*, Department of Computer Science, University of Rochester.

Wallin, D., Zeffer, H., Karlsson, M. and Hagersten, E. (2005) 'VASA: a simulator infrastructure with adjustable fidelity', *Parallel and Distributed Computing and Systems*, pp.259–268.

Zhu, X., Qin, W. and Malik, S. (2004) 'Modeling operation and microarchitecture concurrency for communication architectures with application to retargetable simulation', Paper presented at the *International Conference on Hardware/Software Codesign and System Synthesis*, pp.66–71.

Notes

1 Kent C++CSP threading library provides primitives that can be used to map physical processes at various levels of thread granularity such as functional-level, nano-thread-level, user-level threads, kernel-level threads, as well as networked process level. Although this simulation library allows processes to be modelled as 'network processes' running on a cluster of simulation hosts (Brown, 2004), in this work we have limited the modelling of physical processes to functional-level, nano-thread-level, user-level and kernel-level threads.