2008 11th IEEE High Assurance Systems Engineering Symposium

A Fast Performance Analysis Tool for Multicore, Multithreaded Communication Processors

Hun Jung, Miao Ju and Hao Che
Department of Computer Science and Engineering
University of Texas at Arlington
{hjung, miaoju, hche}@ uta.edu

Zhijun Wang
Department of Computing
Hong Kong Polytechnic University, Hong Kong
cszjwang@comp.polyu.edu.hk

Abstract

To allow fast communication processor (CP) performance testing of task-to-CP-topology mapping, we propose a fast CP simulation tool[*] *with a few novel ideas that make it generic, fast, and accurate. Our major goal is to focus on modeling features common to a wide variety of CP architectures and incorporate relevant CP specific features as plug-ins. This tool not only allows user-defined packet arrival processes and code path mixtures to be tested, but also provides a way to allow the maximum sustainable line rate to be quickly estimated. Case studies based on a large number of code samples available in IXP1200/2400 workbenches show that the maximum sustainable line rates estimated using our tool are consistently within 6% of cycle-accurate simulation results. Moreover, each simulation run takes only a few seconds to finish on a Pentium III PC, which strongly demonstrates the power of this tool for fast CP performance testing.*

1. Introduction

Programmable, multicore, multithreaded communication processors (CPs) are increasingly being adopted in router interface cards and server interfaces for real-time, parallel packet processing at high speed. This type of processor is generally built based on either specially designed processor cores, as in the case of network processors (NPs) (e.g., [15]), or general purpose processor cores (e.g., [1]). A critical challenge using multicore, multithreaded CPs for packet processing is in its initial programming phase, a decision must be made as to how to partition packet processing tasks and map them to one of many possible core topologies to achieve the desired performance in terms of throughput, delay, loss, and power and memory consumption. A misjudgment in this phase may lead to expensive redesign in various programming phases or poor performance. Hence, it is imperative to develop effective tools to aid the decision making process in the initial CP programming phase.

However, such a tool must meet the following three stringent requirements to be useful. First, the tool must be fast enough to allow a potentially large number of partition and mapping choices to be tested in a reasonable amount of time. Second, the performance data it provides must be reasonably accurate[1]. Third, the tool must not assume the availability of the executable program as input for testing, which has yet to be developed in subsequent programming phases. These three requirements pose significant challenges to the development of such a tool. On one hand, to provide reasonably accurate performance data, the tool must take into account processing details that may contribute significantly to the system-level performance, such as the thread level activities, and memory and I/O resource contentions at the thread, core, and system levels. On the other hand, indiscriminately including various processing details in such a tool can quickly make the tool heavyweight, in terms of both time and space complexities. Additional problem is that without the executable program as input, the instruction level effects that may impact the overall system performance cannot be directly accounted for, such as instruction level pipeline aborts and certain thread scheduling disciplines involving multiple issues, such as superscalar and simultaneous multithreading. Then the issue to be considered is whether it is possible to develop a tool for CP performance analysis that meets all the above three

[*] This work is partially supported by the RGC of Hong Kong Polytechnic University under Project Code A-PH43.

[1] The performance data within 10-20% of actual performance in this programming phase should be considered reasonably accurate. This is because such performance data is obtained using a piece of pseudo code with normally inaccurate instruction count as input, not the executable program itself. The program developed in later phases can generally be fine tuned to compensate for such loss of accuracy.

1530-2059/08 $25.00 © 2008 IEEE
DOI 10.1109/HASE.2008.18

requirements. Our initial exploitation of this issue in [16] provides encouraging evidence, indicating that such a tool could be developed. In [16], we developed a fast, packet-latency estimation algorithm based on a simple, generic processor and code path model, overlooking many processing details in order to meet the above three requirements. It turns out that this algorithm provides fairly accurate performance data (within 16% of cycle-accurate simulation data) for almost all the sample cases available in IXP1200/2400 workbenches provided by Intel Internet Exchange Architecture (IXA) Software Development Kit (SDK). The only exception is the *Packet Count* sample case in IXP1200. In this case, there is a dominant effect that cannot be captured by the algorithm, i.e., the serialization effect caused by a critical section.

Motivated by the work in [16], in this approach, we present a novel design methodology and an initial prototype of a CP simulation tool based on this method. This tool meets all three requirements previously described, and hence, can be used as an analysis tool for performance testing of task-to-CP-topology mapping in the initial CP programming phase. Moreover, this tool is generic, in the sense that it can be adapted to various CP architectures. As a result, it can also be used for large design space exploitation. The current tool has focused on the following performance measures: throughput, delay, and loss. Other performance measures, such as power and memory consumptions, will be incorporated in the future.

The rest of this paper is organized as follows. Section 2 discusses the related work. Section 3 describes the methodology and the tool. Section 4 provides the test results for the tool. Finally, Section 5 presents the conclusions and future work.

2. Related Work

There are a vast number of processor simulation tools available [2-15, 18, 19] (e.g. cycle-accurate, allowing detailed timing analysis, and providing primitives for flexible component modeling). Particularly relevant to our work are the Network Processor analysis tools (e.g., [6-15]).

Most NP simulation software (e.g., [6-9]) aims at providing rich features to allow detailed statistical or per-packet analysis, which is useful for program fine tuning, rather than fast NP performance testing. Even for the most lightweight NP simulator described in [6], it is reported that to simulate one second of hardware execution, it takes 1 hour on a Pentium III 733 PC. Moreover, it assumes the availability of the executable program or microcode as input for the simulation. On the other hand, the algorithms for data path functions to NP core topology mapping (e.g., [11] [13] [14]) are generally fast, but at the expense of having to overlook many essential processing details that may have an impact on the overall system performance. To make the problem tractable, a common technique used in these approaches is to partition the data path functions into tasks and each task is then associated with one or multiple known resource demand metrics, e.g., the core latency and program size. Then an optimization problem under the demand constraints is formulated and solved to find a feasible/optimal mapping of those tasks to a pipelined/parallel core topology. Since the actual resource demand metrics for each task are, in general, complex functions of mapping itself, and are a strong function of the number of threads and thread scheduling discipline in use at each core, these approaches cannot provide mappings with high accuracy. Although the approach in [13] accounts for certain multithreading effect, it only works for a single memory access and under a coarse-grained scheduling discipline.

From previous work, all the above existing tools lack of the following feature needed to address the problem at hand. Namely, they are not concerned with how the packet input process should be generated, which is assumed to be provided by the user, rather than part of the tool design. Given that there are virtually unlimited numbers of possible packet arrival processes and mixtures of packets carrying various code paths, it is a daunting task for a user of an existing tool to decide what input processes should be tested, or what statistics should be collected. Note that no matter how fast a simulation tool may be, the simulation time is guaranteed to be prohibitively long if the goal is to perform exhaustive statistical analysis, based on the simulation data collected from a large number of packet arrival processes and mixtures of code paths.

3. Simulation Tool

The design objectives of the tool are: (i) to be generic and adaptable to a wide range of CP architectures; and (ii) to meet all three requirements described in the introduction section.

This section is organized as follows. In section 3.1, we first introduce the design methodology of the tool. In section 3.2, we describe the simulation architecture on the basis of this design methodology. Finally, in section 3.3, we provide a means to allow sustainable line rate for a CP to be quickly estimated.

3.1 Design Methodology

The following three key ideas underlay the CP simulation tool development.

Focus on emulating common CP features while taking the relevant performance impacts of CP-specific features into account through user provided models. Our approach attempts to strike a balance between complexity and simplicity by adopting a hybrid simulation-and-modeling based approach. Specifically, our approach focuses on faithfully emulating important features common to a wide variety of CP architectures (e.g., multithreading and multicore) and account for the relevant performance impacts of CP-specific features (e.g., I/O interface, cache, memory, memory controller, and bus architectures) through user provided models, called plug-ins. For example, by focusing on the throughput, delay, and loss performance, our tool only requires a plug-in that captures the memory access latency for each CP memory. In other words, the plug-in only needs to capture the delay performance aspect of memory accesses, which can be modeled by the user, based on, for example, a queuing model or even an empirical chart, without having to emulate the processing details cycle-by-cycle. This design approach makes the simulation tool generic and adaptable to a wide range of CP architectures with the addition of a set of user provided plug-ins, capturing CP architecture specific features. With limited number of I/O and memory interfaces for CPs in general, the number of plug-ins needed are generally small, e.g., less than a dozen.

Capture important events only. The existing approach which attempts to accurately pin down the CP performance generally resorts to cycle-by-cycle or instruction-level simulation. This not only makes the simulation slow and storage space demanding, but also requires the availability of the executable program as input for the simulation. What is available to us, however, is only a piece of pseudo code for packet processing tasks mapped to each core, which defies the use of cycle-by-cycle simulation. Instead, we propose to identify a sequence of important events identifiable from the pseudo code and perform event-by-event, rather than instruction-by-instruction simulation. Since the number of important events identifiable in each code path is generally small, e.g., one dozen to a few dozen, one can expect that event-by-event simulation would be significantly faster than cycle-by-cycle simulation. Identifying important events is not difficult. For throughput, delay, and loss performance analysis, the important events may include memory and I/O accesses that results in a context switching, cache accesses that may cause a context switching, events that cause serialization effects, such as a critical section, and events that cause run-time code generation, such as packet fragmentation for which the code size is a function of the packet size. Although the instruction level activities, such as per instruction cycle time and instruction-level-pipelining (ILP) aborts, cannot be directly captured based on a piece of pseudo code, the average impact of these activities on the overall throughput, delay, and loss performance may be modeled. For example, by defining an event for code branching and associating with each branching event an average ILP abort cost in terms of wasted cycles, the impact of ILP aborts can be accounted for on average. In fact, using average data to simplify the simulation is not original. For example, the average per instruction cycle time has been widely used in processor performance analysis.

In summary, combining the event-by-event simulation and the separation of CP common features from CP specific features makes the tool very lightweight in terms of both time and space complexities. Moreover, simulation at the event level makes it possible to allow the use of pseudo code, rather than an exact program, as input for the simulation.

Allow for sustainable line rate estimation. No matter how lightweight a simulation tool would be, the simulation time is guaranteed to be prohibitively long if the goal is to perform exhaustive statistic analysis. This is because there are virtually unlimited numbers of possible packet arrival processes and mixtures of code paths the threads in each core may concurrently handle. Unless a user has in mind small numbers of targeted packet arrival processes and code path mixtures to be tested, performing exhaustive statistical analysis in an unconstrained parameter space is guaranteed to be extremely time consuming, if possible. In practice, for most CP programmers and designers, what they really want to know is, for a given task-to-CP-topology mapping, whether the CP can sustain wire-speed forwarding performance or not. Unfortunately, traditionally, the data inputs, including packet arrival processes and code path mixtures, are provided by the user of the tool, rather than part of the tool design. As a result, all the existing CP simulation tools were developed without being concerned with how the input data should be generated. This makes it difficult for a user or designer to effectively use such a tool to test whether the CP can keep up with the line rate or not. To address this issue, as part of the tool design, we develop a systematic approach to allow the sustainable line rate to be estimated for any given task-to-CP-topology mapping. In this approach, the user does not have to provide any packet arrival processes, nor code mixtures as input to the tool, but simply a piece of pseudo code for the tasks mapped to each core. The tool will automatically return the line rate the CP can sustain, under such a mapping.

3.2 Simulation Architecture

In this section, we introduce the generic CP organization, the code path definition, the simulation model, and an approach to estimate the sustainable line rate.

Generic CP Organization: Based on the aforementioned methodology, we consider the generic CP organization depicted in Fig. 1. This organization focuses on the characterization of multicore and multithreading features common to most of the CP architectures, leaving all other components being summarized in highly abstract forms. More specifically, in this organization, a CP is viewed generically as composed of a set of cores and a set of on-chip or off-chip supporting components, such as I/O interfaces, memory, level one and level two caches, special processing units, scratch pads, embedded CPUs, and coprocessors. These supporting components may appear at three different levels, i.e., the thread, core, and system (including core cluster) levels, collectively denoted as MEM_T, MEM_C, and MEM_S, respectively. Each core supports multiple threads which are scheduled based on a given thread scheduling discipline. Cores may be configured in parallel and/or multi-stage pipeline (a two-stage configuration is shown in Fig. 1). Packet processing tasks are partitioned and mapped to different cores at different pipeline stages or different cores at a given stage. A dispatcher distributes the incoming packets to different core pipelines based on any given policies. Backlogged packets are temporarily stored in an input buffer. A small buffer may also present between any two consecutive pipeline stages to hold backlogged packets temporarily. Packet loss may occur when any of these buffers overflow. In this paper, the tool is concerned with the CP throughput, latency, and loss performance only and the power and memory resource constraints are assumed to be met. This implies that we do not have to keep track of memory or program store resource availabilities or power budgets.

Code Path: An important concept for CP performance analysis is the code path. A code path is defined at the core level. For tasks mapped to a given core, a piece of pseudo code for these tasks can be written. Then a unique branch from the root to a given leaf in the pseudo code is defined as a code path associated with that core. An incoming packet to the core is accepted if there is a free thread in the core, and is associated with one code path, or a sequence of instructions that the core needs to execute throughout the life-time that the packet is in that core.

In this paper, a code path is broken down into a sequence of segments of instructions intermediated by events. For each segment, we are only concerned with the segment length, i.e., the number of instructions in the segment (which can be easily estimated on the basis of the pseudo code), or more precisely, the number of core cycles the core arithmetic logical unit (ALU) has to spend on the segment, assuming the average per instruction cycle time is known. Hence, a code path can be formally defined as follows:

T_k (M_k; $m_{1,k}$, $t_{1,k}$, $\tau_{1,k}$, ... , $m_{M_k,k}$, $t_{M_k,k}$, $\tau_{M_k,k}$): Code path k with event $m_{i,k}$ occurred at the $t_{i,k}$-th core clock cycle and with event duration $\tau_{i,k}$, where $k = 1, ..., K$ and $i = 1, 2, ..., M_k$, where K is the total number of code paths in the pseudo code mapped to the core and M_k is the total number of events in the code path.

$|T_k|$: the code path length or the total number of core clock cycles in the code path T_k (M_k; $m_{1,k}$, $t_{1,k}$, $\tau_{1,k}$, ..., $m_{M_k,k}$, $t_{M_k,k}$, $\tau_{M_k,k}$), where $k = 1, 2, ..., K$.

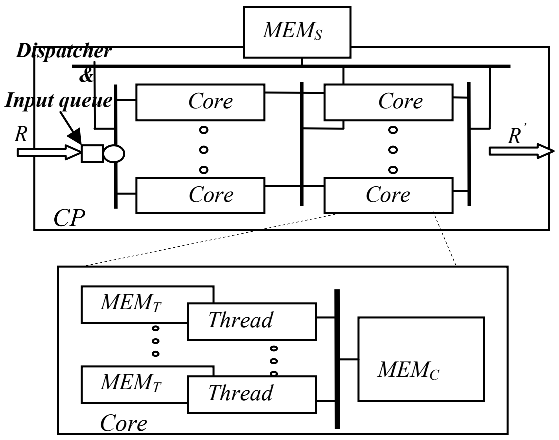
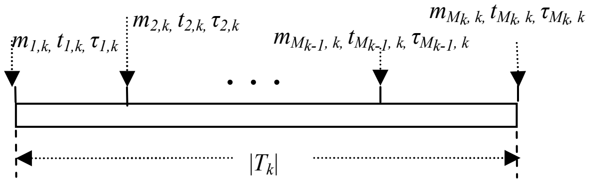


Figure 1. Generic CP Organization

A graphical representation of such a code path is given in Fig. 2.



m_{1,k}, t_{1,k}, τ_{1,k}
m_{2,k}, t_{2,k}, τ_{2,k}
m_{Mk-1,k}, t_{Mk-1,k}, τ_{Mk-1,k}
m_{Mk,k}, t_{Mk,k}, τ_{Mk,k}
|T_k|

Figure 2. T_k (M_k; $m_{1,k}$, $t_{1,k}$, $\tau_{1,k}$, , ... , $m_{M_k,k}$, $t_{M_k,k}$, $\tau_{M_k,k}$)

We note that a code path thus defined is simply a sequence of events with event inter-arrival times ($t_{i+1,k} - t_{i,k}$) for $i = 1, 2, ..., M_k-1$. We also note that the first instruction and the last instruction in the code path must be treated as events. For these events, $\tau_{i,k}= 0$. For an event $m_{i,k} \in MEM_T$, MEM_C, or MEM_S, $\tau_{i,k}$ represents the loaded resource access latency. To account for the serialization effect caused by, for example, a critical section, two events must be included, indicating the start and end of the critical section. Again, for these events, $\tau_{i,k}= 0$.

An important event is defined as one that is expected to have an impact on the throughput, delay, and loss performance. Currently, we have defined the following four types of important events: (1) events for the start and end of the code path; (2) resource access events which may cause significant delay and thread level interactions (context switching), events $m_{i,k} \in MEM_T$, MEM_C, or MEM_S ; (3) events that cause a serialization effect; and (4) events that cause dynamic code generation. More types of events can be incorporated if they are expected to contribute significantly to the throughput, delay, and loss performance. For example, a new type of event that identifies branching points in the code can be included for the purpose of estimation of ILP abort cost caused by branching, if the abort effect cannot be neglected.

Simulation Model: Our simulation tool focuses on three performance measures: throughput, delay, and loss. All three measures can be obtained at runtime as long as the latency L_k, a packet with code path k in each core can be simulated, which can be expressed conceptually as:

$$L_k = |T_k| + \sum_{j=1:M_k} (\tau_{j,k} + \tau^w_{j,k}), \quad (1)$$

where $\tau^w_{j,k}$ is the thread waiting time in the ready state after the event $m_{j\ k}$ finishes and $\tau^w_{j,k}= 0$ if event $m_{j,k}$ does not cause a context switching. For $m_{i,k} \in MEM_T$, MEM_C, or MEM_S, $\tau_{j,k}$ is dependent on the nature of $m_{j,k}$ access (number of memory reads or writes), the access speed (bus speed and memory speed), and access contention resolution mechanisms, such as the memory access pipelining and queuing architectures, which must be estimated based on a user provided plug-in. $\tau^w_{j,k}$ is dominated by multithreading effects, which is the core parameter to be simulated at runtime. $|T_k|$ is the total number of core clock cycles the core ALU spends on the packet.

Hence, for throughput, delay, and loss performance analysis, in general, all we need from the user is a set of plug-ins that estimate $\tau_{j,k}$ for $m_{i,k} \in MEM_T$, MEM_C, or MEM_S at runtime. Note that, although modeling CP-specific features in general is a nontrivial task, it should not be difficult to come up with empirical memory access latency models, e.g., in the form of charts or tables for a given CP. For example, by loading a given memory with different number and types of requests and measuring the corresponding loaded latencies using a cycle-accurate simulator or test board, one can build empirical charts or tables offline to be used to quickly estimate the memory access latency at runtime. There is no need to emulate the microscopic process for memory access at run-time, saving significant simulation time. As we shall see in the next section, with the unloaded memory access latencies provided by Intel, as well as a memory access waiting time estimated based on a simple FIFO queuing model, our simulation tool accurately characterizes IXP1200/2400 performance without further information about IXP1200/2400 specific features.

With the previously described preparation, now we describe our simulation model, which focuses on emulating non-CP specific components, including core topology, multithreading, code path, code path mixtures, and packet arrival processes, pertaining to all the CP architectures, with a limited number of plug-ins for resource access latency estimation. These plug-ins are pre-developed and plugged into the simulation model. Fig. 3 gives a logic diagram for the proposed simulation model, which is composed of four major components: (1) a simulation core based on the generic CP organization described in Fig. 1; (2) code path association with a packet in a core; (3) a packet arrival process; and (4) a set of plug-ins to the simulation core.

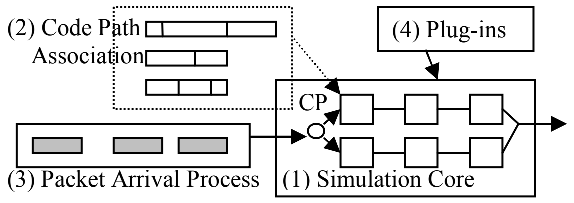


Figure 3. Proposed Simulation Model

Based on the generic CP architecture in Fig. 1, the simulation model focuses on emulating multithreaded cores which can be configured in any pipeline/parallel topology. Each core is modeled at a highly abstract level, running any number of threads based on a given thread scheduling discipline (our current design includes fine-grained, coarse-grained, and TDM-based disciplines). No further details of the core are modeled. A thread in a core that receives a packet will be assigned a code path. The way to assign code paths to threads in a core determines the mixture of code paths in that core. The packet arrival process can be generated from real traces (which also determine the code path mixture in each core), stochastic models, or deterministic models. Traditionally, the code path assignment and packet arrival process generation are not part of the simulation tool design, but as user provided inputs. Since all four components can be designed independent of one another, the design of components (1) and (4) combined constitutes a fast performance analysis tool in the traditional sense. In other words, as in traditional approaches, our tool allows any user provided packet arrival processes and/or mixtures of code paths to be simulated. Our goal, however, is to also design components (2) and (3) (to be discussed in Section 3.3) such that for any given task-to-CP-topology mapping, the tool can quickly return the maximum line rate the CP can sustain.

With the CP organization in Fig. 1 and the event annotated code path in Fig. 2, the proposed fast simulation tool is developed based on the event-driven simulation approach. To help understand why such a tool can be made to execute quickly, here we give an intuitive explanation by way of a simple example. Consider a code path in Fig. 4.

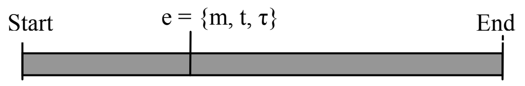


Figure 4. An Example of Event-annotated Code Path

In this code path, there are only three events, the start, end, and *e*. Event e takes place at the *t*-th cycle for resource, *m,* access with loaded latency *τ*. Now consider two threads in a core, each handling a code path as in Fig. 4. They share the ALU resource based on a fine-grained thread scheduling discipline (i.e., switch context at every instruction). Fig. 5 gives the instruction execution timeline for the two code paths. The dark gray parts represent the code path segments. The light gray parts represent the cycles spent on event *e*, i.e., the loaded *m* access latencies. The white part stands for the cycles spent in the ready state waiting for execution after event *e* finishes. In this case, each code path involves three event boundaries: the start of the code path, the end of the code path, and the start and end of event *e*. The arrows represent the switches of control from one thread to the other after executing one instruction. The idea is not to simulate each and every switch of control, but only the cycles at the event boundaries, i.e., the positions indicated by vertical lines. Since each code path may have up to a few dozens of events, only a few dozens of event boundaries need to be simulated per packet. As a result, the event-driven simulation tool that captures only those events can run several orders of magnitude faster than cycle-accurate simulation tools, as our testing results showed (a few seconds per simulation run on a Pentium III PC).

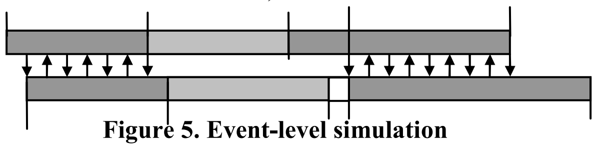


Figure 5. Event-level simulation

3.3 Sustainable Line Rate Estimation

The simulation model proposed in the previous section allows any packet arrival processes and code path mixtures to be simulated. In this section, we go one step further by designing the packet arrival processes and the code path mixtures to allow the maximum line rate a CP can sustain to be quickly simulated.

In the router industry, the performance of a router is judged mainly by whether its network interface cards can sustain wire-speed forwarding performance or not. A typical testing scenario is to use back-to-back, minimum-sized packets running at the line rate as input for the testing and all the packets are loaded with a typical code path, such as the code path that carries out basic tasks for IP forwarding. Following this industry practice, we would have already achieved our objective. However, while the packet arrival process thus generated makes sense, the use of a "typical" code path to determine whether a CP can sustain wire-speed forwarding performance may not always be a good idea, as long as untypical code paths may occur with non-negligible probabilities. For this reason, we adopt the industry practice on packet arrival process, but design our own code path assignment mechanism. In what follows, we discuss these two aspects separately.

Packet Arrival Process: Denote R as the line rate. Then the minimum packet time is *P/R*, where *P* is the size of a minimum-sized packet. We define a deterministic packet arrival process as minimum-sized packet arriving at fixed packet time interval T_P. For this process, the packet arrival rate $r = P/T_P$. The *r* value at which the packet loss is about to occur is then the maximum line rate the CP can sustain and if $r \le R$, we say that the line rate *R* can be sustained. Here, what code path mixture should be used as input to the simulation is yet to be specified, discussed below.

Code Path Assignment: Assume the user of the tool does not have a typical code path mixture in mind to test whether the line rate can be sustained. Our goal is then to identify the worst-case mixture of code paths that gives the lowest line rate the CP can sustain. This will ensure that the estimated sustainable line rate can always be achieved, under any mixtures of code paths. To this end, we first make the following two important observations.

First, intuitively, it is clear that the worst-case scenario for a given core will occur only when all the code paths in a code path mixture are the same. This is because having one code path for all the threads in the core will stress a particular resource the most. For example, if the longest code path may potentially create a bottleneck for the core ALU resource, then loading the core with this code path for all the threads will stress the core ALU resource the most. This observation significantly reduces the complexity for the identification of the worst-case mixture of code paths. In this case, one may then test different potential worst-case code paths separately, rather than different combinations of code paths, significantly reducing the number of test cases.

Second, with a deterministic arrival process and code path mixture, and at the saturated arrival rate, queuing cannot help improve the throughput performance simply because there is no traffic fluctuation to provide rooms for the buffers between pipeline stages to offload the queued packets once the queue levels build up. In this case, we may view a core pipeline as a buffer-free system and the throughput for a core pipeline is determined by the throughput at the bottleneck pipeline stage.

Based on these observations, we can logically decompose the problem into two sub-problems: (1) identifying the bottleneck pipeline stage; and (2) finding the worst-case code path that leads to the minimum throughput for the core at the bottleneck stage. In practice, these two sub-problems may not be separable. In what follows, we present a mechanism to address these problems.

In principle, the worst-case code path for one core must be associated with some given mixtures of code paths (not necessarily the worst-case mixtures of code paths) for other cores. The reason is that packet processing processes in different cores are coupled together through the sharing of supporting components of type MEM_S (see Fig. 1). As a result, to identify the worst-case code path for a particular core, one must also identify the associated code path mixtures for other cores. These code path mixtures are then associated with the arriving packets as input to various cores that simulates the entire system (see Fig. 1) as a whole. This simulation run will return the sustainable line rate for that core and this process must be repeated for all the cores to identify the bottleneck core. However, in practice, this coupling effect is not strong for the following reasons. First, different cores in different pipeline stages generally carry out distinct packet processing tasks. For example, in a three stage pipeline, the first stage may perform initial processing of incoming packets by loading the packets into an external DRAM. The second stage may perform major packet processing functions, which may involve significant table lookups in an external SDRAM. The last stage may mainly provide queue scheduling functions. As a result, the chances for different stages to interact with one another through MEM_S types of resource accesses are small. Second, due to the wide use of multiple external memory interfaces, memory banks, and core clusters sharing different L2 caches in CP design, such coupling effects are further reduced. Hence, in our current tool design, we simply overlook such coupling effects. Nevertheless, our tool can be easily extended to take such effects into account, e.g., by modeling the MEM_S accesses from other cores as a background random process derived from the *core*-access-intensive code path loaded to other cores.

Without considering the inter-core coupling effects, the problem is then reduced to one of finding the worst-case code paths for individual cores, separately. The worst-case code path k by definition generates the largest core latency L_k (see Eq. (1)). However, since without simulation, we have no idea about the values for the last term in L_k. The approach taken is to simply neglect $\tau^{w}_{j,k}$ and approximate L_k by $|T_k| + \sum_{j=1:\,M_k} \tau_{j,k}$, where $\tau_{j,k}$ is now the unloaded latency. This approximate latency can be estimated easily for each and every code path. We simply select an x % of the code paths with the largest L_k values to be considered as potential worst-case code paths to be tested. The reason to choose x % rather than just the one with the largest approximated L_k to be tested is simply to compensate for the inaccuracy of such estimations.

An initial testing of the above approach is encouraging. We tested the above approach against 50 randomly generated pseudo codes with each having 100 to 1000 branches or code paths with multiple MEM_T, MEM_C, and MEM_S types of memory accesses. Since the tool finishes running each case within 10 seconds on a Pentium IV PC, an exhaustive search of the worst-case code path for each pseudo code was performed. The worst-case code path thus found is then compared against the ones found by the above approach. For all 50 cases, we find that the true worst-case code path always falls into the top 1% (i.e., x = 1) of the entire code paths pool. This means that even for a pseudo code with up to 1000 code paths, only ten simulation runs with a fixed T_P are needed to identify the true worst-case code path, which takes a bit more than one minute. For a CP with a dozen of cores, this will take only dozens of minutes to pin down the worst-case code paths for all the cores.

Finally, for each core loaded with the worst-case code path, several simulation runs with different T_p values are performed to identify the maximum sustainable line rate for that core. This process is repeated for all the cores to identify the bottleneck pipeline stage for each core pipeline and finally the maximum sustainable line rate for the entire CP.

4. Simulation Testing

In this section, the accuracy of the proposed tool is tested against the Cycle-Accurate Simulators (CAS), i.e., IXP 1200/2400 SDK Developer workbenches [20]. With a set of code samples available in both IXP1200/2400, the sustainable line rates obtained from our tool are compared with those from CAS. For all the code samples, there are only a few number of code paths for each core and we can afford to perform exhaustive search for the bottleneck core and the corresponding worst-case code path. For this reason, the simulation focuses on testing the accuracy of the simulation tool only, assuming the bottleneck core and the corresponding worst-case code path are known. The code samples and corresponding simulation setups are described in Section 4.1 and the Section 4.2 present the test results.

4.1. Simulation Setups

Since all the cores in IXP1200/2400 run a coarse-grained thread scheduling discipline, our simulation tool is configured to run the coarse-grained thread scheduling algorithm as well. The functions in the core topology for IXP1200 and IXP2400 sample applications are briefly described as follows.

IXP1200 code samples: Four different code samples, *Packet Count* [15], *Generic IPv4 Forwarding*, *Layer-2 Filtering*, and *ATM/Ethernet IP Forwarding*, available in IXP1200 Developer workbench [20] are tested. The complete implementation details can be found in the Intel IXP1200 building blocks application design guide with the Developer workbench. In the following description of code samples, we focus on the functions mapped to the bottleneck core.

Packet Count: this code sample counts the number of packets received. A receive thread checks for data on the MAC port, transfers packet from MAC port to receive buffers. After packet reception is complete, the thread moves the packet into SDRAM and reads the packet header into the core. A counter is maintained in SCRATCH and is incremented on receiving a packet.

Generic IPv4 Forwarding: after packet reception as in *Packet Count*, RFC1812 generic IPv4 forwarding is implemented in this code sample.

ATM/Ethernet IP Forwarding: This code sample is a mixed code implementation of ATM /Ethernet IP forwarding. Only Ethernet-to-ATM flow is considered in the test. The header checksum check, TTL update, and IP lookup are performed in the receive block after packet reception as in *Packet Count*. Then the LLC/SNAP and modified IP headers are written back into the SDRAM. When the frame fragment with EOP (End of Packet) information is received, AAL5 trailer information is written into the SDRAM buffer and the complete PDU is enqueued for CRC generation at the next pipeline stage.

Layer-2 filtering: This code example implements Ethernet protocol, MAC address filtering and layer 2 forwarding in the receive block after packets are received.

Packet Count, *Generic IPv4 Forwarding*, and *Layer-2 Filtering* code samples are mapped to two core pipelined stages as shown in Fig. 6 and ATM/Ethernet Forwarding is mapped to three core pipeline stages as shown in Fig. 7. The original code samples are modified to allow only one core at the receive stage handling packets coming from a single port. As a result, the receive core becomes the bottleneck core to be tested. The code samples can also be changed to allow configuration of the number of threads from one to four.

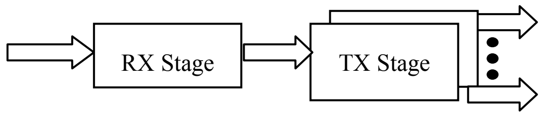


Figure 6. Pipeline Configuration for Packet Count, Generic IP Forwarding and Layer-2 Filtering

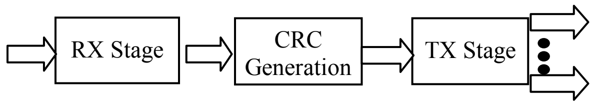


Figure 7. Pipeline Configuration for ATM/Ethernet Forwarding

Packet Count, *Generic IPv4 Forwarding*, and *Layer-2 Filtering* code samples are mapped to two core pipelined stages as shown in Fig. 6 and ATM/Ethernet Forwarding is mapped to three core pipeline stages as shown in Fig. 7. The original code samples are modified to allow only one core at the receive stage handling packets coming from a single port. As a result, the receive core becomes the bottleneck core to be tested. The code samples can also be changed to allow configuration of the number of threads from one to four.

IXP2400 code samples: *IPv4 Ethernet*, *DiffServ POS*, and *MPLS* in IXP2400 Developer workbench [20] are tested. In what follows, the code samples are briefly explained. All of these applications have five blocks and the same pipeline configuration as shown in Fig. 8. The complete implementation details can be found in the Intel IXP 2400 building blocks application design guide with the Developer workbench.

The first block is a packet receive block and uses a scratch ring to communicate with the next block. The second block is a functional block where application specific functional pipeline executes in parallel on four cores. The pipeline has five microblocks in *DiffServ POS* application: PPP decapsulation/classify microblock, 6-tuple classifier microblock, TCM meter microblock (ignored for our simulation setup: in case of no traffic profile in effect, packets may only pass through a classifier and a marker [RFC 2475]), DSCP Marker microblock and IPv4 forwarder microblock. In *MPLS* and *IPv4 Ethernet* applications, the pipeline consists of two microblocks: MPLS processing and IPv4 forwarder microblocks for *MPLS* and Ethernet decapsulation/classify/filter and IPv4 forwarder microblocks for *IPv4 Ethernet*. The third block is the queue manager which performs enqueue/dequeue operations on the hardware-assisted SRAM queues. The queue manager receives enqueue requests from the functional pipeline through a scratch ring. Another scratch ring is fed with dequeue requests from the

CSIX scheduler. The fourth block is the CSIX scheduler which selects constant-length packet segments to be transmitted to the CSIX fabric. The final block is the CSIX transmit block which receives transmit messages from the queue manager and moves packet segments into a transmit buffer.

For all the IXP2400 code samples we tested, the number of threads in use is not configurable in the original code samples, which is fixed at eight. To test the functional block, three of the four cores are disabled and the remaining core creates a bottleneck at this block.

Our simulation tool only needs to run a single core, corresponding to the bottleneck core for sample applications described above. The sustainable line rate for this bottleneck core is compared with that of CAS simulation involving the entire multi-stage pipeline. We assume that in the presence of resource access contentions, the resource access requests will be serviced based on a simple FIFO queuing mechanism. This means that unloaded resource access latencies and a set of simple resource access FIFO queues are the only IXP1200/2400 specific features or plug-ins used in our simulation tool. The rest are generic or common features pertaining to all the CP architectures. This indicates that our simulation tool is indeed generic and easily adaptable to a specific CP architecture.

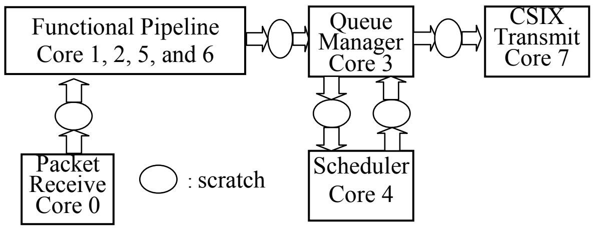


Figure 8. Ingress Blocks for IXP2400 Code Samples

The parameter settings for the simulation are as follows:
IXP1200/2400: ME clock rate = 200/600 MHz, Packet size = 64/64 bytes, SDRAM = 24/64MB, SRAM = 1/64 MB (for each channel of two).

4.2 Test Results

In this Section, we compare sustainable line rates obtained from our tool with those obtained from IXP1200/2400 CAS for the code samples described in Section 4.1. Tables 1-5 give the results for different cases in similar formats. For IXP 1200 case studies in Tables 1-4, the first column gives the number of threads configured; the second and the third columns list the core latencies and the sustainable line rates obtained from our tool and CAS, respectively. The last column lists the percentage difference between the two sets of results. The table format for IXP2400 case studies in Table 5 is similar to the ones for IXP 1200 case studies, except the first column, where now the applications are listed, rather than the number of threads, which for IXP2400 is fixed at eight. As one can see, for all the cases studied, the results obtained from our tool are within 6% of the CAS results. Moreover, each simulation run finishes in a few seconds on a Pentium IV PC. Since there are significant differences between IXP1200 and IXP2400 architectures, such consistent agreement of the two provides strong evidence, indicating that our tool can serve as an effective tool to aid the initial programming of a CP as well as new CP architecture design.

Table 1. The Tool versus CAS (IXP1200) for *Packet Counting*

Thr-eads	Tool		CAS		% Error rate |R1-R2|*100/R2
	Total Latency(TL) (cycles)	Receive rate R1 (Mbps)	TL	R2	
1	296	334	293	337	0.89
2	375	525	380	518	1.40
3	645	460	627	473	2.74
4	875	450	856	460	2.17

Table 2. The Tool versus CAS (IXP1200) for *Generic IPv4 Forwarding*

Thre-ads	Tool		CAS		% Error rate
	TL	R1	TL	R2	
1	560	183	537	183	0.00
2	590	347	600	328	5.79
3	687	447	687	431	3.71
4	936	438	876	449	2.45

Table 3. The Tool versus CAS (IXP1200) for *ATM/Ethernet IP Forwarding*

Thre-ads	Tool		CAS		% Error rate
	TL	R1	TL	R2	
1	732	140	724	140	0.00
2	830	247	812	250	1.20
3	985	312	981	311	0.32
4	1260	326	1184	343	4.96

Table 4. The Tool versus CAS (IXP1200) for *Layer-2 Filtering*

Thre-ads	Tool		CAS		% Error rate
	TL	R1	TL	R2	
1	735	140	730	140	0.00
2	816	251	798	257	2.33
3	960	320	978	314	1.91
4	1168	351	1304	354	0.85

Table 5. The Tool versus CAS (IXP2400) with 8 threads

Applications	Tool		CAS		% Error rate
	TL	R1	TL	R2	
Diffserv Pos	2960	830	2950	832	0.24
MPLS	2824	870	2750	890	2.25
IPv4 Ethernet	2898	849	2935	839	1.19

Finally, we note that in all the code samples, there is a critical section in the receive stage. In particular, the critical section dominates the code path of the *Packet Count* sample, constituting about 67% of the total code path. This dominant serialization effect causes the line rate to decrease when more than two threads are configured as shown in Table 1. This effect is successfully captured by our tool, which accounts for the impact of the critical section by recognizing the start and end of critical section events in the code path. Interested readers may refer to [17] for detailed explanation on why this critical section causes the decrease of line rate when adding more than two threads.

5. Conclusions and Future Work

This paper proposed a novel communication processor (CP) performance analysis methodology and a simulation tool based on this methodology. This tool is generic, fast, and accurate, which make it possible to allow fast performance testing of packet processing tasks to CP core topology mapping and CP design space exploitation.

Current design of the tool does not include thread scheduling disciplines which involve instruction level details, such as simultaneous multithreading. We plan to exploit the possibility of modeling the performance impact of such disciplines on the system-level performance. The goal is to allow the tool to be applicable to this kind of thread scheduling disciplines.

6. References

[1] Cavium Octeon CN38xxx (http://www.cavium.com/) and SUN UltraSPARC T2 (http://www.sun.com).

[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: an Infrastructure for Computer System Modeling, " *IEEE Computer*, Vol. 35, No. 2, 2002.

[3] M. Rosenblum, E. Bugnion, S. Devine, S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *Modeling and Computer Simulations*, Vol. 7, No. 1, pp. 78-103, 1997.

[4] E. Kohler, R. Morris, B. Chen, J. Janotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, Vol. 18, No. 3, pp. 263-297, Aug. 2000.

[5] Z. Huang, J. P. M. Voeten, and B. D. Theelen, "Modeling and Simulation of a Packet Switch System using POOSL," *Proceedings of the PROGRESS workshop 2002*, pp. 83-91, October 2002.

[6] Wen Xu and Larry Peterson, "Support for Software Performance Tuning on Network Processors", *IEEE Network,* July/August 2003.

[7] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao, "NePSim: A Network Processor Simulator with a Power Evaluation Framework," *IEEE Micro, Special Issue on Network Processors for Future High-End Systems and Applications*, Vol. 24, No. 5, pp. 34 – 44, Sept/Oct. 2004.

[8] P. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors," *IEEE Design & Test of Computers*, Vol. 19, No. 6, pp. 17-26, Dec 2002.

[9] "Advanced Software Development Tools for Intel IXP2xxx Network Processors," Intel White Paper, Oct. 2003.

[10] L. Thiele, S. Chakraborty, M. Gries, S. Kiinzli, "Design Space Exploration of Network Processor Architectures," *Network Processor Design: Issues and Practices*, Editors: P. Crowley, M. Franklin, H. Hadimioglu, P. Onufryk, Morgan Kaufmann Publishers, October 2002.

[11] J. Yao, Y. Luo, L. Bhuyan, and R. Iyer, "Optimal Network Processor Topologies for Efficient Packet Processing," in *Proc. of IEEE GLOBECOM,* Nov. 2005.

[12] R. Ramaswamy, N. Weng, and T. Wolf, "Analysis of Network Processing Workloads," in *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 226-235, Austin, TX, March 2005.

[13] N. Weng and T. Wolf, "Pipelining vs Multiprocessors – Choosing the Right Network Processor System Topology," in *Proc of Advanced Networking and Communications Hardware Workshop (ANCHOR 2004) in conjunction with ISCA 2004*, June 2004.

[14] L. Yang, T. Gohad, P. Ghosh, D. Sinha, A. Sen, and A. Richa, "Resource Mapping and Scheduling for Heterogeneous Network Processor Systems," in *Proc. of Symposium on Architecture for Networking and Communications Systems*, 2005.

[15] E. Johnson and A. Kunze, "IXP 1200 Programming", Intel Press.

[16] H. Che, C. Kumar, and B. Menasinahal, "A Fast Latency Bound Estimation Algorithm for a Multithreaded Network Processor," *the 18th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Nov. 2006.

[17] H. Che, C. Kumar, and B. Menasinahal, "Fundamental Network Processor Performance Bounds," *the 4th IEEE International Symposium on Network Computing and Applications (IEEE NCA05).*

[18] J. Emer, et. al., "Asim: A Performance Model Framework," Computer Magazine, pp. 67, Feb. 2002.

[19] M. Vachharajani, N. Vachharajani, D. Penry, J. Blome, and D. August, "The Liberty Simulation Environment," Version 1.0.

[20] Intel, IXP1200/2400 developer workbench, provided by IXA SDK 2.x/3.x