LATA: A Latency and Throughput-Aware Packet Processing System*

Jilong Kuang and Laxmi Bhuyan
Computer Science & Engineering Department
Unviersity of California, Riverside
900 University Ave, Riverside, CA 92521, USA
{jkuang, bhuyan}@cs.ucr.edu

ABSTRACT

Current packet processing systems only aim at producing high throughput without considering packet latency reduction. For many real-time embedded network applications, it is essential that the processing time not exceed a given threshold. In this paper, we propose LATA, a LAtency and Throughput-Aware packet processing system for multicore architectures. Based on parallel pipeline core topology, LATA can satisfy the latency constraint and produce high throughput by exploiting fine-grained task-level parallelism. We implement LATA on an Intel machine with two Quad-Core Xeon E5335 processors and compare it with four other systems (Parallel, Greedy, Random and Bipar) for six network applications. LATA exhibits an average of 36.5% reduction of latency and a maximum of 62.2% reduction of latency for URL over Random with comparable throughput performance.

Categories and Subject Descriptors

C.4 [**PERFORMANCE OF SYSTEMS**]: Design studies

General Terms

Design, Algorithms

Keywords

Parallel-pipelining, packet processing, multicore architecture

1. INTRODUCTION

The explosive growth of network bandwidth requires orders-of-magnitude increase in packet processing power. The advent of commodity multicore platforms has opened a new era of computing for network applications to take advantage of these low-cost machines. More and more network packet processing systems have been developed on such platforms ranging from general-purpose processors (e.g., Intel's Xeon [2]) to network processors (e.g., Intel's IXP platform [1]) and programmable logic devices (e.g., NetFPGA [5]). To exploit available parallelism for better throughput, these systems usually take one of the following three forms:

1. Spatial parallelism, where multiple concurrent packets are processed in different processors independently. Typical examples can be found in early work for TCP (Transmission Control Protocol) parallelism [9] and recent work in scalable DPI (Deep Packet Inspection) design [15].
2. Temporal parallelism (pipelining), where multiple processors are scheduled into a pipeline to overlap periodic executions from different threads. It has been widely adopted in network processors, including Shangri-La [10], auto-partitioning [11], statistical approach [18] and Greedy [25].
3. Hybrid parallelism, which integrates both spatiality and temporality to enjoy the advantages from both sides. It forms a parallel pipeline core topology, where each stage contains multiple parallel cores, such as Random [23] and Bipar [26].

Although the above approaches aim to maximize the throughput, none of them have considered latency reduction issue in packet processing because they employ coarse-grained packet-level parallelism. As emerging latency-sensitive applications become popular, such as online gaming, VoIP, fast IP-lookup and real-time DPI, latency plays a more important role than throughput [20]. Therefore, it is necessary to design a packet processing system that can attain high throughput under given latency constraints.

Traditional task scheduling algorithms, such as list-based scheduling [8] and clustering-based scheduling [24], can reduce program latency by exploiting fine-grained task-level parallelism. However, without pipelining, they suffer from significant throughput deterioration when executing periodical packet processing tasks. Papers [27] and [17] have presented some research results on reducing protocol latency for high-speed gateways and telecommunication systems based on hybrid parallelism. Developing a packet processing system that considers both latency and throughput for multicore architectures is both interesting and challenging.

In this paper, we propose LATA, a LAtency and Throughput-Aware packet processing system for multicore architectures. It adopts hybrid parallelism with parallel pipeline core topology in fine-grained task level to achieve low latency and high throughput. We accomplish the above goal through the following three steps. First, we design a list-based pipeline scheduling algorithm from the task graph. Second, we apply a deterministic search-based refinement process to reduce latency and improve throughput through local adjustment. Third, we devise a cache-aware resource mapping scheme to map the program onto a real machine.

To the best of our knowledge, LATA is the first to consider both latency and throughput in packet processing systems. We implement LATA on an Intel machine with two Quad-Core Xeon E5335 processors and conduct extensive experiments to show its better performance over other systems such as Parallel [9], Greedy [25], Random [23] and Bipar [26]. Based on six real packet processing applications chosen from NetBench [19] and PacketBench [21], LATA exhibits an average of 36.5% reduction of latency across all applications without substantially degrading the throughput. It shows a maximum of 62.2% reduction of latency for URL application over Random with comparable throughput performance.

The rest of this paper is organized as follows. Section 2 introduces LATA system design. Section 3 presents LATA scheduling, refinement and mapping algorithms. Section 4 describes the

*This research was partly supported by NSF grants CNS 0509440, CCF 0811834 and CNS 0832108.

DAC 2010, June 13-18, 2010, Anaheim, California, USA.

experiment framework. The performance evaluation is shown in Section 5 and Section 6 concludes this paper.

2. LATA SYSTEM DESIGN

Figure 1 shows LATA's system design flowchart. Given an application, we first generate its corresponding task graph with both computation and communication information. Then, we proceed in a three-step procedure to schedule and map the task graph according to our novel design. Last, we deploy the program onto a real multicore machine to obtain its performance result.

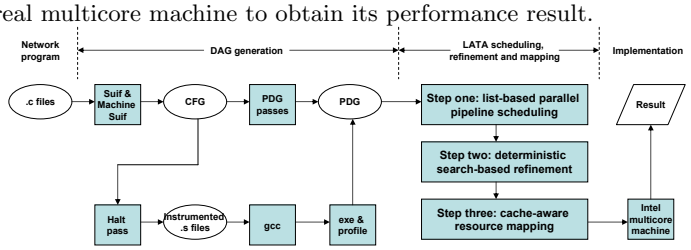

Figure 1: LATA system design flowchart.

2.1 Program Representation

We use program dependence graph (PDG) to represent a program as shown in Figure 2(a). PDG can also be called task graph, which is a weighted directed acyclic graph (DAG) defined by tuple G=(V, E, C, T), where V=$\{n_i$,i=1:$v\}$ is the set of nodes and v=$|V|$, E=$\{e_{i,j}$=<n_i, n_j>$\}$ is the set of communication edges and e=$|E|$. Each node represents a task and each edge represents a communication from one task to the other. C is the set of edge communication times and T is the set of node computation times. $c_{i,j}$ is the communication time on edge $e_{i,j}$ and t_i is the computation time on node n_i [14].

We assume the DAG has a single starting point denoted by head node n_{head} and a single ending point denoted by end node n_{end}. As an illustration in Figure 2(a), each node represents a basic block or a first-level loop in the program, which constitutes the tasks to be scheduled. Although the computation time for each node is easy to obtain (e.g., by inserting timers in the code), the communication time for multicore architectures is hard to measure due to the memory hierarchy. We address this issue next.

2.2 Communication Measurement

We can not accurately calculate the communication time between two cores in a multicore architecture like Figure 8 unless we know the exact location of the cores. In LATA design, we use the average communication cost based on data cache access time, as given in Equations 1 and 2. $Comm_{avg}$ means the average communication cost to transfer a unit data set, which can be approximated by system memory latencies (L1, L2 and main memory access time) and program data cache performances (L1 and L2 cache hit rate). $DataSize$ refers to the transferred data set size between two communicating tasks.

$$Comm = Comm_{avg} \times DataSize \tag{1}$$

$$Comm_{avg} = T_{L1} \times Hit_{L1} + T_{L2} \times (1 - Hit_{L1}) \times Hit_{L2} + T_{MEM} \times (1 - Hit_{L1}) \times (1 - Hit_{L2}) \tag{2}$$

2.3 Problem Statement

We define *latency* as the schedule length of a program and *throughput* as the system throughput. The problem statement is: *given the latency constraint L_0, schedule a program in parallel pipeline core topology so as to maximize the throughput Th.*

The aim is to rearrange the tasks shown in Figure 2(a) into the parallel pipeline task graph shown in Figure 2(b), so that the total execution time $T_1 + T_2 + T_3 + T_4$ is minimized while maintaining the throughput as high as possible. As we know, the throughput can be calculated by the inverse of the longest stage time $\frac{1}{T_{max}}$ in pipelining. Thus, we form our objective function in Equation 3, where L is the scheduled latency.

$$Maximize \;\; Th = \frac{1}{T_{max}} \;\; (\text{s.t.} \;\; L \leq L_0) \tag{3}$$

2.4 DAG Generation

As shown in Figure 1, LATA's system design consists of DAG generation, LATA scheduling, refinement and mapping, and finally implementation and evaluation. We briefly explain the DAG generation in this section and defer other parts to the following sections. To generate the DAG, we first convert the original C program into the SUIF control flow graph (CFG) [7]. For the ease of dependency analysis, we include all the functions of an application into one single file. After that, we write a Machine SUIF pass [4] to extract the PDG following Ferrante's algorithm based on both control and data flow information [13]. Finally, by using the Halt library in Machine SUIF to instrument source code, we profile the program in the task level for both the computation time and communication time.

To measure the computation time, we feed the program with continuous traffic traces to obtain the average execution time and frequency for each task. To measure the communication time, we first use LMbench [3] to get the L1, L2 and main memory access latencies. Then, we use SUIF/machine SUIF compilers to profile the variable liveness set at the entry of each basic block to measure the transferred data set size. Finally, we measure the program L1 and L2 data cache hit rate by PAPI [6]. After collecting all these data, we can get the communication time following Equation 2.

3. LATA SCHEDULING, REFINEMENT AND MAPPING

3.1 List-based Pipeline Scheduling Algorithm

LATA constructs the parallel pipeline topology based on traditional list scheduling algorithm, which is effective in both performance and complexity [16]. Given a DAG, we define node priority based on the computation top level assuming the same unit computation time for each node. According to [22], the computation top level of n_i is the length of the longest path ending in n_i, excluding t_i and all communication time. The purpose of assuming the same unit computation time is to find out all task-level parallelism, since nodes in the same level are independent and hence can be scheduled in parallel. The head node n_{head} belongs to level -1. The level of a certain node depends on the highest level among its predecessors. If its highest level predecessor belongs to level i, then that node belongs to level i+1.

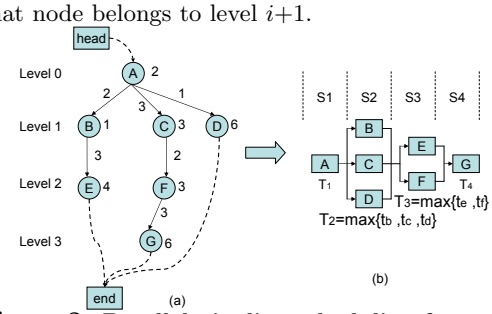

Figure 2: Parallel pipeline scheduling from a DAG.

We define *ready nodes* as those nodes whose predecessors have already been scheduled. Therefore, a *ready node* can be safely scheduled next. LATA starts off by putting the head node into the list, and then iteratively attaches *ready nodes* in the task graph to the last nodes in the list. This step guarantees that nodes in the list are sorted according to their priorities.

After the list is constructed, LATA schedules nodes with the same priority into the same pipeline stage in parallel. Each parallel node takes up one processor and we finally obtain a parallel pipeline topology. In this way, latency can be reduced by hiding the computation time of less expensive tasks. Figure 2(b) shows the parallel pipeline scheduling from Figure 2(a).

In the parallel pipeline topology, we denote a sequential section as a stage with only sequential tasks, such as S_1 and S_4. Similarly, a parallel section refers to a stage with parallel tasks, such as S_2

and S_3. We define *communication critical path* (CCP) as the communication time between two stages, where $CCP_i{=}max\{c_{i,j}\}(n_i{\in}V_i$ and $n_j{\in}V_{i+1})$. The complexity of this step comes from 1) priority assignment, which is $O(V{+}E)$ according to [22], 2) a precedence order among the tasks and 3) CCP calculation. Mergesort with a complexity of $O(VlogV)$ can be used to order the nodes. The complexity associated with calculating CCP is simply $O(S{\cdot}E)$. In conclusion, the complexity is $O(VlogV{+}S{\cdot}E)$.

3.2 Search-based Refinement Process

This step focuses on iteratively finding a better scheduling topology by local adjustment of tasks in two phases. The first phase aims at reducing latency and the second phase aims at improving throughput. Although optimizing task scheduling problem is NP-complete in general [12], our heuristic adopts greedy algorithm and works well in practice with low time complexity.

3.2.1 Latency Reduction

Latency can be reduced by reducing either computation time or communication time. Because computation dominates the overall execution time for most packet processing applications running on multicore architectures, we prioritize computation reduction in designing LATA. Hence, LATA first applies *latency hiding* to reduce computation time. Then, *CCP elimination* and *CCP reduction* are used to reduce communication time.

Computation reduction: We define *critical node* as the node in a pipeline stage which dominates the computation time. Then, *Latency hiding* can be defined as a technique that places a *critical node* from one stage to one of its adjacent stages without violating dependencies, so that its computation time is shadowed by the other *critical node* in the new stage. Backward hiding (BaH) refers to placing a *critical node* into its precedent stage. Forward hiding (FoH) refers to placing a *critical node* into its following stage.

Figure 3 illustrates two cases of *latency hiding*, where the node length reflects the computation time. For all the figures shown in this section, bold lines between two stages represent CCPs. Figure 3(a) is the same as Figure 2(b). In Figure 3(b), we place E into its precedent stage with B, where the computation time of E is shadowed by D. In Figure 3(c), E is placed into its following stage and E's computation time is shadowed by G.

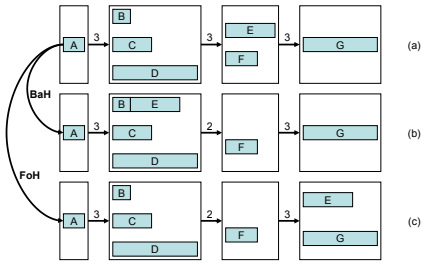

Figure 3: ***Latency hiding*** **on node** E**.**

For each *critical node*, we test whether we can place it into one of its two adjacent stages without violating dependencies or increasing the latency. To break the tie in some cases, we favor the stage with more latency reduction. If both stages happen to reduce the same amount of latency, we favor BaH over FoH. This heuristic increases chances of more potential latency reduction in future iterations. The complexity is $O(S{\cdot}E^2)$ for each iteration, because we have to update the CCP and latency for each attempt, which results in $O(E^2)$. The total complexity is $O(V{\cdot}S{\cdot}E^2)$ after $O(V)$ iterations in this step.

Communication reduction: There are two techniques in communication reduction, namely *CCP elimination* and *CCP reduction*. *CCP elimination* is to eliminate communication time by combining two adjacent stages into one. If every node has only one predecessor in a certain stage, we can attach nodes in that stage to their predecessors in the precedent stage.

As shown in Figure 4, the first elimination combines the last two stages together. G is attached after F, since F is the only predecessor of G in Figure 2(a). The second elimination shown in the figure combines the last two stages again. This time, we attach E to B and FG to C. From Figure 4(c) we see that two CCPs haven been eliminated from the original pipeline scheduling, which results in the latency reduction by 6.

CCP reduction is to reduce the CCP weight by switching a node associated with the current CCP to one of its adjacent stages. Figure 5 shows two reduction techniques BaS and FoS. Backward switch (BaS) refers to switching a node backward to its precedent stage. Forward switch (FoS) refers to switching a node forward to its following stage. For each CCP, we consider the two nodes associated with it. Both BaS and FoS are tested on the task in the two nodes. If any latency reduction can be obtained, we take that action. To break the tie in some cases, we favor the one with more CCP reduction. In case of equal CCP reduction, we choose BaS rather than FoS due to the same reason as in *latency hiding*.

In Figure 5(b), E is switched backward to B, so communication time between B and E is eliminated, and communication time between C and F becomes the new CCP with less weight. Figure 5(c) shows the case where B is switched forward to E. Similarly, we see a decreased CCP between the two stages.

As we decrease the latency, T_{max} will possibly increase, which is unfavorable for the throughput according to Equation 3. So, for each CCP, whether we apply *CCP reduction* or *CCP elimination* is decided by Q, a beneficial ratio defined by $Q{=}\frac{\Delta L}{\Delta Th}$. We start off by selecting the biggest CCP. Then both techniques are tested to get the ratio Q. We take action on the one whose resulting Q is larger, which guarantees minimal throughput sacrifice. This process is iteratively executed until the latency constraint is achieved. The complexity for both techniques is $O(E{\cdot}V)$ in each iteration. Therefore, the total complexity is $O(E^2{\cdot}V)$ after $O(E)$ iterations.

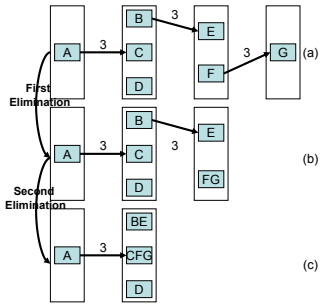

Figure 4: ***CCP elimination.***

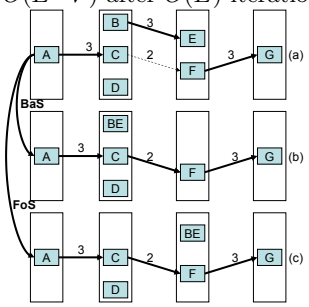

Figure 5: ***CCP reduction.***

3.2.2 Throughput Improvement

So far, we have reduced the latency by various techniques. In this section, we focus on improving throughput without violating the latency constraint. According to Equation 3, we can improve throughput by reducing T_{max} through *decomposition*. During the previous latency reduction process, chances are that many nodes are comprised of several tasks. If a node with T_{max} consists of more than one task, it can be decomposed into two separate nodes to reduce the bottleneck stage time T_{max}.

We define *decomposition* as to decompose one node with multiple tasks into two separate nodes without violating the dependencies. There are four decomposition techniques depending on how the two decomposed nodes are located as shown in Figure 6.

- SeD (Sequential Decomposition): Decompose two tasks from one processor into two adjacent tasks in different processors in a sequential section.
- PaD (Parallel Decomposition): Decompose two tasks from one processor into two parallel tasks in different processors in a parallel section.
- BaD (Backward Decomposition): Decompose two tasks from one processor into one task in the current section and the other in the precedent section.

- FoD (Forward Decomposition): Decompose two tasks from one processor into one task in the current section and the other in the following section.

We proceed the refinement process by iteratively applying the *decomposition* on current T_{max} node until no more throughput gain can be made. During each iteration, we first locate the node where T_{max} comes from. Then, for each task within that node, we attempt to apply the four decomposition techniques. After recording all possible decomposition points and corresponding techniques where positive results appear, we choose the task where the reduction of T_{max} is maximized as the potential decomposition point. If the latency constraint is not violated, we take that action.

The complexity is $O(V^2)$ for each iteration and there are $O(V)$ iterations. Hence, the overall complexity is $O(V^3)$.

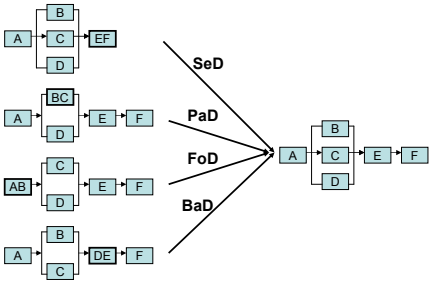

Figure 6: Illustration of four decomposition techniques. Thick boxes indicate bottleneck nodes.

3.3 Cache-Aware Resource Mapping

3.3.1 Pre-mapping

The first step, pre-mapping, assigns a pre-defined number of virtual processors (8 in LATA) to scheduled nodes, considering both computation time balancing and communication time minimization. First, we check all parallel sections to see if we can combine two independent parallel nodes into one without increasing the latency nor reducing the throughput. After this, if we still end up with more scheduled nodes than real nodes, we iteratively bipartition the pipeline into two parts with the cutting point being the minimal CCP. This guarantees a minimal communication overhead [26]. For each bipartition step, we assign virtual processors in proportion to the workload in each portion. With respect to workload, we refer to the total computation time of all the tasks in that stage. At the same time, we avoid assigning more than one virtual processor to a single task.

This recursive algorithm terminates when 1) there is only one virtual processor left unmapped or 2) there is only one stage left with extra virtual processors. In the first case, we assign all the remaining tasks into that virtual processor. In the second case, we assign all the remaining virtual processors into that stage, each virtual processor taking a fair share of workload by round-robin.

3.3.2 Real Mapping

The second step, real mapping, addresses specific task-to-core mapping. Figure 7(a) shows the tree structure of the processing units (PUs) on Xeon chip. From bottom up, a group of two cores shares the same last level cache (L2). Two of these groups (4 cores) share the same socket (S1 or S2). Two of these sockets (8 cores) share the same chip. Obviously, the communication cost between cores is asymmetric as illustrated by the different thickness of the curves in Figure 7(a). As a result, we can take advantage of the tree hierarchy to implement a cache-aware resource mapping.

First, we extract all the communication edges out of the scheduled topology and sort those edges in decreasing order. Figure 7(b) shows a sample scheduling topology after pre-mapping, where all the arrows represent the communication time. The thicker the arrow, the more time-consuming the communication. Second, we start off by picking the most time-consuming edge and then assign the two associated nodes to nodes with minimal communication cost. In our example, A and C are picked first and are assigned to C0 and C2, respectively. Third, we iteratively apply the same greedy algorithm until all the nodes are mapped to real cores as shown in Figure 7(c). During each iteration, we just pick the thickest edge out of all the remaining edges and assign the unmapped nodes with the cores that incur the least communication cost among all the unmapped cores.

When the real system provides more cores than the original scheduled topology, we simply apply real mapping first and then put extra cores to the bottleneck stage for packet-level parallelism.

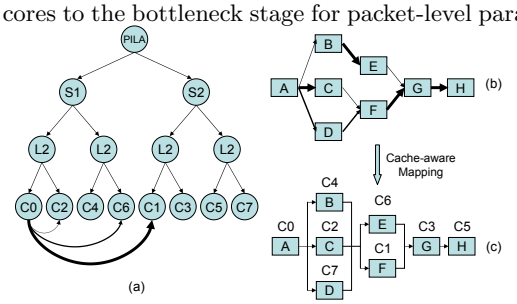

Figure 7: The hierarchy of processing units (PUs) on Xeon chip with communication asymmetry and illustration of cache-aware mapping.

The complexity in this section is dominated by pre-mapping process. Due to the bipartitioning, there are $O(logS)$ iterations. For each iteration, the complexity is simply $O(VlogV)$, which is from the sorting algorithm. Thus, the total complexity is $O(VlogV \cdot logS)$ in the algorithm. Considering the first two steps in LATA system design, we conclude that LATA has a total time complexity of $O(V^3 + V \cdot E^2 \cdot S)$. Since LATA is designed off-line, this complexity is acceptable for packet processing systems.

4. EXPERIMENT FRAMEWORK

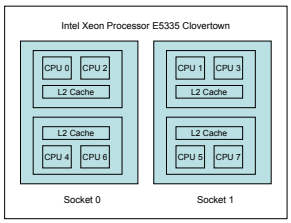

Figure 8: Layout of two Quad-Core Intel Xeon E5335 processors.

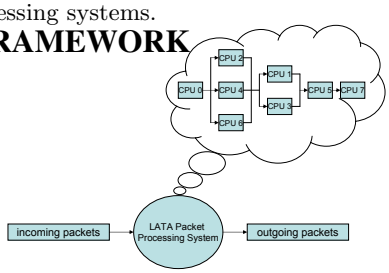

Figure 9: LATA packet processing system adopts parallel pipeline topology.

We implement and evaluate LATA along with four other systems (Parallel [9], Greedy [25], Random [23] and Bipar [26]) to show its performance advantage in both latency and throughput. Latency is measured by the average execution time of one packet in microseconds (usec) and throughput is measured by million packets per second (mpps).

We build our LATA packet processing system on a vendor multicore machine as shown in Figure 8. The target platform, an Intel Clovertown machine, consists of two sockets. Each socket has a Quad-Core Xeon E5335 processor with two cores sharing a 4MB L2 cache. The L1, L2 and main memory access latencies turn out to be 3, 14 and 217 CPU cycles, respectively, as measured by LMbench [3]. Figure 9 illustrates the overall system design. LATA assumes a single incoming and outgoing queue. The central part consists of an 8-core machine organized into a parallel pipeline core topology to exploit both spatial and temporal parallelism. The hardware configuration is set by default as an 8-core machine with instruction cache up to $4k$ instructions. The instruction cache size is an important parameter in partitioning programs and is used when we compare LATA with other Network Processor (NP) systems, whose processing engine has limited memory [1]. Our system is running Linux-2.6.18 OS and we use Pthread libraries to synchronize different tasks.

Six applications are chosen from NetBench [19] and PacketBench [21], including five IP-level programs (Flow, IPv4-trie, Route, DRR and IPchains) and one application-level program (URL). Their functionalities and code sizes (the number of instructions) are listed in Table 1. Our selection of applications is based on the following three metrics: 1) code size should be large; 2) applications should be representative; and 3) applications should be parallelized. The packet trace is from NetBench with 10,000 packets. The routing table used for IPv4-trie is MAE-WEST [21] and the routing table size for DRR, IPchains and Route is set to 128 by default. We scale URL's results by a factor of 0.01 due to figure space limitations. For LATA, we assume the latency constraint is 75% of the sequential execution time for each application.

Table 1: Packet processing applications.

Application	Functionality	Code Size
URL	URL-based switching	1428
Flow	Flow classification	3190
IPv4-trie	IPv4 routing based on trie	4596
Route	IPv4 routing based on radix	6600
DRR	Deficit-round robin scheduling	7633
IPchains	Firewall based on IP source	14735

We classify the four systems into two groups according to the form of parallelism. In the first group, LATA is compared with Parallel system (spatial parallelism), where every processor independently executes different packets in parallel as in [9]. No memory constraint is considered in this group. We also implement a list scheduling algorithm (List) called HLFET (Highest Levels First with Estimated Times) [8] as a reference for the best achievable latency. In the second group, we compare LATA with three NP systems based on pipelining (temporal parallelism) as in Greedy [25] and parallel pipelining (hybrid parallelism) as in Random [23] and Bipar [26]. These systems have limited memory constraints for each processor.

5. PERFORMANCE EVALUATION

5.1 Comparison with Parallel System

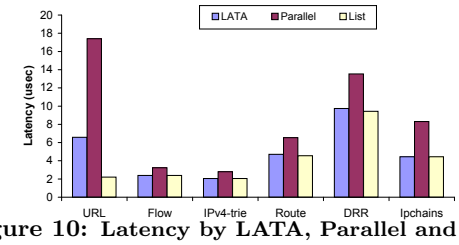

Figure 10: Latency by LATA, Parallel and List.

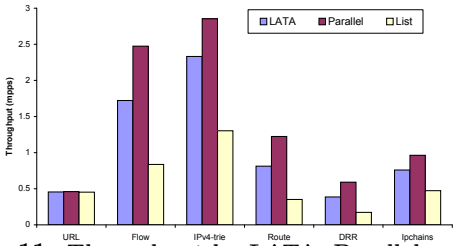

Figure 11: Throughput by LATA, Parallel and List.

Figures 10 and 11 show the latency and throughput for six applications by LATA, Parallel and List. We observe that Parallel suffers from high latency due to its sequential execution of tasks. Compared with Parallel, LATA reduces the latency by an average of 34.2%. Particulary, for URL, LATA achieves the maximal latency reduction of 62.2%. In addition, LATA's throughput is close to that of Parallel in spite of the 75% latency constraint. This is because LATA is capable of optimizing its parallel pipeline core topology to produce good throughput. With respect to List, which is designed to produce the lowest latency, LATA actually matches its latency in most cases by aggressively exploiting task-level parallelism. Furthermore, LATA outperforms List in throughput by an average of 41.0% and a maximum of 56.7% for Route.

5.2 Comparison with Three NP Systems

Figures 12 and 13 exhibit the latency and throughput for the three NP systems. Except LATA, all other systems adopt packet-level parallelism, which suffer from high latency. The slightly lower latency by Bipar and Greedy over Random comes from less communication overhead due to shorter pipeline length. LATA, on the other hand, exposes a substantial latency decrease by an average of 37.3% and maximum of 62.2% for URL compared with Random. Considering the throughput, we observe that LATA catches up with other systems in 5 out of 6 applications in spite of the 75% latency constraint, except the Flow application. However, the average of 30.4% throughput loss in Flow is compensated by 26.0% performance gain in latency reduction. This tradeoff once again proves LATA's uniqueness in satisfying the stringent latency constraint while attaining a comparable throughput.

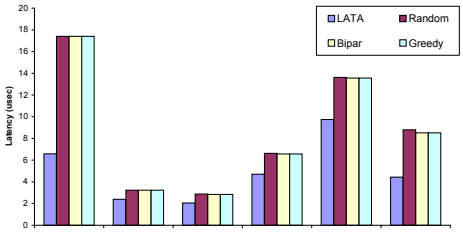

Figure 12: Latency by LATA, Greedy, Random and Bipar.

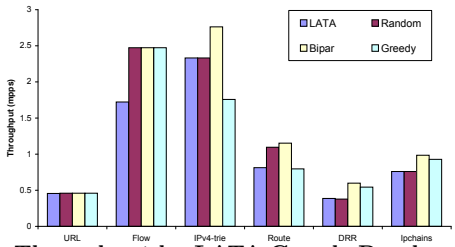

Figure 13: Throughput by LATA,Greedy,Random and Bipar.

5.3 Latency Constraint Effect

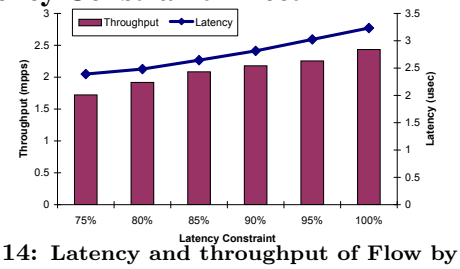

Figure 14: Latency and throughput of Flow by LATA.

In this section, we show how the latency constraint affects the throughput of LATA by alleviating the latency constraint from 75% to 100%. We choose the Flow application as an example because its throughput by LATA is the worst in Figure 13. From Figure 14, we observe that as the latency constraint becomes less stringent, the throughput improves accordingly. In fact, when there is no constraint (at the point of 100%), LATA produces the same throughput as other systems do. This is because LATA spares nodes from parallel sections to help reduce the bottleneck stage time by applying *decomposition*. Originally, those nodes are used to satisfy latency constraints, causing many tasks from sequential sections to be fed into few nodes, which deteriorates the throughput with large T_{max}. This figure also shows the latency performance at each point with an increasing trend, which follows the changing latency constraint. In a word, not only can LATA achieve low latency without substantial throughput loss when the latency constraint is stringent, but also it can attain high throughput when the latency constraint is light.

5.4 Scalability Performance of LATA

We evaluate LATA's scalability by varying the number of cores. Figure 15 demonstrates the latency and throughput for Route. As the number of cores increases, we observe a decreasing trend of latency, which reflects the fact that LATA exploits task-level parallelism to reduce program execution time. When the core resource is not plenty (less than 3), no task-level parallelism can be exploited. When the number of cores increases from 3 to 6, task-level parallelism gradually benefits and an obvious time decrease can be observed. As the core resource continues to increase (more than 6), the latency has reached the lower bound.

In addition, the increasing bars show that the throughput improves with more cores. We can make two interesting observations. First, there is a slight throughput decrease when the number of cores increases from 2 to 3. This seemingly contradictory result can be explained by the fact that LATA prioritizes latency reduction when the latency constraint has not been satisfied. In this case, the extra core is used to reduce latency rather than improve throughput. We can clearly see the latency reduction during that period from the latency curve. Second, while the latency becomes saturated after 6 cores, the throughput continues to improve. This is because that extra cores can reduce bottleneck stage workload, which results in better throughput for the whole system.

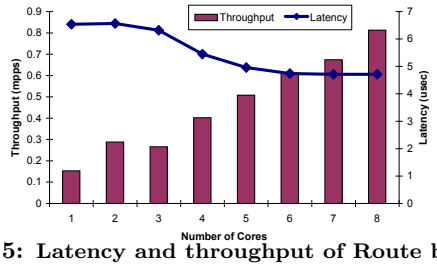

Figure 15: Latency and throughput of Route by LATA.

5.5 Instruction Cache Size Performance

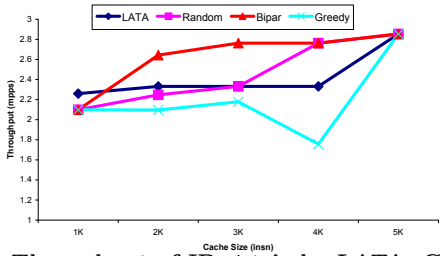

Figure 16: Throughput of IPv4-trie by LATA, Greedy, Random and Bipar.

Lastly, we analyze the effect of the instruction cache size for IPv4-trie. Figure 16 shows the throughput when the cache size varies. As the cache size increases, we observe an increasing trend of throughput. Bipar produces the best throughput in most cases due to its minimal communication cost and balanced workload assignment. Greedy, which performs the worst, suffers from imbalanced task assignment, especially when the cache size is $4k$. LATA and Random sit between Bipar and Greedy. However, LATA has the least cache requirement. When the cache size is as small as $1k$, LATA produces the best throughput. Its throughput slowly grows as the cache size increases from $2k$ to $4k$. After that, LATA's performance catches the best. Since the code size is less than $5k$ for IPv4-trie, all systems produce the same best throughput at $5k$ point. The corresponding latency performance is similar to that of Figure 12 and hence, we omit it due to space limitations.

6. CONCLUSION

In this paper, we design, implement and evaluate LATA, a latency and throughput-aware packet processing system. By adopting hybrid parallelism with parallel pipeline core topology in fine-grained task level, LATA is able to achieve low latency and high throughput. LATA consists of a list-based pipeline scheduling algorithm, a deterministic search-based refinement process and a cache-aware resource mapping scheme. Compared with four other systems for six real network applications, LATA exhibits an average of 36.5% reduction of latency across all applications and a maximum of 62.2% reduction of latency for URL over Random with comparable throughput performance.

7. REFERENCES

[1] Intel IXP2XXX Product Line of Network Processors. http://int.xscale-freak.com/XSDoc/IXP2xxx/IXP2xxx_index.htm.

[2] Intel Xeon Machine. http://www.Intel.com/Xeon.

[3] Lmbench. http://www.bitmover.com/lmbench/index.html.

[4] Machine-SUIF, Harvard University. http://eecs.harvard.edu/hube/software/nci/overview.html.

[5] NetFPGA. http://www.netfpga.org/.

[6] Papi. http://icl.cs.utk.edu/papi/.

[7] SUIF Compiler System. http://suif.stanford.edu/.

[8] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, (12):685–689, 1974.

[9] M. Bjorkman and P. Gunningberg. Locking effects in multiprocessor implementations of protocols. In *Proc. of SIGCOMM '93*, Sept. 1993.

[10] M. Chen, X. F. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming. In *Proc. of ACM PLDI '05*.

[11] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *PLDI '05*.

[12] H. El-Rewini, H. Ali, and T. Lewis. Task scheduling in multiprocessing systems. *IEEE Transactions on Computer*, (12), 1995.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, (3):319–349, 1987.

[14] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE TPDS*, 4(6), June 1993.

[15] D. Guo, G. Liao, L. Bhuyan, B. Liu, and J. J. Ding. A scalable multithreaded l7-filter design for multi-core servers. In *Proc. of ACM ANCS '08*, San Jose, CA, Nov. 2008.

[16] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4), Dec. 1999.

[17] S. Leue and P. A. Oechslin. On parallelizing and optimizing the implementation of communication protocols. *IEEE Transactions on Networking*, (1), Feb. 1996.

[18] A. Mallik, Y. Zhang, and G. Memik. Automated task distribution in multicore network processors using statistical analysis. In *Proc. of ANCS '07*, Orlando, FL, Dec. 2007.

[19] G. Memik, W. Mangione-Smith, and W. Hu. Netbench: A benchmarking suite for network processors. In *Proc. of ICCAD '01*.

[20] nVIDIA. Firstpacket technology improved system performance. *nVIDIA Technical Brief TB-02434-001_v01*, May 2006.

[21] R. Ramaswamy and T. Wolf. Packetbench: A tool for workload characterization of network processing. In *Proc. of WWWC-6 '03*.

[22] O. Sinnen. *Task Scheduling For Parallel Systems*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2007.

[23] N. Weng and T. Wolf. Pipelining vs multiprocessors-choosing the right network processor system topology. In *Proc. of ANCHOR'04*.

[24] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE TPDS*, 5(9), Sept. 1994.

[25] J. Yao, Y. Luo, L. Bhuyan, and R. Iyer. Optimal network processor topologies for efficient packet processing. In *Proc. of Globecom '05*.

[26] J. Yu, J. Yao, L. Bhuyan, and J. Yang. Program mapping onto network processors by recursive bipartitioning and refining. In *Proc. of IEEE/ACM Design Automation Conference (DAC '07)*.

[27] M. Zitterbart. A multiprocessor architecture for high speed network interconnections. In *Proc. of INFOCOM '89*, 1989.