Enhancing Speedup in Network Processing Applications by Exploiting Instruction Reuse with Flow Aggregation

G. Surendra, Subhasis Banerjee, S. K. Nandy
CAD Laboratory, Supercomputer Education and Research Center
Indian Institute of Science, Bangalore 560012, India
Email: {surendra@cadl, subhasis@cadl, nandy@serc}.iisc.ernet.in

Abstract

Instruction reuse is a microarchitectural technique that improves the execution time of a program by removing redundant computations at run-time. Although this is the job of an optimizing compiler, they do not succeed many a time due to limited knowledge of run-time data. In this paper we examine instruction reuse of integer ALU and load instructions in network processing applications. Specifically, this paper attempts to answer the following questions: (1) How much of instruction reuse is inherent in network processing applications?, (2) Can reuse be improved by reducing interference in the reuse buffer?, (3) What characteristics of network applications can be exploited to improve reuse?, and (4) What is the effect of reuse on resource contention and memory accesses? We propose an aggregation scheme that combines the high-level concept of network traffic i.e. "flows" with a low level microarchitectural feature of programs i.e. repetition of instructions and data along with an architecture that exploits temporal locality in incoming packet data to improve reuse. We find that for the benchmarks considered, 1% to 50% of instructions are reused while the speedup achieved varies between 1% and 24%. As a side effect, instruction reuse reduces memory traffic and can therefore be considered as a scheme for low power.

1. Introduction

Network Processor Units (NPU) are specialized programmable engines that are optimized for performing communication and packet processing functions and are capable of supporting multiple standards and Quality of service (QoS) requirements. Increasing network speeds along with the increasing desire to perform more computation within the network have placed an enormous burden on the processing requirements of NPUs. This necessitates the development of new schemes to speedup packet processing tasks while keeping up with the ever increasing line rates. The above aim has to be achieved while keeping power requirements within reasonable limits. In this paper we investigate dynamic instruction reuse as a means of improving the performance of a NPU. The motivation of this paper is to determine if instruction reuse is a viable option to be considered during the design of NPUs and to evaluate the performance improvement that can be achieved due to instruction reuse.

Dynamic instruction reuse [1][2] improves the execution time of an application by reducing the number of instructions that have to be executed dynamically. Research has shown that many instructions are executed repeatedly with the same inputs and hence producing the same output [3]. Dynamic instruction reuse is a scheme in which instructions are buffered in a *Reuse Buffer* (RB) and future dynamic instances of the same instruction use the results from the RB if they have the same input operands. The RB is used to store the operand values and result of instructions that are executed by the processor. This scheme is denoted by S_v ('v' for value) and was first proposed in [1]. The RB consists of *tag*, *input operands*, *result*, *address* and *memvalid* fields [1]. When an instruction is decoded, its operand values are compared with those stored in the RB. The PC of the instruction is used to index into the RB. If a match occurs in the *tag* and *operand* fields, the instruction under consideration is said to be reused and the result from the RB is utilized. We assume that the reuse test can be performed in parallel with instruction decode [1]. The reuse test will usually not lie in the critical path since the accesses to the RB can be pipelined. The tag match can be initiated during the instruction fetch stage since the PC value of an instruction is known by then. Execution of load instructions involves an address computation and then accessing the memory location specified by the address. The address computation part of a load instruction can be reused if the instruction operands match an entry in the RB, while the actual memory value (outcome of load) can be reused if the addressed memory location was not written by a store instruction. The *memvalid* field indicates whether the value loaded from memory is valid while

1530-1591/03 $17.00 © 2003 IEEE

the *address* field indicates the memory address. When a store instruction is executed by the processor, the *address* field of each RB entry is searched for a matching *address*, and the *memvalid* bit is reset for matching entries. In this paper we assume that the RB is updated by instructions that have completed execution and are ready to update the register file. This ensures that precise state is maintained with the RB containing only the results of committed instructions. Instruction reuse improves performance since "reused instructions" can bypass some stages in the pipeline with the result that it allows the dataflow limit to be exceeded. Performance is further improved since subsequent instructions that are dependent on the reused instruction are resolved earlier and can be issued earlier.

Whereas dynamic instruction reuse has been exploited in the context of general purpose processing applications [1], to the best of our knowledge, this is the first paper that enhances the utility of instruction reuse with a flow aggregation scheme in network processing applications. Specifically, the following are the main contributions of this paper - (i) We evaluate dynamic instruction reuse for NPU benchmarks and show that significant instruction reuse can be exploited. (ii) We propose a flow aggregation scheme that exploits the locality in packet data to improve reuse. The idea is to store reuse information of "related" packets (flows) in separate *Reuse Buffers* (RB's) so that interference in RB's is reduced. (iii) We evaluate the impact of instruction reuse on resource contention and memory accesses.

The rest of the paper is organized as follows. We describe the flow aggregation approach in section 2, present simulation results in section 3 and conclude with section 4.

2. Improving reuse by aggregating Flows

A flow (commonly used in congestion control terminology) may be thought of as a sequence of packets sent between a source/destination pair following the same route in the network. A router inspects the IP addresses in the packet header and treats packets with the same source/destination address pairs as belonging to a particular flow. A router may also use application port (layer 4) information in addition to the IP addresses to classify packets into flows. A flow can be either implicitly defined or explicitly established. In the former, each router watches for packets between particular source/destination pairs by inspecting the headers. In the latter, the source sends a flow setup message across the network, declaring that a flow of packets is about to start (connection oriented systems). When packets with a particular source/destination IP address pair traverse through an intermediate router in the network, one can expect many more packets belonging to the same flow (*i.e.* having the same address pair) to pass through the same router (usually through the same input and output ports) in the near future. All packets belonging to the same flow will be similar in most of their header (both layer 3 and layer 4) fields and many a time in their payload too. For example, the source/destination IP addresses, ports, version and protocol fields in an IP packet header will be the same for all packets of a particular connection/session. Header processing applications such as firewalls, route lookup, network address translators, intrusion detection systems *etc*, are critically dependent on the above fields and yield higher reuse if flow aggregation is exploited. Instruction reuse can be improved if applications that process these packets somehow identify the flow to which the packet belongs to, and uses different RB's for different flows. The idea is to have multiple RB's, each catering to a flow or a set of flows so that similarity in data values (at least header data) is preserved ensuring that evictions in the RB is reduced.

A simple example (figure 1) is used to illustrate how flow aggregation reduces the possibility of evictions in a RB and enhances reuse. For simplicity, let us assume that an ALU operation (say addition) is computed by the NPU on incoming packet data a_i, b_i. Figure 1(a) shows the "reuse" scheme without flow aggregation using a single 4 entry RB while 1(b) exploits flow aggregation using two RB's. Assume that incoming packets are classified into two flows - Pkt_1, Pkt_2 and Pkt_6 belong to $flow_A$ while Pkt_3, Pkt_4, and Pkt_5 belong to $flow_B$. The RB (fig 1(a)) is updated by instructions that operate on the first four packets. When the contents of Pkt_5 is processed (there is no hit in the RB), the LRU entry (i.e. a_1, b_1) is evicted and overwritten with a_5, b_5. This causes processing of the next packet Pkt_6 to also miss (since the contents were just evicted) in the RB. Assume that a flow aggregation scheme is used with multiple RB's so that program instructions operating on packets belonging to $flow_A$ query RB_1 and those operating on $flow_B$ query RB_2 for exploiting reuse. Instructions operating on Pkt_5 will be mapped to RB_2 (which will be a miss) while instructions operating on Pkt_6 mapped to RB_1 will cause a hit and enable the result to be reused leading to an overall improvement in reuse (compared to fig1(a)). Obviously, a good mapping strategy is necessary to uncover higher reuse.

One can relax the previous definition and classify packets related in some other sense as belonging to a flow. For instance, the input port through which packets arrive at a router and the output port through which packets are forwarded could be used as possible alternatives. For every packet that arrives, the NPU must determine the RB to be associated with instructions that operate on that packet. Routers that classify packets based on the IP addresses are required to parse the IP header and maintain state information for every flow. Flow classification based on the output port involves some computation to determine the output port. The output port is determined using the Longest

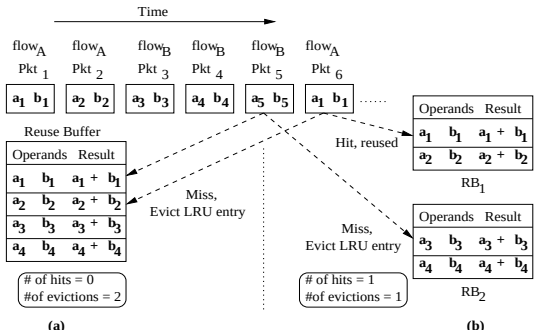

Figure 1. An example comparing (a) ordinary instruction reuse with (b) flow based reuse.

Prefix Match (LPM) algorithm and is computed for every packet irrespective of whether reuse is exploited or not [6]. Most routers employ *route caches* to minimize computing the LPM for every packet thereby ensuring that the output port is known very early [6]. A one-to-one (instruction) mapping between an output port and a RB is not practical since the number of output ports is usually greater than the number of RB's that can be incorporated on-chip. A many-to-one mapping scheme in which instructions operating on packets destined to different output ports query the same RB is necessary which by itself opens up a design space that needs to be carefully explored. Classification of flows based on the input port involves little or no computation (since the input port through which a packet arrives is known) but uncovers a smaller percentage of reuse for some applications.

2.1. Architecture proposal

For single processor systems the architecture proposed in [1] with a few minor changes is sufficient to exploit flow based reuse. However, for multiprocessor and multithreaded systems which are generally used in designing NPUs, extra modifications are required. The architecture proposed in this section can be applied to both non-blocking and blocking multithreaded execution models. The essence of the problem at hand is to determine the appropriate RB to be used by instructions operating on a packet and switch between RB's when necessary. The NPU is essentially a chip multiprocessor consisting of multiple RISC processing engines called microengines (using the terminology of Intel IXP1200 [7]) which support hardware multithreading with zero cycle context switching. Programs can run on multiple microengines and multiple threads. It is the job of the programmer to partition tasks across threads as well as programs across microengines. The inter-thread and inter-processor communication also has to be explicitly managed by the user. Each microengine has a RB array consisting of $N + 1$ RB's - RB_0, ..., RB_N (see figure 2). RB_0 is a default RB that is queried by instructions before the *flow id* of a packet is computed. The NPU also consists of a Flow id table that indicates the *flow id* for a packet and a mapper (which can be programmed) that identifies the RB to be used by instructions operating on that packet. The Flow id table and the mapper are accessible by all microengines of the NPU and also by the memory controller which is responsible for filling an entry on the arrival of a new packet. The *flow id* field is initialized to a default value (say 0) which maps to the default RB (RB_0). This field is updated once the actual *flow id* is computed based on any of the schemes mentioned previously. The *packet id* of the packet currently being processed by a thread is obtained by tracking (hardware required for this is maintained by the memory controller) the memory being accessed by load instructions (we assume that a packet is stored in contiguous memory). Each thread stores the *packet id* of the packet currently being processed, the *thread id* and the *flow id* to which the packet belongs. The selection of the RB based on the *flow id* is made possible by augmenting the Reorder Buffer (RoB) with the *thread id* and the RB id (the mapper gives the *flow id* to RB id mapping). Instructions belonging to different threads (even those in other microengines) access the Flow id table which indicates the RB to be queried by that instruction. The *flow id* field indicates the default RB (RB_0) initially. After a certain amount of processing, the thread that determines the output port updates the *flow id* entry in the Flow id table for the packet being processed. This information is known to all other threads operating on the same packet through the centralized Flow id table which is accessed by all threads. Once the *flow id* is known, the mapper gives the exact RB id (RB to be used) which is stored in thread registers as well as in the RoB.

When the processing of the packet is complete, it is removed from the memory i.e. it is forwarded to the next router or sent to the host processor for local consumption. This action is again initiated by some thread and actually carried out by the memory controller. At this instant the *flow id* field in the Flow id table is reset to the default value. In summary, reuse is always exploited with instructions querying either the default RB or a *flow id* specified RB.

2.2. Impact of reuse on resources

Instructions contend for various resources such as functional units and memory ports as they flow through the pipeline. An effect of instruction reuse is that demand for resources is reduced since reused instructions do not consume functional units allowing other ready instructions to execute early. This changes the schedule of instruction execution resulting in clustering or spreading of request for resources, thereby, increasing or decreasing *resource con-*

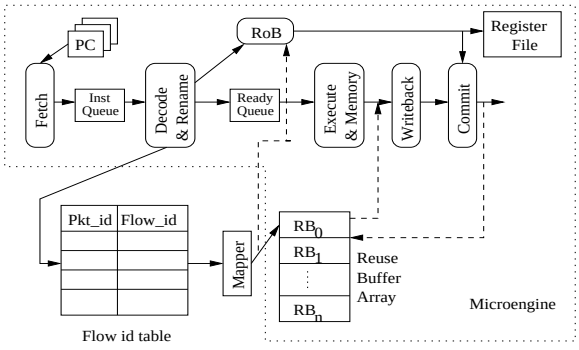

Figure 2. Microarchitecture to exploit instruction reuse using the Flow aggregation scheme.

tention [2]. Resource contention is defined as the ratio of the number of times resources are not available for executing ready instructions to the total number of requests made for resources. Exploiting reuse for load instructions helps in reducing the number of memory (on-chip and off-chip) accesses [4], bus transitions and port contention.

2.3. Operand based indexing

Indexing the RB with the PC enables one to exploit reuse due to dynamic instances of the same static instruction. Redundancy across dynamic instances of statically distinct instructions (having the same opcode) can be captured by indexing the RB with the instruction opcode and operands. One way to implement operand indexing is to have an opcode field in addition to other fields mentioned previously in the RB and search in parallel the entire RB for matches. In other words, an instruction that finds a match in the opcode and operand fields can read the result value from the RB. This is in contrast to PC based indexing where the associative search is limited to a portion of the RB. We use a method similar to the one proposed in [5] to evaluate operand based indexing. Operand indexing helps in uncovering slightly more reuse than PC based indexing (for the same RB size). This can be attributed to the fact that in many network processing applications, certain tasks are often repeated for every packet. Since there is significant correlation in packet data, the inputs over which processing is done is quite limited and hence network (especially header) processing applications tend to reuse results that were obtained while processing a previous packet.

3. Simulation Methodology and Results

The goal of this paper is to demonstrate the idea and effectiveness of aggregating flows to improve reuse and not the architecture for enabling the same. Hence, we do not simulate the architecture proposed in the previous section (this will be done as future work) but use a single threaded single processor model to evaluate reuse. We modified the SimpleScalar [8] simulator (MIPS ISA) and used the default configuration [8] to evaluate instruction reuse on a subset of programs representative of different classes of applications from two popular NPU benchmarks - CommBench [9] and NetBench [10] (see table 1). It must be noted that we use SimpleScalar since it is representative of an out-of-order issue pipelined processor with dynamic scheduling and speculative execution support. In other words, we assume that the NPU is based on a superscalar RISC architecture which is representative of many NPUs available in the market. Further, using Simplescalar makes it easy to compare results with certain other results in [9] and [10] that also use the same simulation environment.

Every application has a certain function (or piece of code) which reads a new packet. When the PC of an instruction matches with the PC of this function, the output port for the packet is read from a precomputed table of output ports. This gives the RB to be used for the current set of instructions being processed. The RB id is stored in the RoB along with the operands for the instruction and the appropriate RB is queried to determine if the instruction can be reused. The results therefore obtained are valid independent of the architecture proposed in the previous section. Since threads and multiple processors are not considered during simulation, the results reported in this paper give an upper bound on the speedup that can be achieved by exploiting flow aggregation. However, the bound can be improved further if realistic network traffic traces that are not anonymized in their header and payload are available.

Inputs provided with the benchmarks were used in all the experiments except FRAG for which randomly generated packets with fragment sizes of 64, 128, 256, 512, 1024, 1280 and 1518 bytes were used [11]. We evaluate instruction reuse for ALU and Load instructions for various sizes of the RB. We denote the RB configuration by the tuple (x, y) where x represents the size of the RB (number of entries) and y the associativity (number of distinct operand *signatures* per entry). x takes on values of 32, 128 and 1024 while y takes on values of 4 and 8. Statistics collection begins after 0.1 million instructions and the LRU policy is used for evictions since it was found to be the best scheme.

3.1. Reuse and Speedup - Base case

Table 1 shows reuse recovered by different RB configurations and speedup for the benchmark programs considered. We find that that network processing applications on average have slightly lesser instruction repetition patterns compared to SPEC benchmark results [1]. It is seen that a high number of hits in the RB need not necessarily lead

Table 1. Reuse and Speedup without flow aggregation for different RB configurations. R=% instructions reused, S=% improvement in speedup obtained due to reuse. % speedup due to operand indexing and % reduction in memory traffic due to load instruction reuse for a (32,8) RB is shown in the last two columns.

Bench-mark	(32,4) R/S	(32,8) R/S	(128,4) R/S	(128,8) R/S	(1024,4) R/S	(1024,8) R/S	ope-rand	Mem traffic
FRAG	7.9/3.7	10.4/5	20.4/4.9	25.8/5.7	24.4/8.3	29.8/10.7	5.3	42.1
DRR	12.6/0.16	12.8/0.18	15.5/0.5	16.5/0.64	18.2/0.86	20.3/1.1	0.4	11.6
RTR	15.2/3.8	16.3/4	33.2/6.1	35.9/6.3	47.6/8.1	49.4/8.6	9.2	71.3
REEDENC	19.8/2	21.7/2.17	20.3/2.05	21.9/2.23	25.2/2.95	27.4/3.13	4.7	8.7
REEDDEC	6.6/1.76	6.8/1.84	11.8/4	12.9/4.2	16.6/5.6	18.7/6.5	6	4.9
CRC	19.1/19.6	19.3/19.67	20.7/19.84	21.1/19.91	21.8/19.82	22.4/20.1	20.4	35.1
MD5	1.4/1.3	1.5/1.36	3.5/2.3	3.6/2.39	14.2/8.3	14.9/8.47	1.6	34.35
URL	18.8/9.4	19.2/10.2	19.9/11.2	20.32/11.44	22.2/12.7	22.7/12.92	13.1	42

to a higher speedup. This is because speedup is governed not only by the amount of reuse uncovered but also by the availability of free resources. Though resource demand is reduced due to reuse, resource contention becomes a limiting factor in obtaining higher speedup. An appropriate selection of both the number entries in the RB and the associativity of the RB are critical in obtaining good performance improvements. DRR is rather uninteresting yielding a very small speedup even with an (1024,8) RB. A closer examination of the DRR program reveals that though it is loop intensive, it depends on the packet length which varies widely in our simulations. A more significant reason for the small speedup gain is due to the high resource contention (figure 4). On increasing the number of integer ALU units to 6, multiply units to 2 and memory ports to 4 (we shall refer to this configuration as the *extended configuration*), a speedup of around 5.1% was achieved for a (32,8) RB.

3.2. Speedup due to flow aggregation

Simulation is carried out using 8 network interfaces (ports) with randomly generated addresses and network masks (mask length varies between 16 and 32). We use separate RB's for ALU and load instructions to reduce the size of the RB. This is because only load instructions utilize the *memvalid* and *address* fields in the RB which need not be present for ALU instructions. We found that the flow aggregation scheme with 4 RB's usually gives the best result for the benchmarks considered. Our simulation results indicate that a single RB is capable of capturing as much as 70% load instruction reuse. This number does not increase significantly when multiple load RB's are used. Hence, all results reported in this paper make use of 4 ALU RB's and 1 load RB to exploit flow based reuse. The flow aggregation is based on the output port (we use a route cache to determine this value) with a simple mapping scheme (for RTR we use the input port scheme since this is the program that computes the output port). We map instructions executing packets destined for port 0 and 1 to RB_1, 2 and 3 to RB_2 and so on. This type of mapping is clearly not optimal and better results can be obtained if other characteristics of the network traffic are exploited. Since most traffic traces are anonymized, this kind of analysis is difficult to carry out and we do not explore this design space. Figure 3 shows the speedup results due to flow aggregation for FRAG and RTR programs. Flow aggregation is capable of uncovering significant amount of reuse even when smaller RB's are used. For example, for the FRAG program, 5 (128,8) RB's (4 for ALU and 1 for load instructions) produces the same speedup as a single RB with a (1024,8) configuration. The solid lines represent speedup due to ALU instructions in the base case while the dotted lines show results due to the flow based scheme for ALU instructions only. The dashed lines indicate the additional contribution made by load instructions to the overall speedup. To examine the effect of reducing resource contention on speedup, we tried the *extended configuration* and obtained a 4.8% improvement in speedup

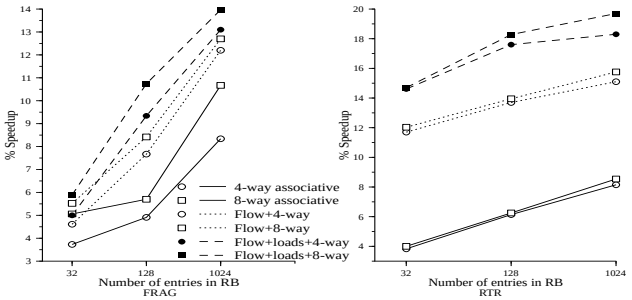

Figure 3. Speedup with flow aggregation

for RTR (2.3% for FRAG) over the flow based scheme (with (32,8) RB). Speedup is improved by 2% to 8% over the base scheme for other programs (this in conjunction with table 1 gives approximate values). The reuse and speedup results of these are not shown explicitly for lack of space.

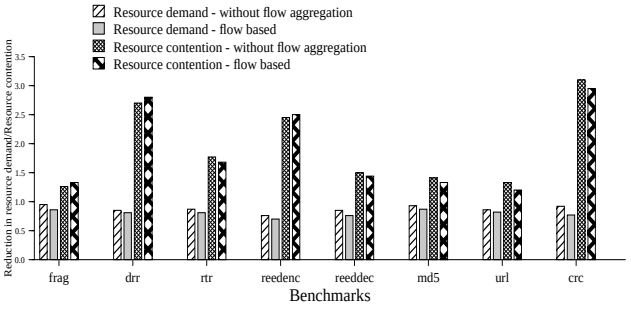

Figure 4. Resource demand and resource contention due to reuse

Figure 4 shows that resource demand due to flow reuse is lower than the base scheme. Resource demand is calculated by normalizing the number of resource accesses with reuse to that without reuse. The figure also shows resource contention normalized to the base case. As mentioned in section 2.2, contention may increase or decrease due to reuse. Resource contention comes down drastically on using the *extended configuration* (see section 3.1). Load instruction reuse also reduces memory traffic [4] significantly (see last column of table 1) thereby decreasing power consumption. Power reduction is also achieved due to reduced number of executions when an instruction is reused. This may however be offset due to accesses to the RB.

Also, as shown in table 1 (column 8 vs column 3), operand indexing is capable of uncovering additional reuse and speedup. An extra speedup of about 2% to 4% is achieved when flow aggregation is used along with operand indexing. Since a reused instruction executes and possibly commits early, it occupies a RoB slot for a smaller amount of time. This reduces the stalls that would occur as a result of the RoB being full. Flow based reuse has the ability to further improve reuse thereby reducing the RoB occupancy even further. Flow based reuse reduces the occupancy of the RoB by 1.5% to 3% over the base reuse scheme.

4. Conclusions

In this paper we examined instruction reuse in network processing applications. To further enhance the utility of reuse by reducing interference, a flow aggregation scheme that exploits packet correlation and uses multiple RB's is proposed. The results indicate that exploiting instruction reuse with flow aggregation does significantly improve the performance of our NPU model (24% improvement in speedup). Future work would be to simulate the architecture proposed, evaluate power issues and determine which instructions really need to be present in the RB. Interference in the RB can be reduced if critical instructions are identified and placed in the RB. Further, a detailed exploration of various mapping schemes is necessary to evenly distribute related data among RB's. Finally, we believe that additional reuse can be recovered from payload processing applications if realistic (non anonymized) traffic is used.

References

[1] A. Sodani and G. Sohi, "Dynamic Instruction Reuse," *In Proc. of ISCA-24*, July 1997, pp. 194-205.

[2] A. Sodani, G. Sohi, "Understanding the Differences between Value Prediction and Instruction Reuse," 31^{th} *Annual ACM/IEEE International Symposium on Microarchitecture*, Dec 1998, pp. 205-215.

[3] A. Sodani and G. Sohi, "An Empirical Analysis of Instruction Repetition," *In Proc. of ASPLOS-8*, 1998.

[4] J. Yang and R. Gupta, "Load redundancy removal through instruction reuse," *In Proc. Intn'l Conf. on Parallel Processing*, Aug 2000, pp. 61-68.

[5] C. Molina, A. Gonzalez and J. Tubella, "Dynamic Removal of Redundant Computations," *In Proc. Intn'l. Conf. on Supercomputing*, June 1999.

[6] F. Baker, "Requirements for IP Version 4 Routers," *RFC - 1812*, Network Working Group, June 1995.

[7] Intel IXP1200 Network Processor - Hardware Reference Manual, Intel Corporation, December 2001.

[8] D. Burger, T. M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," Technical Report CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[9] Tilman Wolf, Mark Franklin, "CommBench - A Telecommunications Benchmark for Network Processors," *IEEE Symposium on Performance Analysis of Systems and Software*, Apr 2000, pp. 154-162.

[10] Gokhan Memik, B. Mangione-Smith, W. Hu, "NetBench: A benchmarking Suite for Network Processors," *In Proc. ICCAD*, Nov. 2001.

[11] S. Bradner, J. McQuaid, "A Benchmarking Methodology for Network Interconnect Devices", *Request For Comments - 2544*, IETF, March 1999.