International Journal of Parallel Programming, Vol. 31, No. 6, December 2003 (© 2003)

On the Effectiveness of Flow Aggregation in Improving Instruction Reuse in Network Processing Applications

G. Surendra,[1] S. Banerjee,[1] and S. K. Nandy[1]

Instruction Reuse is a microarchitectural technique that exploits dynamic instruction repetition to remove redundant computations at run-time. In this paper we examine instruction reuse of integer ALU and load instructions in network processing applications and attempt to answer the following questions: (1) How much of instruction repetition can be reused in packet processing applications?, (2) Can the temporal locality of network traffic be exploited to reduce interference in the *Reuse Buffer* and improve reuse? and (3) What is the effect of reuse on microarchitectural features such as resource contention and memory accesses? We use an execution driven simulation methodology to evaluate instruction reuse and find that for the benchmarks considered, 1 to 50% of the dynamic instructions are reused yielding performance improvement between 1 and 20%. To further improve reuse, a flow aggregation scheme as well as an architecture for exploiting the same is proposed. This scheme is mostly applicable to header processing applications and exploits temporal locality in packet data to uncover higher reuse. As a side effect, instruction reuse reduces memory traffic and improves performance.

KEY WORDS: Network processors; instruction reuse; value prediction; multiprocessors.

[1] CAD Laboratory, Supercomputer Education and Research Center, Indian Institute of Science, Bangalore, India 560012. E-mail: {surendra@cadl;subhasis@cadl;nandy@serc}.iisc.ernet.in

0885-7458/03/1200-0469/0 © 2003 Plenum Publishing Corporation

1. INTRODUCTION

Application Specific Integrated Circuits (ASIC) have been traditionally associated with handling packet-processing tasks (data plane jobs) in routers to keep up with line rates. The drawbacks of ASIC based designs such as long design cycles and lack of flexibility have prompted the network community to opt for special purpose *Network Processing Units* (NPU) to handle packet processing tasks. NPUs are programmable and are capable of supporting multiple standards and Quality of service (QoS) requirements. However, the increasing network speeds have placed an enormous burden on the processing requirements of NPUs that are expected to carry out a variety of tasks ranging from intelligent packet forwarding/routing to providing QoS in today's multimedia rich network traffic. This necessitates the development of new schemes to speedup packet processing tasks while keeping up with the ever-increasing line rates.[1] In this paper we investigate *dynamic instruction reuse* (IR) as a means of improving the performance of an 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 reuse.

It has been shown that many static instructions produce only a small number of values and hence multiple executions of these instructions (i.e., dynamic instances) leads to the repeated generation of the same set of values.[2–4] This repetitive pattern in the values generated by instructions is exploited by techniques such as value prediction[2–4] (which is a speculative technique) and instruction reuse[5] (a non-speculative technique). Dynamic Instruction Reuse (IR)[5–10] (or computation reuse) improves the execution time of an application by reducing the number of instructions that have to be executed dynamically. Although removal of redundant computations is the job of an optimizing compiler,[9, 10] they do not succeed many a time due to limited knowledge of run-time data. A dynamic instruction can be reused if it has the same operands and produces the same output as a previous instruction (usually a previous instance of the same instruction). During program execution, instructions are buffered (cached) 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. Performance gains are achieved since reused instructions can bypass some pipeline stages with the result that it allows the dataflow limit to be exceeded. In case of dynamically scheduled processors, 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, to the best of our knowledge, this is the first paper that evaluates IR in network-processing applications as well as enhances it with a flow aggregation scheme. Specifically, the following are the main contributions of this paper—(i) we evaluate dynamic instruction reuse for a few packet-processing applications and show that instruction repetition is quite prevalent especially in header processing applications. (ii) We propose an aggregation scheme that combines the high-level concept of network traffic, i.e., "flows" with the low level microarchitectural feature of programs, i.e., repetition of instructions and data and propose an architecture that exploits temporal locality in incoming packet data to improve IR. The idea is to store reuse information of *related* packets (flows) in separate RB's (or separate sections of a single RB) so that interference in the RB's is reduced. We find that the benefits that can be achieved by exploiting flow aggregation significantly depends on the incoming traffic pattern as well as the application being executed. (iii) We evaluate the impact of IR on resource contention and memory accesses when flow aggregation is exploited.

The rest of the paper is organized as follows. In Section 2, we provide a brief overview of IR. We discuss the concept of network flows and how flow aggregation can be used to improve IR in Section 3. In Section 4, we describe the simulation methodology used to report results and conclude in Section 5.

2. DYNAMIC INSTRUCTION REUSE—AN OVERVIEW

In this section we briefly describe the methodology used by Sodani and Sohi[5] to exploit IR. We employ the same technique to analyze base IR in this paper. Research has shown that over 80% of the dynamic instructions executed in SPEC programs are repeated and this repetition is caused by a few static instructions.[7] It has been argued[7] that the repetitive nature of values being generated by instructions is due to (i) external inputs being repetitive, (ii) concise program representation (e.g., loop structures in which the index variables execute with same values) and (iii) non-scalar variables being represented using data structures (e.g., instructions accessing the data structure are often executed before the actual data variable is accessed). In other words, instruction/data repetition is an artifact of the way in which a program is represented and is less dependent on input data. In case of packet-processing applications, we find that IR (especially the flow based reuse proposed in Section 3) is significantly affected by the nature of traffic pattern (input data) too.

IR is exploited using a RB which is similar to a small cache. The RB is used to store the operand values and result of instructions that are executed by the processor. In this scheme denoted by S_v (**v** for value), the RB consists of a *tag, input operands, result, address,* and *memvalid* fields.[5] When an instruction is decoded (or register read is complete), its operand values are compared with those stored in the RB. The Program Counter (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. It is assumed that the reuse test can be performed in parallel with instruction decode/register read.[5] In some architectures, the register read operation is deferred to a later stage in the pipeline (just before execution) in which case, the reuse test has to be initiated at this point in time. One must remember that the reuse test (access to the RB to determine if an instruction can be reused) can be carried out only if all operands of the instruction concerned are available. The reuse test according to the authors of Ref. 5 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. It has been found that load and store operations frequently fetch/write to the same address locations, often with the same values.[3,4] 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 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 the *address* field indicates the memory address. When the processor executes a store instruction, the *address* field of each RB entry is searched for a matching *address*, and the *memvalid* bit is reset for matching entries.

Since the result value of a reused instruction is available from the RB, the execution phase is avoided reducing the demand for resources. In case of load instructions, reuse (outcome of load being reused) reduces port contention, number of memory accesses[11] and bus transitions. In most current day processors that have hardware support for clock gating, this could lead to significant savings in energy in certain parts of the processor logic.[12] This could be significant in statically scheduled processors where instructions dependent on reused instructions cannot be issued earlier. In dynamically scheduled processors, IR allows dependent instructions to execute early which changes the schedule of instruction execution resulting in clustering or spreading of request for resources. This leads to either an increase or decrease in *resource contention*;[6] where 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.

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 (i.e., squash reuse[5]—IR due to control independent instructions executed along a wrong path when the RB is updated with speculative instructions—is not exploited). We consider only integer ALU and load instructions since floating point instructions seldom occur in network applications. We also use separate RB's for load and ALU instructions since it reduces the overall storage. For the rest of the paper we refer to the instruction reuse scheme mentioned above (i.e., proposed in Ref. 5) as the *base reuse scheme*.

3. IMPROVING REUSE BY AGGREGATING FLOWS

A flow 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 Internet Protocol (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. 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 interfaces) 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 some portions of their payload too (e.g., when packets are encapsulated, many fields which belonged to a header will now be part of the payload of the new packet). For example, the source/destination IP addresses, ports, version, header length and protocol fields in an IP packet header will be the same for all packets of a particular connection/session. It is also reasonable to assume that packets of the same connection have the same packet length. Packet header processing applications such as firewalls, route lookup, network address translators, intrusion detection systems etc, are critically dependent on the above fields and have the potential to uncover higher reuse if flow aggregation is exploited. IR can be improved if the processor that processes these packets somehow identifies the flow to which the packet belongs to, and uses different RB's for different flows. The idea is to have multiple RB's (or a single RB which is

suitably partitioned), 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.

Flow aggregation may be viewed as a data placement scheme that places related or similar data at similar locations (in the same RB or same sections of an RB). This ensures that queries to a RB will result in a high probability of hits leading to higher IR. It must be noted that while instruction (and data) repetition is prevalent, it has to be captured or uncovered using an appropriate RB to obtain performance benefits. The aggregation strategy in network applications works at the granularity of packets, i.e., given a packet, the processor must determine which RB (or section of a RB) to query to exploit IR for instructions that will operate on that packet. In case of multiprocessor systems, an application is partitioned into tasks that are mapped to multiple processors which implies that the total number of dynamic instructions executed is divided between the PEs. Since now there are fewer instructions to be executed on each PE, fewer data values will be generated/used yielding better hit rate if a RB is used.

A simple example (see Fig. 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

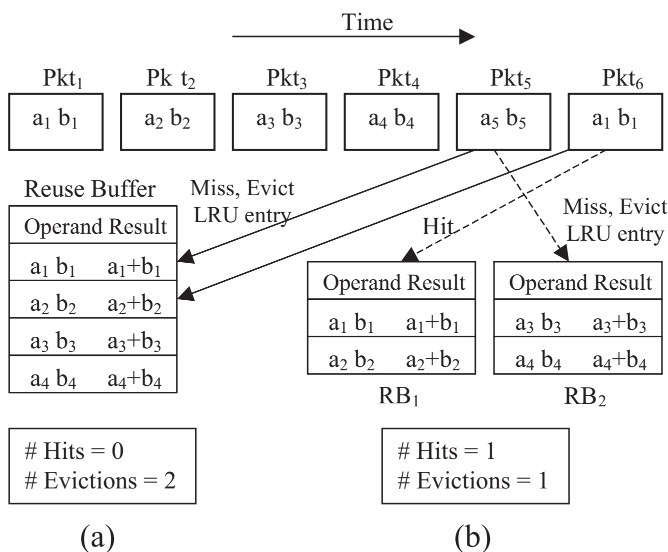

Fig. 1. An example comparing (a) normal instruction reuse with (b) flow based reuse.

by the NPU on incoming packet data $\mathbf{a_i}$, $\mathbf{b_i}$. Figure 1(a) shows the IR scheme without flow aggregation using a single 4 entry RB while Fig. 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., $\mathbf{a_1}$, $\mathbf{b_1}$ is evicted and overwritten with $\mathbf{a_5}$, $\mathbf{b_5}$. This causes processing of the next packet Pkt_6 to also miss (since the contents were just evicted) in the RB. Now assume that a flow aggregation scheme is used with multiple (or partitioned) 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 IR. This allows more reuse to be uncovered since instructions operating on Pkt_5 will be mapped to RB_2 (which will again 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 IR. The amount of IR that can be uncovered depends on the nature of traffic and data values, and it is quite possible that IR could decrease if smaller RB's are used or if ill-behaved traffic patterns occur (more on this in Section 4.2).

3.1. Flow Identification

One can relax the previous definition of a flow (Section 3) and classify packets related in some other sense as belonging to a flow. For instance, the input interface through which packets arrive at a router and the output interface through which packets are forwarded could be used as possible alternatives. This is a natural way of flow classification since in most circumstances packets of the same flow travel along the same path from the source to the destination. For every packet that arrives, the NPU must determine the RB to be associated with instructions that operate on that packet. Flow classification based on the output port involves some computation to determine the egress interface. This egress interface is determined using the Longest Prefix Match (LPM) algorithm and is computed for every packet irrespective of whether IR is exploited or not.[13] Most routers employ *route caches* to minimize computing the LPM for every packet thereby ensuring that the output interface is known very early. Classification of flows based on the input port involves little or no computation (since the input interface through which a packet arrives is known) but uncovers a smaller percentage of reuse for some applications. In case of routers that have a large number of ports, a many-to-one mapping between interfaces and RB's to be queried by instructions is necessary to ensure that the solution is practical. Although a very accurate classification scheme is

usually not required, consistency must be maintained in mapping packet flows to RB's for appreciable improvement in results.

3.2. Architecture Proposal

For single processor systems the architecture proposed in Ref. 5 with a few minor changes is sufficient to exploit flow based IR. However, for multiprocessor and/or multithreaded systems which are generally used in designing NPUs,[14, 15] extra modifications are required. The description given in this section is a proposal of an architecture that enables flow aggregation to be exploited and does not mimic the architecture used by us to report results in the paper. A more realistic evaluation using a multiprocessor simulation environment and an in-depth study of various architectural alternatives is currently being carried out as part of future work.

The NPU is essentially a chip multiprocessor consisting of multiple Processing Elements (PE's) each of which could support multiple threads of execution. Figure 2 shows the microarchitecture of a single PE. Each PE

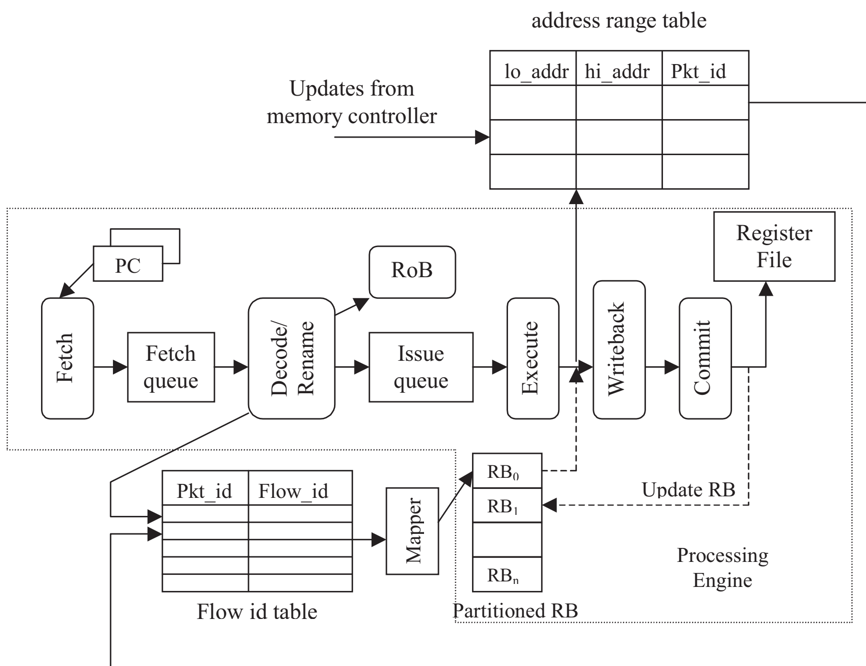

Fig. 2. A possible microarchitecture to exploit IR using the flow aggregation scheme.

has a RB array consisting of N+1 RB's (or a single RB that is partitioned into N+1 portions)—RB_0,..., RB_N. RB_0 is the default RB that is queried by instructions before the *flow id* of a packet is computed (for output port-based scheme only). The NPU also consists of a *Flow id table* that indicates the *flow id* for a packet and a mapper that identifies the RB to be used for that packet. The Flow id table and the mapper are accessible by all PE's 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. One scheme for implementing flow based IR is to tag packets that arrive at the router or NPU. This tag serves as a packet identifier, which after mapping, becomes the RB identifier and is used by instructions operating on the packet to query the appropriate RB. Another scheme would be one in which the *packet id* of the packet currently being processed by a thread is obtained by tracking the memory being accessed by read/load instructions (we assume that a packet is stored in contiguous memory). This scheme is described in more detail in this paper.

The essence of the problem at hand is to determine the appropriate RB to be queried by instructions operating on a packet and switch between RB's when necessary. This implies that the hardware must know which packet a thread is processing at any instant in time. The packet currently being processed by a thread is not known to the hardware since accesses to a packet are seen as load/store instructions to some memory address. This memory address has to be associated with a *packet id* for the Flow id table to be accessed. This is achieved by introducing an *address range table* that is maintained by the memory controller. The *address range table* contains a *packet id*, *lo_addr* and *hi_addr* fields. The *lo_addr* entry corresponds to the starting address in memory where the packet is stored. The *hi_addr*, which denotes the ending address of the packet depends on the packet length and is known to the memory controller (from the link layer information).

When a new packet arrives, it is transferred from the MAC controller to the memory by the memory controller. The number of bytes to be transferred is indicated by the link layer to the memory controller.[13] The memory controller updates a new entry in the *address range table* for the packet just received and fills up the *lo_addr* and *hi_addr* fields. Instructions (except loads) that operate on the new packet to determine its *flow id* access the default RB (as mentioned before, instructions belonging to a thread for which the *flow id* is not known access the default RB). When the PE executes load instructions, the *address range table* is accessed (in parallel to the default RB which is queried for exploiting reuse) to determine if the effective address of the load instruction matches any one of the entries. A match

in the *address range table* is said to occur when the effective address computed lies between *lo_addr* and *hi_addr*. This implies that the load instruction accessed some data from the packet whose *id* is given by the *packet id* field in the table. This *packet id* field is stored in a register within the thread context that executed the load instruction. All subsequent instructions belonging to the same thread continue to use the same *packet id*. Each thread stores the *packet id* of the packet currently being processed, the *thread id* and the *flow id* to which the packet belongs in local thread registers. The selection of the RB based on the *flow id* is made possible by augmenting the instruction queue and 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 access the *Flow identifier table* that 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 (for output-port based scheme) updates the *flow id* entry in the *Flow identifier table* for the packet being processed. This information is known to all other threads operating on the same packet through the centralized *Flow identifier table*. 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 instruction queue.

When the processing of the packet is complete, it is removed from memory, i.e., it is forwarded to the next router or sent to the host processor for local consumption. This action is initiated by some thread which resets the flow *id* field in the *Flow identifier table*. In summary, IR is always exploited with instructions querying either the default RB or a RB specified by the *flow id*. The main drawback of the above scheme is the complexity (number of read/write ports, table size which depends on the number of packets that can be stored at any instant in the NPU) involved in designing the centralized *Flow identifier table* and *address range table* that could be the potential bottleneck in the system. Also, the address range table is also not always indicative of the correct packet being processed since it makes certain assumptions about the underlying architecture.

An alternative architecture that is popular (and more practical) in many NPUs and that would solve many of the above problems is to introduce a classification step that sends all packets of the same flow to the same processor.[16] In such an architecture, explicit mapping of the *flow id* to the RB to be queried is not required and many of the hardware structures shown in Fig. 2 can be done away with. The main drawback of this scheme is that the PE's could be non-uniformly loaded (e.g., when packets of some flows arrive at a faster rate than others) resulting in performance degradation.

4. SIMULATION METHODOLOGY AND RESULTS

The goal of this paper is to understand if placing data in different RB's (data placement is done indirectly by determining which RB to query/update) using the high level concept of packet flows is beneficial or not. At the time of writing this paper, a complete multiprocessor simulation environment (as proposed in the previous section) was not yet available. Hence, we use a single processor model in our evaluations.

Before proceeding any further we shall briefly digress to understand the state-of-the-art in network processor benchmarks. This will enable us to understand the current problems facing the network processor benchmarking community and justify our use of certain benchmarks in this paper. Benchmarking network processors is a challenging problem since they employ widely varying (heterogeneous) hardware architectures with diverse programming models encompassing a myriad of applications[17, 18] Efforts are being made by the NPF Benchmarking Working Group[19] to standardize benchmarks for NPUs that is widely acceptable to vendors and that allows a fair comparison between NPUs. Benchmarks for NPU can be classified in a hierarchical manner into system-level, function-level, micro-level and hardware-level classes. Two popular NPU benchmarks that are publicly available are CommBench[20] and NetBench.[21] The benchmark programs in both are divided into Header and Payload Processing Applications (HPA/PPA) and consist of small program kernels belonging to the micro-level and function-level categories that are more "general-purpose" than network specific.

Table I. Base IR and Speedup for Different RB Configurations. R = % Instructions Reused; S = % Improvement in IPC. Reduction in Memory Traffic for a (32,8) RB Is Shown in the Last Column

Benchmark	32,4 R	32,4 S	128,4 R	128,4 S	1024,4 R	1024,4 S	Mem Traffic
FRAG	7.9	3.7	20.4	4.9	24.4	8.3	42.1
DRR	12.6	0.16	15.5	0.5	18.2	0.86	11.6
RTR	15.2	3.8	33.2	6.1	47.6	8.1	71.3
REED ENC	19.8	2	20.3	2.05	25.2	2.95	8.7
REED DEC	6.6	1.76	11.8	4	16.6	5.6	4.9
CRC	19.1	19.6	20.7	19.84	21.8	19.84	35.1
MD5	1.4	1.3	3.5	2.3	14.2	8.3	34.3
URL	18.8	9.4	19.9	11.2	22.2	12.7	42

We modified the SimpleScalar[22] simulator (MIPS ISA) and used the default configuration[22] to evaluate IR on a subset of programs representative of different classes of applications from CommBench[20] and NetBench[21] (see Table I) (these were used primarily due to lack of standardized set of NPU benchmark programs). It must be noted that we use SimpleScalar since it is representative of an out-of-order issue pipelined processor with dynamic scheduling and support for speculative execution. 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 enables easy comparison with some statistics obtained in Refs. 20 and 21 that use the same environment. Though IR can be exploited in statically scheduled processors as well as in in-order issue processors, the returns obtained will be much lower since dependent instructions cannot be issued early.

We assume that the reuse test can be performed in parallel with instruction decode just as in Ref. 5. The simulator is modified so that instructions query the RB's during the decode/register read stage. Basically, the flow aggregation scheme requires the *flow id* of every packet to be determined and an appropriate RB to be queried. The RB to be used by instructions operating on a packet is determined by computing the LPM for the packet and storing it in a table. Packets are processed in sequence since Simplescalar is a uniprocessor single threaded simulator. Every application has a certain function (or piece of code) that reads a new packet. When the PC of an instruction matches the PC of the function that reads a new packet, the output port for the packet is read from a precomputed table of output ports. This identifies the RB to be used for the current set of instructions being processed. The RB identifier is stored in the Register Update Unit (RUU) along with the operands for the instruction and the appropriate RB is queried to determine if the instruction can be reused.

The 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.[23] We evaluate IR 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 1, 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. A realistic evaluation of network processing applications is possible if benchmarks that truly represent real world applications are available. Traffic traces available at various sites in

the Internet are usually anonymized (sometimes with a lot of fields removed) making it difficult to analyze IR which is highly data dependent. For some of our experiments, we use the traffic traces collected at the University of Buffalo and Columbia University (these traces are available at http://pma.nlanr.net/PMA/) since they have more realistic packet contents (although anonymized, including interface numbers which can be used for aggregation) compared to other traces.

4.1. Reuse and Speedup—Base Case

Table I shows IR uncovered by different RB configurations and speedup for the benchmark programs considered. Results indicate that network-processing applications exhibit substantial IR which depends on the correlation in packet data. The locality of data in network traffic varies depending on where packets are collected. For example, we can expect high locality at edge routers while the actual advantage that can be derived from locality depends on the working set sizes which is usually large in most applications.[24] It should also be noted that there is no direct relation between hits in the RB and the speedup achieved since speedup is governed not only by the amount of IR uncovered but also by the availability of free resources and the criticality of the instruction being reused. Though resource demand is reduced due to IR, 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 (1024,4) RB. A closer examination of the DRR program reveals that though it is loop intensive, it depends on the packet length that varies widely in our simulations. A more significant reason for the low gain in speedup is due to the high resource contention (see Fig. 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 in DRR for a (32,8) RB.

4.2. Flow Aggregation Results

Instruction reuse and speedup that can be achieved due to flow aggregation depends significantly on the traffic pattern. We carried out the simulation 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 since only load instructions utilize the *memvalid* and *address* fields in the RB. We find that the

flow aggregation scheme with 2 RB's is sufficient to uncover significant IR for most benchmarks considered. We report results only for those cases for which the returns are considerable (best case results). The flow aggregation scheme is based on the output port with a simple mapping scheme (for RTR benchmark we use the input port scheme since this is the program that computes the output port). We map instructions operating on 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 expected if other characteristics of network traffic are exploited in determining a mapping. 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 best-case speedup improvement due to flow aggregation for FRAG and RTR programs over the base scheme. A breakdown of the contribution due to reusing ALU and load instructions is also shown. Flow aggregation is capable of uncovering significant amount of IR; sometimes even when smaller RB's are used (this is highly dependent on the input data). For example, in case of the FRAG program, three RB's with (128,8) configuration results in the same speedup as a single RB with a (1024,8) configuration. We carried out experiments with other traces and obtained varying amounts of IR and speedup. While IR invariably increases due to the flow aggregation scheme (we also obtained results in which base reuse is better than flow aggregation for some programs), speedup, being dependent on other factors (such as criticality, availability of resources), shows little improvement in many cases. To examine the effect of reducing *resource contention* on speedup, we tried the *extended configuration* and obtained a 4.8% improvement in speedup for RTR (2.3% for FRAG) over the flow-based scheme (with (32,8) RB). Determining IR and

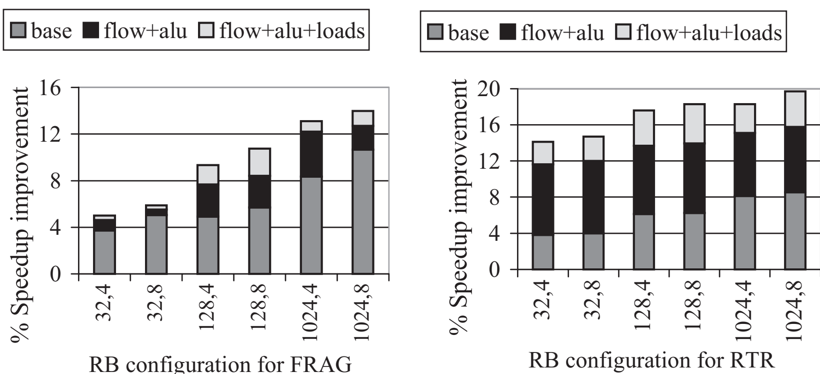

Fig. 3. Speedup improvement due to IR for FRAG and RTR.

speedup due to flow aggregation for payload processing applications is rather difficult since most network traffic traces are anonymized (with the payload portion being removed). We used the inputs provided with the benchmarks as well as data traffic from our LAN environment to carry out experiments related to payload applications. This obviously is not very precise and does not emulate real world traffic since packets within a LAN environment are usually highly correlated resulting in optimistic results. Speedup improvement between 2 to 4% (best case results) was obtained for these benchmarks. Results of IR with flow aggregation for payload processing applications are not shown in this paper since it requires further evaluation with realistic traffic payload.

For some other traffic traces (e.g., that collected from the University of Buffalo) the returns due to flow aggregation are not too large (see Table II). In many of the simulations with the FRAG benchmark, we found that a non-partitioned RB results in better hits indicating that the normal IR scheme is good enough. This usually happens when the size of the RB is less than 1024 entries. We observe that flow aggregation with 2 RB's of (512,2) configuration yields a larger number of RB hits than a single RB (base reuse) of (1024,2) configuration. The results indicate (as expected) that for small size RB's, a single signature per PC (i.e., **(x, 1)** RB configuration) uncovers higher reuse as the RB now contains a larger set of static instructions. As the RB size is increased and the working set of instructions is nearly captured in the RB, the number of different instances per PC (values generated) significantly influences the amount of reuse that can be exploited. A comparison between results obtained for FRAG indicates that traces collected at the University of Buffalo (Table II) have higher repeatability (and larger hit rates) in data values compared to the randomly generated values used in Table I.

Table II. A Comparison between Normal IR and Reuse Due to Flow-Aggregation Based on the Input Interface for the FRAG Benchmark

Base Reuse (1 RB)		Flow Reuse (2 RB)	
RB Config	% hits	RB Config	% hits
(32,1)	8.86	(16,1)	7.15
(16,2)	4.58	(8,2)	2.81
(128,1)	19.98	(64,1)	14.5
(64,2)	17.35	(32,2)	10.6
(1024,1)	20.11	(512,1)	18.42
(512,2)	25.8	(256,2)	26.12
(2048,1)	20.11	(1024,1)	18.42
(1024,2)	25.91	(512,2)	26.34

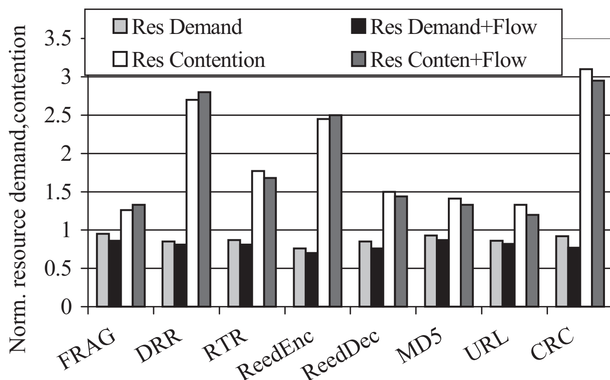

Fig. 4. Resource demand and contention due to IR.

4.3. Impact of IR on Resources

Figure 4 shows that resource demand due to flow aggregation is lower than in the base IR scheme. Resource demand is calculated by normalizing the number of resource accesses with IR to that without IR. The figure also shows resource contention normalized to the base case. As mentioned in Section 2, contention may increase or decrease due to IR. We find that resource contention comes down drastically on using the *extended configuration*. Load instruction reuse also reduces memory traffic[11] significantly, (see last column of Table I) which decreases the activity factor over the high capacitance buses making it a possible candidate for low power. Energy savings can also be obtained due to reduced number of executions when an instruction is reused. However, certain amount of energy is spent in accessing the RB. Previous work[12, 25, 26] has shown that an average power improvement of 5.4% can be achieved by exploiting IR. Since a reused instruction executes and possibly commits early, it occupies a RUU slot for a smaller amount of time. This reduces the stalls that would occur as a result of the RUU being full. Flow based reuse has the ability to further improve IR thereby reducing the RUU occupancy (1.5 to 3%) even more compared to the base scheme.

5. CONCLUSIONS AND FUTURE WORK

In this paper we examined instruction reuse in a few packet-processing applications. To further enhance the utility of reuse by reducing inter-

ference, a flow aggregation scheme that exploits correlation in packet data was proposed. The flow-based scheme utilizes multiple RB's (or a single partitioned RB) to uncover larger reuse and speedup. A best-case speedup improvement varying between 1 and 20% was obtained when both ALU and load instructions were exploited. Further, since the flow aggregation scheme reduces memory accesses even more, the number of bit transitions on the processor memory bus decreases further reducing energy consumption. This is probably compensated to some extent due to accesses to the Reuse Buffer. We find that the hit rate in the RB with flow aggregation is higher than base reuse only when the size of the RB is sufficiently large. This however varies between applications and input data sets. We are in the process of carrying out a detailed evaluation of the architecture proposed along with suitable power models to estimate the effectiveness of the flow-based scheme. Initial results on a multiprocessor simulator indicate that reuse is higher within each processing element due to localization of instructions and data.

Exploiting flow aggregation to recover larger reuse is not restricted to header processing applications alone. Internet traffic studies show that significant temporal correlation exists in network traffic which could be exploited by the flow based scheme for payload processing applications too. We will investigate this more extensively in future work. The only problem we foresee is in obtaining real packets whose headers and payload are both not anonymized. An important issue that is worth considering is to 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. Similarly, rather than using the plain LRU replacement policy, a policy that combines LRU and the notion of criticality will yield better results. Further, a detailed exploration of various mapping schemes is necessary to evenly distribute related data between RB's. These will be carried out as part of future work.

REFERENCES

1. P. Paulin, Network Processors: A Perspective on Market Requirements, Processors Architectures, and Embedded S/W Tools, *Proc. of the Design Automation and Test in Europe Conference (DATE)*, pp. 420–427 (2001).
2. M. H. Lipasti and J. P. Shen, Exceeding the Dataflow Limit via Value Prediction, *Proc. of the 29th International Symposium on Microarchitecture*, pp. 226–237 (December 1996).
3. Y. Sazedis and J. Smith, The Predictability of Data Values, *Proc. of the 30th Annual International Symp. on Microarchitecture*, pp. 248–258 (December 1997).
4. M. Lipasti, C. Wilkerson, and J. Shen, Value Locality and Load Value Prediction, *Proc. of ASPLOS-VII*, pp. 138–147 (October 1996).

5. A. Sodani and G. Sohi, Dynamic Instruction Reuse, *Proc. of the 24th Annual International Symposium on Computer Architecture (ISCA)*, pp. 194–205 (July 1997).
6. A. Sodani and G. Sohi, Understanding the Differences Between Value Prediction and Instruction Reuse, *Proc. of the 32nd Annual International Symposium on Microarchitecture*, pp. 205–215 (December 1998).
7. A. Sodani and G. Sohi, An Empirical Analysis of Instruction Repetition, *Proc. of ASPLOS-VIII* (1998).
8. C. Molina, A. Gonzalez, and J. Tubella, Dynamic Removal of Redundant Computations, *Proc. Intn'l. Conf. on Supercomputing* (June 1999).
9. D. Connors, H. Hunter, B. C. Cheng, and W. M. Hwu, Hardware Support for Dynamic Activation of Compiler Directed Computation Reuse, *Proc. ASPLOS-IX*, pp. 222–233 (November 2000).
10. D. Connors and W. M. Hwu, Compiler-Directed Dynamic Computation Reuse: Rationale and Initial Results, *Proc. of the 32nd International Symp. on Microarchitecture* (December 2000).
11. J. Yang and R. Gupta, Load Redundancy Removal Through Instruction Reuse, *Proc. of the Intn'l Conf. on Parallel Processing*, pp. 61–68 (August 2000).
12. D. Brooks, V. Tiwari, and M. Martonosi, Wattch: A Framework for Architectural-Level Power Analysis and Optimizations, *Proc. of 27th ISCA* (June 2000).
13. F. Baker, Requirements for IP Version 4 Routers, *RFC—1812*, Network Working Group (June 1995).
14. Intel IXP1200 Network Processor—Hardware Reference Manual, Intel Corp. (December 2001).
15. P. Crowley, M. Fiuczynski, J. Baier, and B. Bershad, Characterizing Processor Architectures for Programmable Network Interfaces, *Proceedings of the International Conference on Supercomputing* (May 2000).
16. B. Chen and R. Morris, Flexible Control of Parallelism in a Multiprocessor PC Router, *Proc. of the 2001 USENIX Annual Technical Conference*, pp. 333–346 (June 2001).
17. P. Chandra, F. Haday, R. Yavatkar, T. Bock, M. Cabot, and P. Mathew, Benchmarking Network Processors, *Proc. of the Workshop on Network Processors, 8th International Symp. on High Performance Computer Architecture* (February 2002).
18. M. Tsai, C. Kulkarni, C. Sauer, N. Shah, and K. Keutzer, A Benchmarking Methodology for Network Processors, *Proc. of the Workshop on Network Processors, 8th International Symp. on High Performance Computer Architecture* (February 2002).
19. S. Audenaert and P. Chandra (NPF Benchmarking Working Group co-chairs), *Network Processors Benchmark Framework*, NPF Benchmarking Workgroup, www.npforum.org.
20. T. Wolf and M. Franklin, CommBench—A Telecommunications Benchmark for Network Processors, *IEEE Symposium on Performance Analysis of Systems and Software*, pp. 154–162 (April 2000).
21. G. Memik, B. Mangione-Smith, and W. Hu, NetBench: A Benchmarking Suite for Network Processors, *Proc. of ICCAD* (November 2001).
22. 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).
23. S. Bradner and J. McQuaid, A Benchmarking Methodology for Network Interconnect Devices, *Request For Comments—2544*, Internet Engineering Task Force (IETF) (March 1999).
24. S. Melvin and Y. Patt, Handling of Packet Dependencies: A Critical Issue for Highly Parallel Network Processors, *Proc. CASES* (2002).

25. D. Citron, D. Feitelson, and L. Rudolph, Accelerating Multimedia Processing by Implementing Memoing in Multiplication and Division Units, *Proc. of ASPLOS-VIII*, pp. 252–261 (October 1998).
26. M. Azam, P. Franzon, W. Liu, and T. Conte, Low Power Data Processing by Elimination of Redundant Computations, *Proc. of ISLPED* (1997).