Published in the *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES),* October 8-11 2002, Grenoble, France.

# Handling of Packet Dependencies: A Critical Issue for Highly Parallel Network Processors

Stephen Melvin
FlowStorm, Inc.
650 Saratoga Ave.
San Jose, CA 95129
+1 415-608-5149
melvin@zytek.com

Yale Patt
University of Texas, Austin
ENS 541
Austin, TX 78710
+1 512-471-4085
patt@ece.utexas.edu

## ABSTRACT
Network processors are being asked to perform increasingly complex operations on packets of information at faster and faster rates. Because processor performance and memory cycle times are not keeping up with this demand, there is a fundamental need for simultaneous processing of multiple packets, and the degree of this parallelism is increasing. Sometimes a dependency exists between two packets currently being operated on, and as the ratio of packet processing time to packet transmission time increases, these dependencies are more likely to impact performance. Thus, the way packet dependencies are handled will become critical. In this paper we show that there is potentially a dramatic difference in performance between optimal and non-optimal solutions. We argue that this is the key challenge that must be addressed in highly parallel network processors. We discuss how work in thread level speculation relates to this problem and describe a practical hardware implementation that requires little or no changes to software and with near optimal performance.

## Categories and Subject Descriptors
C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors); C.3 [**Special-Purpose And Application-Based Systems**] – *Real-time and embedded systems*; B.3.2 [**Memory Structures**]: Design Styles - *Associative memories, Cache memories, Shared memory.*

## General Terms
Measurement, Performance, Design, Experimentation.

## Keywords
Network processors, packet processing, processor architecture, memory synchronization, multithreaded processors, parallel processing, packet dependencies, thread level speculation.



## 1. INTRODUCTION
Internet networking hardware involves processing of packets of information for many purposes and at many stages in a network. Routers, firewalls, gateways, load balancers and servers all process packets of information in some way. Where in the network the processing takes place (i.e. in the core or close to the edge) has a great deal to do with what types of processing needs to take place and how fast that processing must occur. In general, processing closer to the core takes place faster and involves less work. For example, many core routers perform only layer 2 packet forwarding (i.e. link layer header modification), which can be done with minimal processing overhead. Edge routers, however, typically perform more functions, such as traffic shaping, monitoring, billing and quality of service enforcement. In both situations, the need for processing is constantly evolving, and there is an increasing need to do more at faster rates.

Two key trends are the increase in network speed and the increase in the amount of processing that needs to take place at each stage in the network. Together these trends are forcing network processing solutions into greater degrees of parallelism. Figure 1 illustrates this point with four different scenarios for a network processor. Here the term "network processor" is used to generally refer to any processing engine that can perform programmable operations on packets of information.

In the first scenario of Figure 1, the processing time of the packet is the same or smaller than the transmission time of the packet. In this scenario, the code need not be concerned with dependencies between packets, and ordinary single-threaded non-parallel processors can be used. In the other scenarios of figure 1, the processing time for a packet is substantially longer than the transmission time of one packet of information. The common trend is that the need for more complex operations (and thus larger workloads) and/or the increase in network speeds has lead to these situations.

In many cases the workload time is dominated by memory latency due to poor locality of data references and large working set sizes. This means that the limitation on packet throughput is driven by memory throughput, which has tended to increase at a rate even slower than single-threaded processor performance, further driving packet processing solutions into parallel packet processing scenarios.

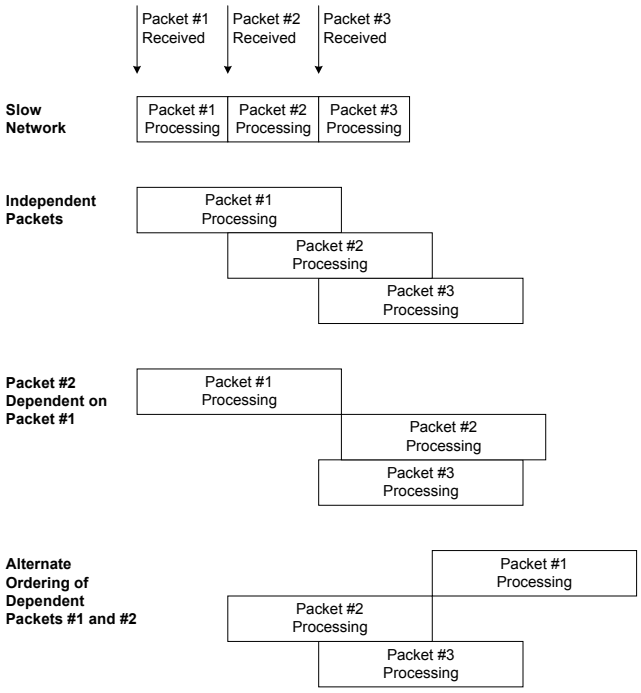


**Figure 1: Packet Processing Scenarios**

In the case that all packets can be operated on independently, as shown in the second scenario of figure 1, processing can be pipelined neatly and no conflict arises between code processing simultaneous packets. This would be the case in certain types of stateless firewalls and forwarding engines, where each packet is evaluated according to static rules and does not depend on any other packets. Thus, no state is changed by packet that affects a future packet. The forwarding tables and firewall rules might be dynamically modified, but this typically happens on a time scale orders of magnitude greater then the time to process a single packet. A parallel packet processing solution for this second scenario is relatively easy to implement. The code working on one packet need not be aware of other packets and there is no need to synchronize memory operations between packets.

In the more general case that dependencies can arise between packets, a more complicated situation exists. This is shown in the third and fourth scenarios of figure 1. This would be the case if both packets are from the same TCP connection and due to, for example, encryption or TCP state maintenance, there is a need to update state in memory between the processing of the two packets. One or more memory locations written by one packet will be read by the other packet. Note that packet #3 in these scenarios is independent from both packets and can be processed as soon as it arrives.

Other examples in which packet dependencies can arise would be the updating of traffic management counters and the updating of routing or address translation tables. In the latter case, two packets may be dependent even if they are from completely independent connections if they hash to the same table entry. One packet may want to modify a table entry while another packet is querying the same entry. The fourth scenario in figure 1 illustrates that in some, if not most cases it does not matter which order two dependent packets are processed, as long as they are serialized to prevent incorrect results.

In these cases where simultaneous processing of packets is required, and where dependencies can exist between packets, it can be complicated to enforce those dependencies. Currently, there are two common approaches to this problem. The first solution is a software solution, where software locks are included in the code to cause dependent packet processing to be delayed until an earlier packet has been completed. These software semaphores are used to lock out subsequent dependent packets from accessing state until the first packet has updated it. In [2], different locking scheme are discussed and it is shown that TCP packets are limited to about a speedup of 2 on a per-flow basis. The second solution involves hardware, where packet classification hardware serializes all packets that can possibly be dependent. In a multiprocessor, this can involve generating a hash function that sends all packets of the same flow to the same processor, and distributes the load across multiple. In [3], a multiprocessor PC router is described that relies on a hashing and classification step.

Unfortunately, packet processing code is often large and complex and modifying it to incorporate new locking mechanisms is not trivial. Even when such code is relatively simple, guaranteeing that software locks have been correctly programmed for all possible network traffic scenarios can be hard to verify. Furthermore, requiring hardware to enforce sequentiality when it is not needed lowers performance. This is because often the entire packet processing is not dependent such that a partial overlap is possible. The importance of a partial overlap of packet workload can be appreciated by referring to figure 2. In the case that a packet reads data as its first instruction and writes that same address as its last instruction, indeed there can be no overlap of processing. This is generally not the case however. The second scenario of figure 2 illustrates the case that the second packet can start before the first packet is completed, even though they are dependent. It is also the case that due to conditional branches, packets that are sometimes dependent may not always be dependent. Thus conservative locking and large grained locking can yield significantly sub-optimal solutions.

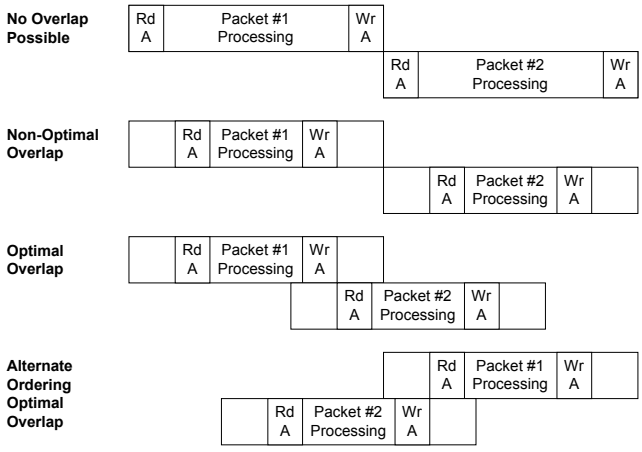


**Figure 2: Overlap of Dependent Packets**

It is also the case that hardware solutions that group flows for multiprocessors suffer from the problem of guaranteeing that the grouping is relatively uniform over time in order to balance work across the multiple processing elements. The classification of packets to direct them to processing elements is constrained by having to preserve correctness and can't take advantage of a more dynamic load balancing approach.

This paper is divided into five sections. In the next section we describe a simulation study in which we measured potential packet dependencies on real network traffic. Section three explains how optimally enforcing packet dependencies relates to work in conventional multithreaded speedup. Section four introduces a hardware model for near optimal enforcement of packet dependencies using no software changes. Finally section five offers conclusions.

## 2. SIMULATION STUDY

We studied actual packet traces and simulated them against workload models. We chose to use abstract models of workloads rather than actual benchmarks for several reasons. First, applications are constantly evolving and choosing an application today that will be representative of future applications is difficult at best. Second, the real world processing taking place on most equipment is often proprietary and not in the public domain. Finally, we are initially more interested in high-level characteristics of the applications being run, rather than on individual memory accesses and sequences of instructions.

In contrast, we felt it was important to simulate based on real world packet traffic rather than on analytical models or synthetic traces. The performance of dependency enforcement hardware and the characteristics of packet dependencies are very much driven by the patterns of packets flowing and these are very hard to create without actual measurements.

The packet traces used in this simulation study were received from he National Laboratory for Applied Network Research (NLANR) [1]. Our simulations measured packets captured on two OC12 links, which run at 655Mbps. The first was the NASA Ames link to the MAE West exchange. Approximately 100.2 million packets, taken over a 2-day period in August 2001 were simulated. The second OC12 link was at the Indiana University GigaPOP. Approximately 107.0 million packets, taken over a 2-day period in November 2001 were simulated. The NASA link averaged approximately 92,000 packets per second and the Indiana link averaged approximately 83,000 packets per second for the traces analyzed.

The packet traces were evaluated for packet dependencies to determine how often packets from the same flow were in proximity to one another. Here we defined "flow" in the usual sense of meaning the same source and destination IP addresses and the same source and destination UDP or TCP port numbers. We measured all flows in addition to only TCP flows for comparison. In many applications, TCP flows exhibit more dependency restrictions than UDP or other protocol flows [2].

The result of the packet dependency analysis is shown in figure 3. This graph illustrates the probability that a given packet will be dependent on some other packet given a certain packet window size. For example, the entry for 100 packets for the NASA workload considering only TCP flows, is 14.7%. This means that if the previous 100 packets are currently being worked on, there is a 14.7% probability that the next packet received will be dependent on one of those packets. Figure 3 illustrates that even at these relatively high rates, where a large number of flows are aggregated into a single link, packet dependencies are significant. As network processors exploit more packet-level parallelism, the packet window necessarily increases. Packet window sizes of 100 or even 1000 are not unreasonable given large workloads and faster network rates. Thus, it is clear that efficient handling of packet dependencies is a significant issue.

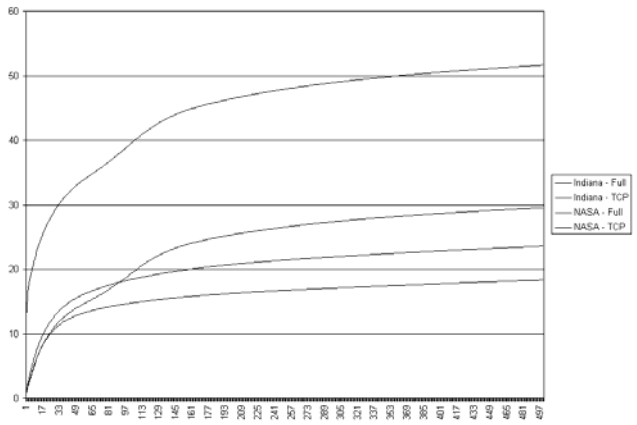


**Figure 3: Dependency Probability vs. Window Size**

The next set of measurements performed on the packet traces was to measure a variety of packet dependency models for a fixed workload size of 500 microseconds. Workload dependency models were used to characterize the relationship between packet operations as they relate to other packets. The following 7 models were simulated with the following model codes. In all models, packets from the same TCP flow were considered dependent.

| Model | Description |
|---|---|
| **none** | no overlap |
| **ovl25** | 1/4 of workload overlapped |
| **ovl50** | 1/2 of workload overlapped |
| **ovl75** | 3/4 of workload overlapped |
| **pr25** | 25% probability of overlap |
| **pr50** | 50% probability of overlap |
| **pr75** | 75% probability of overlap |

The importance of a partial overlap of packet workload was discussed above and is illustrated in figure 2. Probabilistic overlap is also significant. An optimal system can overlap only when a conflict actually occurs, allowing situations where not all packets from the same flow actually conflict to operate in parallel. The

partial overlap and probabilistic overlap models were simulated to explore the potential speedup that an optimal system can achieve over a software or alternative hardware solution that overlaps packets more conservatively.

Figure 4 illustrates the total number of packets that are active for each of the dependency models. It shows that even though the average number of active packets is relatively constant, around 50, the *maximum* number of packets drops dramatically from over 1000 to under 200. This shows that a system implementing optimal overlap of packets can potentially greatly reduce the amount of queuing hardware, processing engines, and/or extra packet latency.

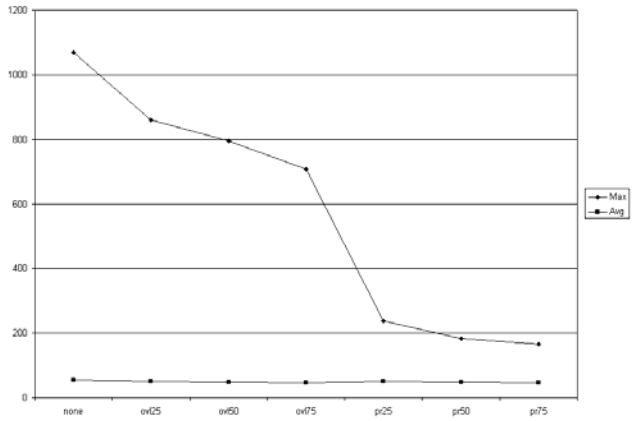


**Figure 4: Number of Packets Simultaneously Processing**

Figures 5 and 6 illustrate the maximum and average latency for each of the dependency models. The maximum latency in a fully non-overlapped system is almost 400,000 microseconds. This represents long dependency chains for dependent TCP packets. Partial and probabilistic overlap greatly reduces this maximum latency. If only 25% of the packets of a flow are truly independent (pr25), this maximum latency is cut to under 12,000. Average latency is also reduced, from about 600 in the totally non-overlapped case, to a near-optimal 507 in the case of 3/4 of the workload overlapped (ovl75 model). Thus, it is clear that handling optimal overlap in the partial and probabilistic cases, something not possible with today's network processors, could become a significant performance issue when hundreds of packets are being processed simultaneously.

# 3. APPLICABILITY OF CONVENTIONAL MULTITHREADED TECHNIQUES

Optimally enforcing packet dependencies in a highly parallel environment relates in very general terms to the area of multithreaded processing. Much work has been developed to speedup conventional multithreaded workloads on general-purpose processors where dependencies exist between threads [4, 5, 6, 7, 8, 9, 10, 11, 12, 13]. Much of this work involves hardware support for speculative execution in conjunction with compile-time optimizations and/or the optimization of software synchronization primitives. It would be nice to be able to apply these techniques to network processing

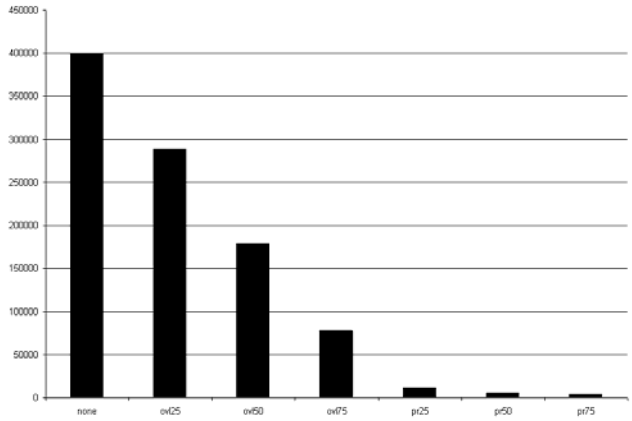


**Figure 5: Maximum Latency**

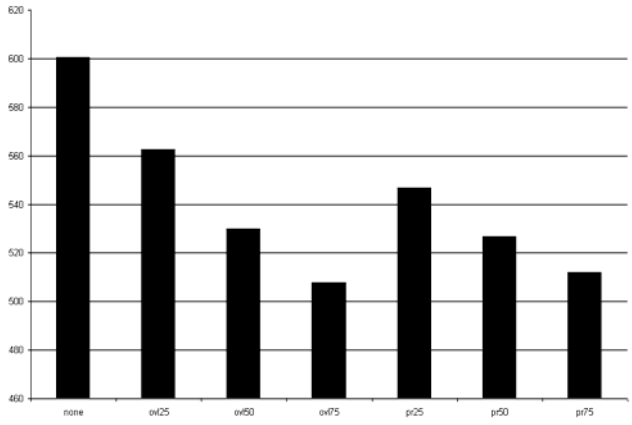


**Figure 6: Average Latency**

.

Unfortunately little of the work in thread level speculation can be applied directly to packet processing scenarios. In a packet processor, the work that is being performed on multiple packets in parallel cannot easily be treated as a single sequential program since the order of packet arrival is unpredictable. A more correct view is that each packet instantiates a separate copy of the same single-threaded program. Also in cases where order of arrival must be preserved, the correct sequencing of these workloads is an external constraint visible only to hardware. Thus there must be some mechanism to convey this information to speculative execution hardware. None of the references cited above have analyzed network specific workloads on packet data streams.

Steffan et al. [9] describe an approach to thread level speculation in which cache coherency mechanisms are enhanced to detect dependencies between threads. In principle each packet could be treated as an *Epoch* in this scheme with hardware enforcing data dependencies. However, this relies on coherent cache mechanisms and often network processors have little or no caching since locality is often low. Furthermore it may not be

sufficient to merely detect data dependency conflicts. As we will describe in more detail below, an optimal solution requires prediction of conflicts and packet processing contains many opportunities to drive conflict prediction.

Martinez and Torrellas [11] describe a method of allowing locks to be executed speculatively, such that the performance of fine-grained locking can be achieved without the programming effort. Rajwar and Goodman [12] describe a very similar technique. While this technique could be applied to packet processing, it would require that locking primitives be programmed into packet workloads. It is possible in principle to go further than these techniques and allow fully automatic synchronization (no locking constructs in software at all). Also, it may not be sufficient in some applications to merely have software locks without some other hardware mechanism, since this alone does not guarantee that packet workloads acquire the locks in the order of arrival.

In the Multiscalar paradigm [6], a sequential program is broken down into tasks that can be executed in parallel with hardware enforcing inter-task dependencies. This work relates to the general area of hardware support for parallel execution of multiple threads. However in packet processing each packet typically operates within its own context. There is no sharing of registers or control flow between packets; the only dependencies are expressed through shared external variables.

The Transactional Memory of Herlihy and Moss [13] offers the ability to do a conditional commit and software retry on failure. In theory this could be applied to packet processing if each packet is considered to be its own transaction. However it may not be as optimal if conflicts are only detected at the end of the packet. Also, it still does not solve the problem of guaranteeing that the true order is based on packet order of arrival, not on which packet tries to commit first.

# 4. HARDWARE ENFORCED PACKET DEPENDENCIES

In this section we describe a hardware mechanism for enforcing packet dependencies without any changes in software. This mechanism has the important advantage that packet dependencies are enforced by hardware only when required and potentially with no discarded work. The software can be written with the view that each packet is handled in order of arrival with no overlap with subsequent packets. This mechanism optimizes performance for the common case that no dependencies are present and doesn't require the hardware to be pre-configured with knowledge of what packet sequences will and will not be dependent. We present this mechanism as a proof of concept, to show that practical solutions can be developed for network processing that allow near optimal enforcement of packet dependencies.

## 4.1 Network Processor Assumptions

We consider a generalized network processor in which multiple packets are processed simultaneously. This can be implemented using a multiprocessor, a multithreaded processor or a combination of both. The mechanism is not dependent on the type of processor used. It would be useful in any form of network processor in which some form of simultaneity exists such that more than one packet at a time is being processed.

There are a few requirements placed on the network processor. The first is that a time stamp or sequence number is associated with each packet as it arrives. The sequence number is used to enforce the sequential processing model. When a packet enters the processor, a new sequence number is generated and that sequence number is associated with the instructions that are processing that packet. The network processor is required to include this sequence number with all memory operations (reads and writes) performed. An additional requirement is that when a packet enters the processor and when a packet leaves the processor (or when processing ends for that packet), a notification of that event is made to the hardware. The information about when a packet enters and leaves is used to know when write data can be committed to memory.

Finally, the network processor must have the ability to restart processing for each packet. To accomplish this a signal is generated that indicates that a conflict has been detected, which requires the network processor to discard all work associated with the indicated sequence number and to restart processing that packet. The processor does not have to reconstruct any information, but only to start from the beginning of the packet again assuming that all memory writes have been discarded. It must be possible to restart a packet even after packet processing has ended for that packet. This is needed because an older packet that is still processing may do something that nullifies the processing that has already been completed by a younger dependent packet.

## 4.2 The Packet Dependency Mechanism

The hardware mechanism described in this section is responsible for guaranteeing that the results obtained while simultaneously processing multiple packets are the same as would be obtained if all packet processing occurred in the sequence of packet arrival. This hardware mechanism is placed between the packet processing engine and the memory system.

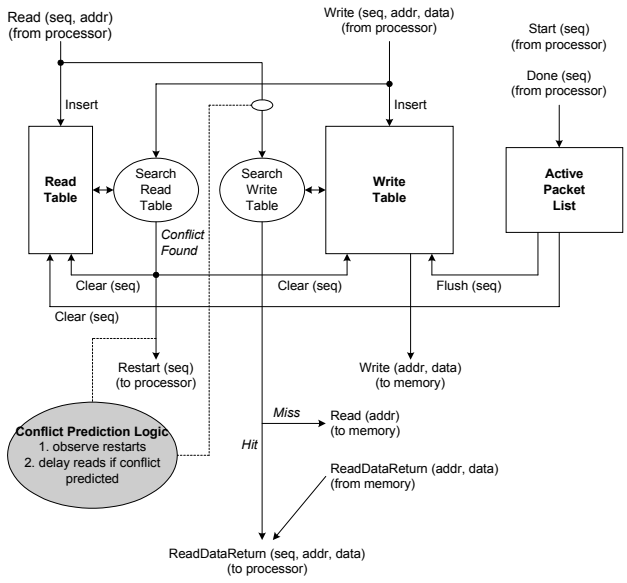


**Figure 7: A Dependency Enforcement Mechanism**

The network processor generates four types of commands to the mechanism: memory read, memory write, packet start and packet done. Each command includes a sequence number associated with the packet for which the operation applies. Returning to the network processor are the read data returning, and a restart signal that is used to indicate to the network processor that a particular packet needs to be restarted.

Figure 7 illustrates a diagram of the implementation of the mechanism. There are two tables, a "Read Table" and a "Write Table" at the core of the design. The Read Table records all memory reads. The sequence number and the address are recorded, but not the data. Each time a read is accepted from the network processor, an entry is inserted in the Read Table. To save space in the read table, it may be constructed to not save all address bits, but to discard some of the least significant address bits. If an entry being inserted matches an entry already in the table, it need not be inserted. The Read Table is searched whenever a write is accepted from the network processor to determine if a conflict has arisen.

The Write Table records all writes, including the sequence number, the address and the data. Each time a write is accepted from the network processor, an entry is made in the Write Table. To save space in the Write Table, an entry can include a larger amount of storage than a single reference can generate. For example a Write Table entry could contain 64 bytes of data with valid bits associated with each byte. This would allow multiple writes to the same 64-byte area with the same sequence number to be combined into the same entry. Two different accesses with different sequence numbers cannot be combined. The Write Table is searched whenever a read is accepted to determine if data should be bypassed from the memory system. A record is kept of the fact that data has been forwarded from one packet to another so that when a packet is restarted; all packets potentially corrupted by that packet are also restarted.

When a sequence number is completed, the active packet list is updated and if the oldest packet active is no longer being processed, data from the Write Table is flushed to memory and all entries for that sequence number are cleared from the Read Table.

In summary, the operations that are performed for each type of command from the network processor are as follows:

Read

1. Insert entry into Read Table including sequence number and address.
2. Search Write Table for the newest matching entry that has the same or an older sequence number. If one exists, forward that data back to the network processor. If none exists, send this request to the memory system.

Write

1. Insert entry into Write Table including sequence number, address and data
2. Search Read Table for any entry that is newer than this sequence number and matches the address. If one is found, then:
   a. Signal restart to network processor of the sequence number associated with the match
   b. Delete all Read Table entries with that sequence number
   c. Delete all Write Table entries with that sequence number

   Repeat steps a, b and c. if there are multiple matching sequence numbers, and continue down all dependency chains

Start

1. Add entry to active packet list

Done

1. Mark packet as done in active packet list
2. If packet is the oldest done
   a. Delete all Read Table entries for that sequence number
   b. Flush all Write Table entries for that sequence number to memory
   c. Delete all Write Table entries for that sequence number
   d. Delete the packet from the active packet list

   Repeat steps a. through d. if the next oldest packet is done

A possible performance enhancement would be to save the data that is returned for each read in the Read Table. This would allow the conflict detection logic to determine if in fact wrong data had actually been returned, and only signal a restart if a data mismatch occurred. This would be advantageous if the packet processing code commonly overwrites data with the same value as is already stored, which may happen in certain applications. In addition, carefully written software could take advantage of a network processor with this enhancement by purposefully writing an expected value into a variable early in its processing. In the common case, when the final value is written it will be the same and subsequent packets will not need to be restarted.

Another performance enhancement that would allow the amount of discarded work to be limited would be to have a checkpoint-backup mechanism within the network processor in conjunction with the use of sub-sequence numbers. The processor would then be able to restart processing from a safe checkpoint rather than having to back up to the beginning of the packet processing.

In many cases, it is not necessary to guarantee that processing yields the same results as if packets had been processed in the specific order of arrival, as long as the results are correct for any specific order. In other words, the packet processing code does not care which packet is processed first as long as there is a consistent order. Packets may in principle be reordered by the network in many cases and higher-level protocols cannot depend on any specific order. This principle has been illustrated in the last scenarios of figures 1 and 2 above. (However, it is important to note that there are sometimes performance implications to reordering packets if code is written to optimize for cases where dependent packets are processed in order.)

An alternative mode of operation for the packet dependency mechanism described above is to not enforce a specific order, but to signal a conflict only when no consistent order exists. In this mode the packet sequence number is being used as merely a packet identifier rather than a numeric order. The mechanism is modified so that when a write occurs, a packet sequence is defined for any previous read. Then when two packets resolve to two conflicting sequences, one must be restarted.

## 4.3 Conflict Prediction and Restart Processing

The mechanism described above can be used to enforce the correctness of a sequential packet processing model in a processor that processes packets simultaneously. Ideally, there should a minimal amount of discarded work. (The importance of minimizing discarded work is particularly relevant in multithreaded processors, where multiple threads all contend for the same execution resources.) In cases where conflicts are common, more careful optimization of the restart mechanism should be implemented. Figure 8 illustrates three difference scenarios for handling packet conflicts.

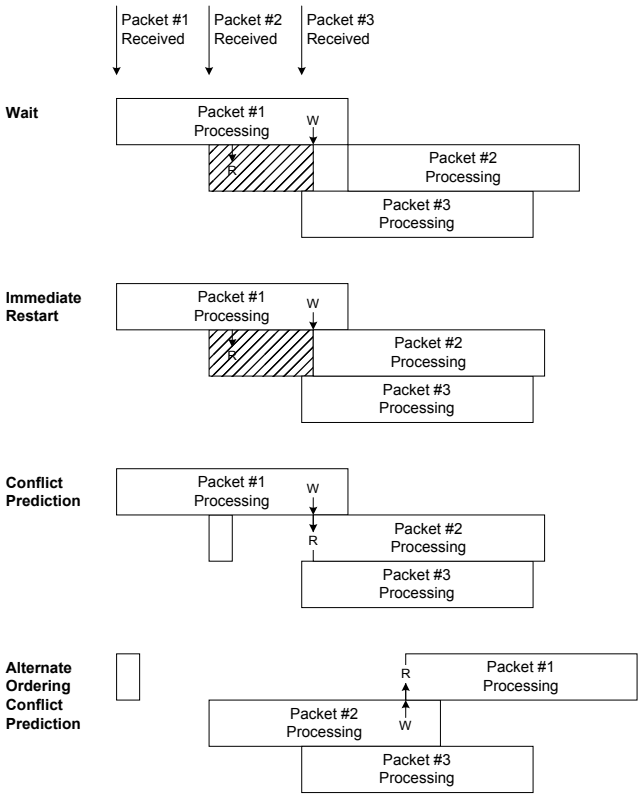


**Figure 8: Conflict Detection Processing Examples**

In this figure the arrow and the *R* represents a read in packet #2 for which a conflict arises, and the arrow and the *W* indicates a write in packet #1 that is the source of the conflict. In the first scenario, as soon as the conflict is detected, processing is stalled until the packet that generated the conflict has completed. This can be accomplished by having an additional stall signal to the network processor, or by having the mechanism delay the return of the first read operation received after processing resumes for packet #2.

This first scenario is a conservative approach that essentially predicts that if a first conflict between two packets arises, then other conflicts between those two packets are likely to arise and therefore no more work should be expended on the second packet until the first packet is done.

In the second scenario of figure 8, processing is immediately restarted on the second packet after work is discarded and the Read Table and Write Table are flushed. This approach essentially predicts that there is likely to be only one conflict between two packets, so the second packet should not be further delayed and processing should continue immediately. If a second conflict arises then the process is repeated. This may tie up processing resources within the network processor that could be used for performing other tasks. The performance of the first two approaches shown in figure 8 is dependent on the traffic characteristics and on the characteristics of the packet workload.

Many more sophisticated mechanisms are possible. A *conflict predictor* could be implemented that would observe sequences of packet conflicts and develop a history to guess when memory operations would conflict. The information used in the predictor could be information from the packet header of each packet (e.g. source and destination IP number, and source and destination port number), as well as the memory addresses of the read and write involved. Ideally, a hardware conflict predictor would know when a read operation is received that it should not be allowed to complete since a write from an earlier packet is likely in the future. The third scenario of figure 8 illustrates the ideal situation when the hardware predictor is used to stall the memory read until the write from packet #1 occurs. The memory read is then completed and processing continues. In this case, there is no discarded work, and packet #2 is delayed the minimum time necessary to guarantee correct results.

Note that in the third scenario the second packet completes in the earliest time and the least machine resources are used. The advantage of a dynamic hardware conflict predictor is that it can adapt to changing application conditions and only apply stalls to read operations when there is a high likelihood of a conflict. The forth scenario of figure 8 illustrates the opposite ordering of packets #1 and #2 in the case that the read and write occur at the same point. It may be acceptable for the hardware to guarantee either ordering rather than adhere to a strict order of arrival model.

Figure 9 illustrates the maximum amount of discarded work for each of the dependency models discussed in section 2. This represents the amount of work that would be discarded in an optimal solution as a percentage of the total useful work that is performed by the network processor. This work would be discarded if there were no prediction, and using the immediate restart method described above (the second scenario in figure 8). This figure shows that optimal overlap can reduce the amount of discarded work from 20% down to just a few percent. By using conflict prediction, the discarded work can be further reduced from these numbers.

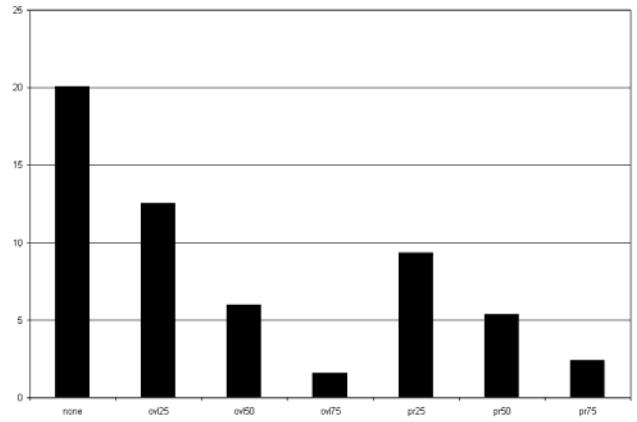


**Figure 9: Maximum Discarded Work (%)**

## 5. CONCLUSIONS

The simulation study presented here shows that as workloads increase relative to processing speed, the problems posed by packet dependencies are severe. Thus, there is a growing need to address packet dependencies where a fully sequential model is impractical and simple solutions are too low in performance. The advantage in performance for optimal solutions is that dependencies are only enforced when needed. This has significant advantages over simpler techniques that enforce dependencies based on broader assumptions.

Average packet latency can be reduced from 20% over the workload time to 1.6% over the workload time if a more optimal overlap is implemented in comparison to solutions (either hardware or software) that overlap packets more conservatively. Maximum packet latencies can be more dramatically reduced from almost 800 times the workload down to just over 8 times the workload in cases where 25% of the packets in a flow are fully dependent and the other 75% are independent.

This paper has also described a hardware mechanism to enforce packet dependencies near optimally. This mechanism involves no software changes so it allows code to be executed without any memory synchronization changes. Thus, a programming model and previously developed code for single threaded uniprocessors can be applied to multithreaded and/or multiprocessing network processors.

## 6. REFERENCES

[1] Passive Measurement and Analysis project, National Laboratory for Applied Network Research. (http://moat.nlanr.net/pma).

[2] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance issues in parallelized network protocols. In Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), pages 125-137, November 1994.

[3] Benjie Chen and Robert Morris, Flexible Control of Parallelism in a Multiprocessor PC Router, Proceedings of the 2001 USENIX Annual Technical Conference (USENIX '01), pages 333 - 346, June, 2001.

[4] M. Franklin and G. Sohi, "ARB: A hardware mechanism for dynamic reordering of memory references," IEEE Transactions on Computers, vol. 45, pp. 552-571, May 1996.

[5] S. Gopal, T. N. Vijakumar, J. E. Smith and G. S. Sohi, "Speculative versioning cache," Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4), Las Vegas, February 1998.

[6] G. Sohi, S. Breach, and T. Vijaykumar, "Multiscalar processors," Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 414-425, Ligure, Italy, June 1995.

[7] J. G. Steffan and T. Mowry, "The potential for using thread-level data speculation to facilitate automatic parallelization," Proceedings of the Fourth International Symposium on High-Performance Computer Architecture (HPCA-4), Las Vegas, February, 1998.

[8] L. Hammond, M. Willey, and Kunle Olukotun, "Data Speculation Support for a Chip Multiprocessor," Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), San Jose, October 1998.

[9] J. Steffan, C. Colohan, Antonia Zhai, and T. Mowry, "A Scalable Approach to Thread-Level Speculation," Proceedings of the 27th Annual International Symposium on Computer Architecture, Vancouver, Canada, June 2000.

[10] M. Cintra, J. Martinez, and J. Torrellas, "Architectural Support for Scalable Speculative Parallelization in Shared-Memory Multiprocessors," Proceedings of the 27th Annual International Symposium on Computer Architecture, Vancouver, Canada, June 2000.

[11] J. Martinez and J. Torrellas, "Speculative Locks for Concurrent Execution of Critical Sections in Shared-Memory Multiprocessors," Workshop on Memory Performance Issues, ISCA 2001.

[12] R. Rajwar and J. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," Proceedings of the 34th Annual International Symposium on Microarchitecture, Austin, Texas, December 2001.

[13] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural support for lock-free data structures," Proceedings of the International Symposium on Computer Architecture, pages 289-300, San Diego, CA, May 1993.