Optimizing Packet Accesses for a Domain Specific Language on Network Processors

Tao Liu[1,2], Xiao-Feng Li[3], Lixia Liu[3], Chengyong Wu[1], Roy Ju[4]

[1]Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
{liutao, cwu}@ict.ac.cn
[2] Graduate School of Chinese Academy of Sciences, Beijing, China
[3] Intel China Research Center Ltd., Beijing, China
{xiao.feng.li, lixia.liu}@intel.com
[4] Microprocessor Technology Labs, Intel Corporation, Santa Clara, CA, USA

Abstract. Programming network processors remains a challenging task since their birth until recently when high-level programming environments for them are emerging. By employing domain specific languages for packet processing, the new environments try to hide hardware details from the programmers and enhance both the programmability of the systems and the portability of the applications. A frequent issue for the new environments to be widely adopted is their relatively low achievable performance compared to low-level, hand-tuned programming. In this paper we present two techniques, Packet Access Combining (PAC) and Compiler-Generated Packet Caching (CGPC), to optimize packet accesses, which are shown as the performance bottleneck in such new environments for packet processing applications. PAC merges multiple packet accesses into a single wider access; CGPC implements an automatic packet data caching mechanism without a hardware cache. Both techniques focus on reducing long memory latency and expensive memory traffic, and they also reduce instruction counts significantly. We have implemented the proposed techniques in a high level programming environment for network processor named Shangri-La. Our evaluation with standard NPF benchmarks shows that for the evaluated applications the two techniques can reduce the memory traffic by 90% and improve the packet throughput by 5.8 times, on average.

1 Introduction

Network processors (NPs) have been proposed as a key building block of modern network processing systems. To meet the challenging performance and programmability requirements of network applications, network processors typically incorporate some unconventional, irregular architectural features, e.g. multiple heterogeneous processing cores with hardware multithreading, exposed multi-level memory hierarchy, and banked register files, etc. [9, 11]. Effective utilization of these features is critical to the performance of NP-based systems. However, the state-of-the-art of programming with NPs is still at a low level, often assembly language, which requires extensive knowledge of both the applications and the architectural details of the target system. A low-level programming task is tedious, time-consuming, and error-prone. It is difficult to port an application across different network processors even within the same family. A high-level programming environment is hence desirable to facilitate the packet processing application development on NPs. The key to the success of such a programming environment is not only its ease of programming, but also its ability to deliver high performance.

Packet processing systems typically store packets in a packet buffer in DRAM, which usually has a large capacity but a long access latency compared to other memory levels. Since there are a large number of packet accesses in network applications, DRAM bandwidth needs to be high enough to sustain maximal packet processing throughput. Although the DRAM access latency can be partially hidden using multithreading, the bandwidth problem remains critical. Actually, DRAM bandwidth has been considered as the bottleneck of network application performance in some prior studies [1, 8, 12]. Our approach is to optimize the packet accesses automatically in a compiler, which reduces both the packet access count and the aggregate access size, so that the total access time and bandwidth requirement are effectively reduced.

In this paper, we present two techniques used for packet access optimizations. The first one is *Packet Access Combining (PAC)*, which reduces the number of packet accesses by merging several access requests into one; and the second technique is *Compiler-Generated Packet Caching (CGPC)*, which implements an automatic packet data caching mechanism to minimize the number of accesses to the packet buffer in DRAM as well as reduce the instruction count.

We implemented the proposed optimizations in Shangri-la [3], which is a programming environment for network processors, and targets the Intel IXP family [11]. Shangri-La encompasses a domain-specific programming language designed for packet processing named *Baker* [2], a compiler that automatically restructures and optimizes the applications written in Baker, and a runtime system that performs resource management and dynamic adaptation at runtime. The compiler consists of three components: a profiler, a pipeline compiler, and an aggregate compiler. The profiler extracts runtime characteristics by simulating the application with test packet traces. The pipeline compiler is responsible for pipeline construction (partition application into a sequence of staged aggregates, where an aggregate includes the code running on one processing element) and data structure mapping. The aggregate compiler takes aggregate definitions and memory mappings from the pipeline compiler and generates optimized code for each of the target processing cores. It also performs machine dependent and independent optimizations, as well as domain-specific transformations to maximize the throughput of the aggregates. The work presented here is implemented in the pipeline compiler and the aggregate compiler.

Our experiments are performed on Intel IXP2400, which contains eight Microengines (MEs) for data plane processing and one XScale core for control plane processing. IXP2400 has four types of memory levels: local memory, scratchpad, SRAM and DRAM. Experimental results show that our approach can reduce the memory traffic by 90% and improve the throughput by a factor of 5.8X, on average.

The rest of the paper is organized as follows. Section 2 introduces the related features of the Baker language. Section 3 and Section 4 describe Packet Access Combining and Compiler-Generated Packet Caching, respectively. Section 5 presents the experimental results. Section 6 reviews related work. Section 7 concludes the paper.

2 Baker Language and Packet Access Characteristics

Baker is a domain-specific programming language for packet processing on highly concurrent hardware. It presents a data-flow programming model and hides the architecture details of the target processors. Baker provides domain-specific constructs, such as *Packet Processing Functions (PPFs)* and *Communication Channels (CCs)*, to ease the design and implementation of packet processing applications, as well as enable effective and efficient compile-time parallelization and optimizations.

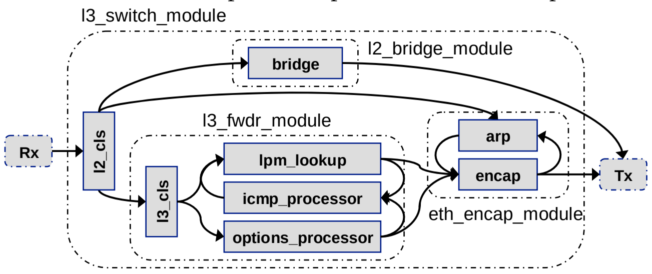

Fig. 1. The packet flow graph of Layer 3 Switch Baker program (L3-Switch): bridges Ethernet packets and switches IPv4 packets

Baker programs are organized as data flow graphs (referred to as *packet flow graphs*) with the nodes representing *Packet Processing Functions* and the arcs representing *Communication Channels*, as shown in **Fig. 1**. A PPF can have its private data, functions and channel endpoints, and performs the actual packet processing. CCs are logically asynchronous and unidirectional queues, and can be created by wiring the input and output endpoints of PPFs. Baker also provides *module* as a way to encapsulate PPFs, shared data and configuration functions. *Rx* and *Tx* are native modules provided by system vendors which can be used as a device driver to receive and transmit packets with external interfaces, respectively.

```
protocol ether {
  dst : 48;
  src : 48;
  type : 16;
  demux{ 14 };
};
```

```
protocol ipv4 {
  ver : 4;
  length : 4;
  ...
  ttl: 8;
  prot: 8;
  checksum: 16;
  ...
  demux{length << 2};
};
```

```
void A.process(ether_packet_t* pkt){
  ipv4_packet_t* p;
  mac_addr_t mac;
  mac = pkt->dst;
  ...
  if(fwd){
    p = packet_decap(pkt);
    channel_put(l3_fwdr_chnl,p);
  }
}
```

Fig. 2. Protocol construct and packet primitives in Baker

The format of the packet header of any protocol can be specified using the *protocol* construct, as illustrated in **Fig. 2**. These definitions introduce new types called *ether_packet_t* and *ipv4_packet_t*, which are processed as built-in types to support operations on Ethernet and IPv4 packet headers, respectively. To access the packet fields of a particular protocol header, programmers must specify a pointer to packet and the field name of corresponding protocol construct. The pointers to packets are referred as *packet handles*. As illustrated in **Fig. 2**, *pkt* is a packet handle to *ether_packet_t*, thus *pkt->dst* represents the *dst* field of Ethernet header. We called the reference to a packet field as a *packet access*.

Baker provides an encapsulation mechanism to layer different packet protocols. The *packet_encap/packet_decap* primitive is to add or remove a protocol header to or from the current packet. As illustrated in **Fig. 2**, *p = packet_decap(pkt)* will remove the Ethernet header from the *pkt* packet so as to convert it to an IPv4 packet.

Besides packet accesses and packet encapsulations, Baker also provides other primitives to ease the manipulations of packets. For example, *channel_get* and *channel_put* are for receiving and transmitting packets through a channel, respectively.

These primitives constitute a packet abstraction model which provides a very convenient way for programmers to write network applications without concerning the underlying implementations. To keep the portability, all packet primitives are implemented as intrinsic functions in the runtime system. The Baker primitives implemented in the runtime system are briefly described below.

The *packet handle* actually points to *metadata* in SRAM, which is data that is associated with a packet but does not come directly from an external source. The metadata is useful to store the packet-associated information generated by one PPF and pass it to another PPF to be processed. For example, the output port is likely part of metadata. The pointers (*head pointer* and *tail pointer*) in the metadata point to the actual packet data in DRAM, as illustrated in **Fig. 3**.

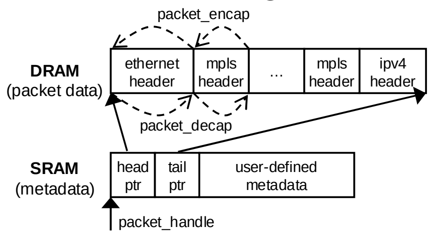

Fig. 3. The layout of packet data and metadata

Packet encapsulations are implemented as intrinsic calls: *packet_encap/packet_decap(packet_handle, size)*. The *size* is the number of bytes to add to or remove from the head. As the example in **Fig. 2**, *p = packet_decap(pkt)* will be converted to *packet_decap(pkt,14)*. The *14* is the length of Ethernet header, which can be determined by the *demux* field in the protocol construct. The implementation of this intrinsic simply increases the *head pointer* in the metadata by 14 bytes.

Packet accesses (packet reads and writes) are implemented as intrinsic calls: *packet_read/packet_write(packet_handle,offset,size,data)*. For example, *data=p->ttl* and *p->ttl=data* can be converted to *packet_read(p,64,8,data)* and *packet_write (p,64,8,data)*, respectively. The size of *8* means that this packet access will retrieve or modify a bit field which is 8-bit wide, and the offset of *64* specifies that the distance to the beginning of the current protocol header is 64 bits. The fourth parameter, *data*, is the input or output data to be read from or written to as specified by programmers. The two intrinsic calls, referred to as *packet access intrinsics* will access DRAM to retrieve or modify packet data. They resolve the DRAM address by the value of *head pointer* plus the offset parameter.

In the Intel IXP2xxx network processors, DRAM can only be accessed in multiples of 8 bytes starting on any 8-byte boundary. Although *packet_read* and *packet_write* intrinsics can specify arbitrary offset and size, the runtime system must take care of address alignment and access granularity. For example, write accesses smaller than 8-byte cause read-modify-write operations to merge data, and the runtime system will generate a mask to select which bytes to be written into DRAM. In a read-modify-write operation it will cause two DRAM accesses.

3 Packet Access Combining

In general, a *packet_read* intrinsic has one DRAM access (for packet data) and one SRAM access (for packet metadata) and dozens of other instructions. A *packet_write* doubles the cost. In a Baker program, each of the packet accesses may operate on only a few bits of the packet header. However, since each DRAM access operates at an 8-byte granularity, a naive code generation that translates a packet access into an intrinsic call can cause a significant waste of DRAM bandwidth and incur unnecessary execution time due to the long DRAM access latency.

The idea of PAC optimization is based on the observation that many packet reads (writes) access contiguous locations. It is possible for the compiler to automatically merge several packet reads (writes) into one, so that only one *packet_read* (*packet_write*) intrinsic is issued to load (store) all of the needed data at once. Thus the DRAM access count can be reduced.

PAC optimization should not change the semantics of the original program, so the application of PAC must comply with control and data dependence requirements. When combining two packet reads, there are two requirements that must be satisfied:

1. *Dominance*: The first read must dominate the second read in flow graph;
2. There are no intervening packet writes along the path from the first read to the second read altering the packet data that the second read will use.

Correspondingly, the requirements of combing two packet writes are:

1. *Control Equivalence*: The first packet write dominates the second and the second post-dominates the first.
2. There are no intervening packet reads (writes) along the path from the first write to the second write using (altering) the packet data of the second write.

The conditions for combining more than two packet reads (writes) can be derived from the requirements above since the compiler can always merge the first two packet reads (writes) into one and then merge this new one with the third read (write). The compiler can follow this process iteratively till all of the reads (writes) that satisfy the conditions are combined.

Fig. 4 gives an example of PAC optimization. **Fig. 4.a** is the flow graph before a PAC optimization. The packet accesses are represented as packet access intrinsic calls. There are two packet reads and two packet writes accessing nearby but different fields of IPv4 header. PAC wants to merge the two reads into a single read, and the two writes into a single write. The flow graph after combining is shown in **Fig. 4.b**. The benefit of PAC is clear: two packet access intrinsic calls were removed. To formalize the solution of the combining problem, we develop a bit-field dataflow analysis on these packet accesses.

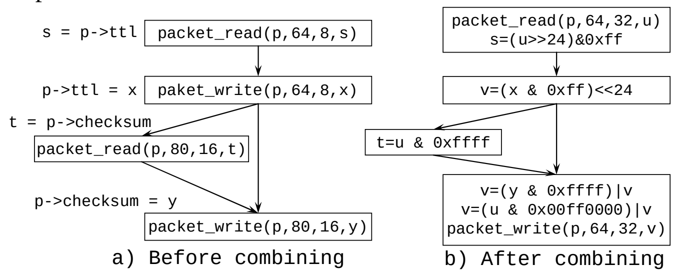

Fig. 4. An example of PAC

3.1 Algorithm

According to the requirements described above, only those packet accesses that satisfy the following conditions can be combined: First, all accesses must be of the same type (read or write), and operate on the same packet. Second, the offsets and sizes of all accesses must be known at compile-time. Third, the size of the combined access must be within the burst size of a single DRAM access. Last, there shouldn't be any violation of control and data dependence due to combining these accesses.

Packet access combining can be performed in the following four steps:

1. *Collect the candidate packet access information*

We first traverse a program function to collect the necessary information for each packet access, including the packet handle, offset and size. This information will be used in the succeeding steps.

2. *Compute the dominance relations*

As discussed above, these packet accesses to be combined must satisfy the dominance relationship (control dependence). Because one basic block (BB) can only have one branch or call instruction, these packet access calls must be in different BBs. Hence, the dominance relationship of packet accesses can be represented as dominate tree of BBs.

3. *Perform a packet field live analysis*

We perform a data-flow analysis on packet fields of packet accesses. In the analysis, a packet read can be considered as a use to a bit-field of packet buffer, and a packet write can be considered as a definition. To uniquely identify each packet access and describe the bits information of them, a triplet *{bb,ph,pf}* was introduced to represent *packet access info* during the iterative dataflow analysis. The *bb* depicts the basic block that the packet access resides in. The *ph* (packet handle) indicates which packet instance it will access. The *pf* (packet field) is a bit vector each bit of which represents a bit in the packet buffer. The corresponding bits that the packet access will read or modify are set to valid while other bits are set to invalid. If the *packet access info* is propagated across a *packet_encap* or *packet_decap* call, its *pf* must shift corresponding bits because the current *head pointer* has been changed. The dataflow analysis of packet reads is a backward dataflow problem. Its corresponding flow equations are specified as **Fig. 5**. $\mathtt{PF}_{\mathtt{rev\_in}}(\mathtt{BB}_{\mathtt{i}})$ and $\mathtt{PF}_{\mathtt{rev\_out}}(\mathtt{BB}_{\mathtt{i}})$ are the sets of reversed input and output packet accesses information of $\mathtt{BB}_i$, respectively. After the bitwise dataflow analysis, $\mathtt{PF}_{\mathtt{rev\_in}}$ of each BB contains all possible packet accesses which can be propagated to the exit of this BB. We said a packet access *s* is *live* at $\mathtt{BB}_i$ if $s \in \mathtt{PF}_{\mathtt{rev\_in}}(\mathtt{BB}_i)$ and the valid bits in `s.pf` has not been changed with respect to its original BB (`s.bb`). A packet access live at a given program point indicates that it can be combined with another packet access resided at this point without violating any data dependence.

$$PF_{rev\_in}(BB_i) = \bigcup_{BB_j \in Succ(BB_i)} PF_{rev\_out}(BB_j) \qquad Gen(BB_i) = \begin{cases} \{(i, s.ph, s.pf)\} & \text{if } BB_i \text{ has packet access "}s\text{"} \\ \phi & \text{otherwise} \end{cases}$$

$$PF_{rev\_out}(BB_i) = Cap(BB_i, Kill(BB_i, PF_{rev\_in}(BB_i) \cup Gen(BB_i)))$$

$$Kill(BB_i, set) = \begin{cases} \{(x.bb, x.ph, (\sim s.pf)\,\&\,x.pf) \mid x.ph = s.ph, \forall x \in set\} \cup \{x \mid x.ph \neq s.ph, \forall x \in set\} \\ \qquad \text{if } BB_i \text{ has packet write "}s\text{"} \\ set \qquad \text{otherwise} \end{cases}$$

$$Cap(BB_i, set) = \begin{cases} \{(x.bb, x.ph, x.pf >> bits) \mid \forall x \in set\} & \text{if } BB_i \text{ has } packet\_encap(bits) \\ \{(x.bb, x.ph, x.pf << bits) \mid \forall x \in set\} & \text{if } BB_i \text{ has } packet\_decap(bits) \\ set & \text{otherwise} \end{cases}$$

Fig. 5. Data-flow equations of packet field live analysis for packet reads

4. Finalize the combining

For each packet access, the candidates can be selected by taking a bitwise OR operation on the current packet access's *pf* field and those of all live packet accesses at this point. If the bit width of combined result does not exceed the width limit of DRAM instructions, the corresponding live packet access is a candidate. We use the *combining density* to describe data reuse characteristics as defined in Eq. (**1**). In this equation *field_len1* and *field_len2* are the valid bit widths of the *pf* fields in the current packet access and candidate packet access, respectively. *combined_len* is the valid bit width after the combination. For example, if the two packet accesses are to the same packet field, the value of combining density equals the width of the packet field. If the packet fields are adjacent, the value is zero, and so on. We will first combine the packet accesses whose combining density is higher.

$$\text{CombiningDensity=field\_len1 + field\_len2 - combined\_len}\,. \qquad (\mathbf{1})$$

After the combination, the offset and size of current packet access are adjusted to retrieve all needed packet data and the redundant packet access is eliminated. The cached packet data can be kept in registers.

The algorithm of PAC can be easily extended to handle more complex cases. For example, it can combine two packet writes even if they are to non-adjacent fields of a packet. By using a dominator packet read to cache the data of the gap between two packet writes, we can combine the two packet writes with the cached gap into a wide write. Furthermore, it can combine packet writes located in basic blocks that are not control equivalent. It may still be worth combining if we can reduce the number of packet writes on the critical path. To maintain correctness, compensation packet writes must be generated in the corresponding exits to cold paths.

4 Compiler-Generated Packet Caching

By default, for each packet access our compiler will generate a packet access intrinsic call which is implemented in the runtime system. This approach, though allows the flexibility of changing the implementation of the packet buffer without modifying the compiler, will incur significant performance overhead. In fact, we may not need to invoke the intrinsic call to load the packet data for every reference in the program. If we preload all needed packet data into a cache, the subsequent packet accesses can be replaced by cache accesses. Actually, packet data accesses exhibit good spatial locality w.r.t. different fields in the same packet [15]. Based on this observation, we propose a new approach to implement packet accesses, named *Compiler-Generated Packet Caching (CGPC)*. CGPC tries to identify the critical path of the packet flow in a network application based on profiling information and optimize all packet accesses along the path. If there are multiple accesses to the same packet in the critical path, the related packet data will be buffered in the fastest level of memory (e.g., the local memory in IXP2400), and those accesses that can be resolved statically will be replaced by the accesses to the buffered data. For those accesses that can only be resolved at run time, efficient code sequence will be generated to calculate the offset and alignment and perform the access. Actually, CGPC can be considered as an extreme situation of PAC that it tries to combine all the packet accesses in a thread into only one packet read at the thread entrance and one packet write at the thread exit.

4.1 Algorithm

CGPC is performed in two steps. First, an inter-procedure analysis, referred to as *Packet Flow Analysis*, is to identify the critical path in the packet flow graph and calculate associated information of each packet access and *packet_encap/decap*. Second, a compiler generates the instructions for each packet access and *packet_encap/decap* based on the packet flow analysis information.

4.1.1 Packet Flow Analysis

The information needed by the packet flow analysis is collected by a profiler. By utilizing user-supplied packet traces, the profiler simulates the execution of network applications at a high-level Intermediate Representation (IR) in the compiler. After the simulation, the profiling information, such as execution frequency and access statistics, is available. The pseudo code of the algorithm for the packet flow analysis is presented in **Fig. 6**. `Flow_Anaysis` is a recursive function which starts the analysis from the endpoint of the channel coming out of the *Rx* module. The cached packet data should be preloaded at the entry of the packet flow, but the preload width can not be determined until the analysis is finished. During the analysis, the value of the current *head pointer* is tracked and updated whenever encountering a *packet_encap/decap*. However, different intrinsic calls and control structures complicate this process. If a *packet_encap/decap* sits inside a loop with an unknown loop count, inside an if-branch, or inside a circle of the packet flow graph, we may not be able to track a constant value of *head pointer* statically.

```
Process_Instrinsic_Call(currCall){
  if(currCall is packet_encap/decap){
    if(is_in_loop){
      set unresolved flag;
      set currCall dynamic;}
    else{
      Increase/Decrease currOfst;
      set currCall eliminable;}
  }
  if(currCall is packet_read/write){
    if(access offset is variable||unresolved)
      set currCall dynamic;
    else{
      set currCall static;
      calculate absolute offset and size;}
    update preload & writeback range;
  }
  if(currCall is channel_put){
    if(send packet to Tx or Xcale){
      set packet_is_over;
      if(cache has been written) writeback cache;
    }
    if(send packet to ME){
      if(cache has been written) writeback cache;
      callee=Get_End_Func(currChannel);
      Flow_Analysis(callee->first_Stmt);
    }
  }
  if(currCall is packet_drop)
    set packet_is_over;
  …… // other cases
}
```

```
Flow_Analysis(currStmt){
  switch(currStmt){
    case Intrinsic_Call:
     {Process_Intrinsic_Call(Intrinsic_Call);
      break;}
    case Call:
     {callee=Get_Callee(Call);
      if(callee has been analysed) break;
      else{
        Flow_Analysis(callee->first_Stmt);}
      break;}
    case Loop:
     {set is_in_loop flag;
      estimate loop count by profiler;
      Flow_Analysis(Loop body);
      if(not in outer loop) reset is_in_loop flag;
      break;}
    case If:
      {Flow_Analysis(if condition);
       Flow_Analysis(then branch);
       then_ofst=currOfst;
       Flow_Analysis(else branch);
       else_ofst=currOfst;
       if(then_ofst==else_ofst) break;
       if(packet_is_over in then/else branch)
         set currOfst to else_ofst/then_ofst;
       else
         set unresolved flag; break;}
    …… // other cases
    default:
      {Flow_Analysis(kid nodes of currStmt);}
}}
```

Fig. 6. The algorithm of packet flow analysis

For each packet access, if the *head pointer* is not resolved as a compile-time constant or its offset parameter is a variable, it will be marked as *dynamic*. They need a compiler to generate code to compute the offset and alignment at runtime so as to access the cached data. Other packet accesses will be marked as *static* and will have their offsets and alignments calculated at compile-time. Since the offsets of static packet accesses are known at compile-time, we can use the absolute offset in the cache to access packet data across different protocol layers. As a result, some *packet_encap/decaps* become redundant if they are used only to provide the encapsulation protection for static packet accesses. These *packet_encap/decaps* are marked as *eliminable*, which means they can be removed safely. Other *packet_encap/decaps* are marked as *dynamic* which will be used in generating code for dynamic packet accesses. When packets flow to the *Tx* module or heterogeneous cores (e.g., XScale), the packet flow path is ended and the cached packet data should be written back to DRAM if it has been modified. If we use a processor-local memory (e.g., local memory in ME) as a cache and packets flow across different cores (e.g., MEs), the cached data should be written back to DRAM when it comes out of one processing core and reloaded when it enters another core.

Fig. 7 illustrates the critical packet flow path of L3-Switch. The *head pointer* can always be determined statically along this path. All packet accesses are resolved except one in the *lpm_lookup* PPF, which is used to verify the checksum of IPv4 header. Its offset is a variable and this access is executed ten times for every processed packet. We need to insert code to compute its offset at runtime.

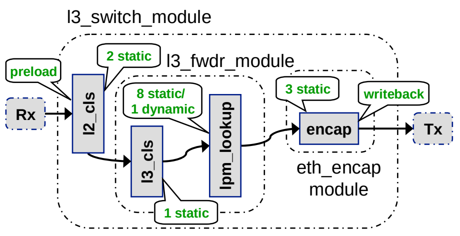

Fig. 7. The critical path of L3-Switch

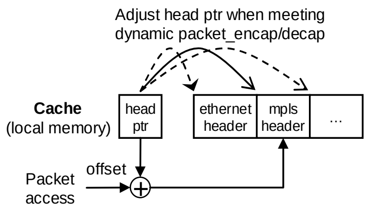

Fig. 8. Dynamic offset and alignment resolution

4.1.2 Compiler-Generated Packet Accesses

After the packet flow analysis, the flags (as shown in **Fig. 6**) and necessary information are annotated on each packet access and *packet_encap/decap*. In the code generator, the actual code is generated according to the flags and the information. If the packet access is static, the cache can be accessed directly with a constant offset and size provided by the packet flow analysis. An unaligned access can be effectively optimized to a wide access followed by some shift instructions. As for a dynamic access, the offset and alignment must be calculated at runtime. Our solution is illustrated in **Fig. 8**. We use a variable to track the value of *head pointer* and initialize it when the compiler preloads the cache. When a packet flows across a dynamic *packet_encap/decap*, additional instructions are executed to update its value at runtime. We can then use the variable of *head pointer* to generate code for the dynamic packet access. The absolute offset of a dynamic packet access in cache can be determined by adding the original offset to the current *head pointer*. A check is performed on the absolute offset. If the offset within the cache, it can directly access the cached data. Otherwise, it will fall through to invoke the original intrinsic call.

After the optimization, a DRAM access is performed only when preloading and writing back the cache. An unaligned DRAM access will cause a much higher cost than the aligned one. For example, an unaligned write would need a write-after-read operation to keep the unwritten section intact, which needs to be implemented in two DRAM accesses. Instead, our compiler implements all preload and write back operations at the aligned boundaries. All intermediate packet accesses' offsets are adjusted according to the alignment. As a result, our implementation properly aligns all DRAM accesses. Although this approach may waste some cache space to hold unused data, it avoids the write-after-read operations on DRAM and reduces the alignment instructions.

5 Evaluations

We have evaluated the proposed optimizations with representative workloads on real network processors. In this section, we will present the hardware evaluation environment, benchmarks, and experimental results.

5.1 Benchmark Applications

We use three typical network applications, L3-Switch, MPLS and Firewall, for our evaluation. They are all written in Baker. L3-Switch and MPLS were evaluated using the NPF standard configurations [16, 17]. Firewall was evaluated using a packet trace internally developed.

Layer 3 Switch (L3-Switch) [16] implements Ethernet bridging and IPv4 routing. For each packet received, it performs table lookups to determine the next hop, decrements the Time-To-Live (TTL), and updates the checksum for the packet header.

Multi-Protocol Label Switch (MPLS) [17] attaches one or more labels in the head of each packet and routes the packet based on the label rather than the destination address. By using the label as an index into a forwarding table, the routing process can be accomplished more quickly.

Firewall sits between a private network and its Internet connection, protecting the internal network against attacks. The firewall takes actions, such as passing or dropping a packet, based on an ordered list of user-defined rules. These rules specify the actions to take when the fields of incoming packets (e.g. source and destination IPs, source and destination ports, protocol etc.) match certain patterns.

5.2 Experimental Environment

Our evaluations were conducted on a RadiSys ENP-2611 evaluation board, which contains an Intel IXP2400 network processor running MontaVista Linux on the XScale core. IXP2400 consists of eight multi-threaded MicroEngines (MEs) for traffic processing, an Intel XScale core for control plane processing, 8MB SRAM, and 64MB DRAM [10]. An IXIA packet generator with two 1Gbps optical ports was used to generate packet traffics and collect statistics. When the ports are used in full duplex mode, the peak input rate is 2Gbps.

Table 1. The parameters of different levels of memories in IXP 2400 (Unit B stands for Bytes)

Memory Type	Size	Access time (Cycles)	Start Address Alignment	Min Length	Max Length
Local Memory	2560B	3	4B boundary	4B	4B
Scratchpad	16KB	60	4B boundary	4B	64B
SRAM	256MB	90	4B boundary	4B	64B
DRAM	2GB	120	8B boundary	8B	128B

The memory hierarchy of IXP2400 consists of four different memory levels: local memory, Scratchpad, SRAM, and DRAM, with increasing capacities and access latencies. **Table 1** lists their access parameters. There is no hardware cache; any access to the memory units is carried out explicitly with specific instructions for respective memory types.

For all configurations in our evaluation, six MEs with each ME having eight thread contexts all ran the same code from the critical path of an application. The other two

MEs were dedicated to receive (Rx) and transmit (Tx) module, respectively. The cold path and control plane code of the application were mapped to XScale.

5.3 Packet Access Count and Aggregate Access Size

We compared the number of packet-related DRAM accesses and the packet forwarding rates for the three applications, with and without the proposed optimizations. The BASE configuration enables only typical scalar optimizations. We evaluated these two optimizations on top of BASE separately. PAC enables the packet access combining. Procedure inlining was performed to expose more opportunities for combining. CGPC represents the compiler-generated packet caching. Since CGPC can be considered as an aggressive version of PAC, we have not evaluated the combined effect.

Table 2. Memory access statistics (per packet) and instruction counts

		DRAM Access Count	**Aggregate Access Size (Bytes)**	**Instruction Count**[1]
L3-Switch	**BASE**	29	696	2033
	PAC	13	200	1190
	CGPC	2	72	770
MPLS	**BASE**	16	384	1851
	PAC	9	212	1428
	CGPC	2	48	1495
Firewall	**BASE**	24.2	580	1742
	PAC	4.4	140	572
	CGPC	1	32	375

Table 2 shows the DRAM access count and aggregate access size per packet and the instruction count for each benchmark application. We can see that PAC can reduce the DRAM access dramatically. CGPC has the lowest number of DRAM accesses and reduces the aggregate access size by 90% on average (L3-Switch: 89.7%, MPLS: 87.5%, Firewall: 94.5%). Taking L3-Switch as an example, its packet accesses are marked in **Fig. 7**. There are 9 static packet reads, 5 static unaligned packet writes, and 1 dynamic packet read on the critical path. The dynamic packet read is caused by a checksum checking, which iterates through the packet header in a unit of 2-byte. PAC merges the static packet accesses but cannot catch the dynamic one. CGPC can deal with all of the packet accesses, thus only need DRAM accesses in the preload and write back operations. MPLS presents a challenge to our techniques initially. It pushes, swaps, and pops MPLS labels dynamically, which may include an arbitrary number of MPLS headers and our techniques can not determine the cache layout statically. However, the results demonstrate that CGPC remain effective for

[1] The instruction count is an approximate number of the instructions actually executed in MicroEngines for one packet processing. It includes critical path code and packet accesses. A packet read takes about 50 instructions and a packet write takes about 100 instructions.

this dynamic situation. Overall, PAC and CGPC not only reduce the memory traffic, but also reduce the number of executed instructions.

5.4 Forwarding Rate

The forwarding rates of three applications on the minimum sized 64-byte packets are presented in **Fig. 9**. The numbers of MEs to execute the applications are plotted on the X-axis and the achieved forwarding rates are plotted on the Y-axis. To obtain the full benefits of PAC, we unrolled the checksum checking loop in L3-Switch before applying PAC to convert the dynamic packet read to static. PAC reduces the packet processing time by removing considerable DRAM accesses and instructions. As a result, it gets a higher forwarding rate. CGPC provides a higher performance impact than PAC because it has no excessive DRAM accesses and the solution for resolving the offset and alignment is effective. Compared to BASE, CGPC improves the throughput by 5.8 times on average (L3-Switch: 7.6; MPLS: 3.9; Firewall: 5.9).

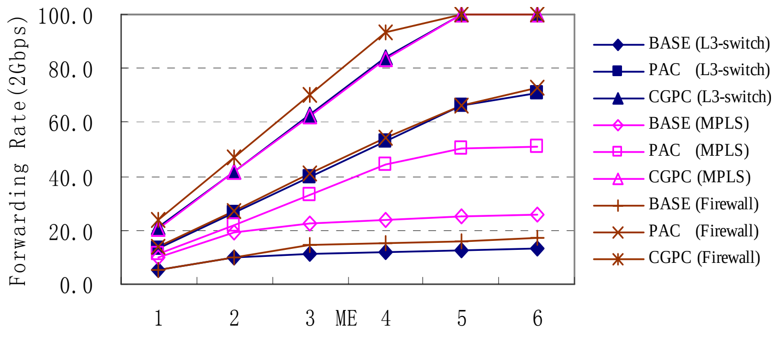

Fig. 9. Performance of L3-Switch, MPLS and Firewall

In the BASE configuration, all three applications get their memory bus saturated when the number of MEs increases. However, PAC provides good scalability by relieving the contention of DRAM bandwidth. Compared to PAC, CGPC generates fewer instructions and DRAM accesses so that it obtains nearly perfect scalability and reaches the full line rate quickly. The result shows the system performance is largely determined by both the instruction count and DRAM bandwidth. We also applied these optimizations on SRAM accesses without as apparent benefits as DRAM accesses. It is because IXP2400 has only one DRAM controller but two independent SRAM controllers.

6 Related Work

Several high-level programming languages, such as microC [11] and picocode [9], have been introduced with their corresponding NPs. But they are all extended to expose hardware details and their performances heavily rely on the use of such features. A number of domain-specific languages, such as Click [13], NesC [7], etc., have been developed to ease programming, and they are more hardware-independent and include special constructs to express the tasks in packet processing applications. But they do not focus on efficient compilation.

Mudigonda et al. [15] analysed the characteristics of packet data and application data accesses. They exhibit the spatial locality of packet data accesses and temporal locality of application data accesses. They use a cache to improve the hit rate of application data structures. Iyer et al. [12] studied a cache-based memory hierarchy of packet buffer. Hasan et al. [8] proposed several techniques to exploit row locality (i.e. successive accesses falling within the same DRAM rows) of DRAM accesses. But their techniques needed hardware support and focused on the input- and output-side of packet processing, which can be implemented in our *Rx* and *Tx* modules. Sherwood et al. [18] designed a pipelined memory subsystem to improving the throughput in accessing application data structures.

Davidson and Jinturkar [6] described a memory coalescing algorithm for general purpose processors similar to packet access combining. This algorithm replaced narrow array access with double-word accesses in unrolled loops. It performed a profitability and safety analysis on programs, and generated alignment and alias checks at runtime if necessary. But Packet Access Combining works on a whole procedure and focuses on packet accesses. It utilizes some domain knowledge and does not need a complex alias analysis. Thus, PAC is always profitable when it can be applied.

There are several techniques which can be used to improve packet accesses. McKee et al. [14] designed a separate stream buffer to improve the performance of stream accesses. Chen et al. [4] described a hardware-based prefetching mechanism to hide memory latency.

7 Conclusion

Performance and flexibility are two major but sometime conflicting requirements to packet-processing systems and the programming environments associated with them. High level programming environments with domain specific languages can satisfy the flexibility requirement. However, how to utilize hardware features effectively to achieve high performance with automatic compiler supports in such programming environments requires more explorations. In this paper, we address one major type of memory accesses in network applications – accesses to packet data structures, which constitute a significant portion of the total memory accesses. We propose two compilation techniques to reduce the latencies of packet accesses and the contention of DRAM bandwidth.

Packet access combining tries to reduce the number of packet accesses by utilizing wide memory references and code motion. It does not incur extra memory space compared with caching. Furthermore, it is hardware-independent and always beneficial when applied. Compiler-generated packet caching can be viewed as compiler-controlled caching. It buffers the packet data to be referenced and replace all of the packet accesses on the critical path with accesses to a buffer in cache. Through a profiling-based program analysis, it minimizes the required cache size and the number of cache misses.

We performed experiments on a real packet processing platform with three representative network applications, L3-Switch, MPLS and Firewall. The experimental results demonstrate that the efficiency of packet accesses is critical to the system performance, and our techniques can reduce the number of packet accesses and the total memory bandwidth requirements significantly.

Reference

1. W. Bux, W. E. Denzel, T. Engbersen, A. Herkersdorf, and R. P. Luijten. “Technologies and building blocks for fast packet forwarding.” *IEEE Communications Magazine*, pp. 70-77, January 2001.
2. M. Chen, E. Johnson, R. Ju. “Compilation system for throughput-driven multi-core processors.” In *Proc. of Micro-37*, Portland, Oregon, December 2004.
3. M. Chen, X. Li, R. Lian, J. Lin, L. Liu, T. Liu, and R. Ju. “Shangri-la: Achieving high performance from compiled network applications while enabling ease of programming.” In *Proc. of ACM SIGPLAN PLDI*, Chicago, Illinois, USA, June 2005.
4. T. Chen and J. Baer. “Effective Hardware-based Data Prefetching for High-performance Processors.” *IEEE Transactions on Computers*, 44(5), May 1995.
5. T. Chiueh and P. Pradhan. “High-performance IP routing table lookup using CPU caching.” In *IEEE Infocom’99*, New York, NY, March 1999.
6. J. W. Davidson and S. Jinturkar. “Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses.” In *Proc. of ACM SIGPLAN PLDI*, pp. 186-195, June 1994.
7. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. “The nesC Language: A Holistic Approach to Networked Embedded Systems.” In *Proc. of ACM SIGPLAN PLDI*, June 2003.
8. J. Hasan, S. Chandra, and T. Vijaykumar. “Efficient Use of Memory Bandwidth to Improve Network Processor Throughput.” In *ISCA*, 2003.
9. IBM PowerNP Network Processors, http://www-3.ibm.com/chips/techlib/techlib.nsf/products/IBM_PowerNP_NP4GS3.
10. Intel Corporation. *Intel IXP2400 Network Processor: Hardware Reference Manual*. 2002.
11. Intel IXP family of Network processors, http://www.intel.com/design/network/products/npfamily/index.htm.
12. S. Iyer, R. R. Kompella, and N. McKeown. “Analysis of a memory architecture for fast packet buffers.” In *Proc. IEEE Workshop High Performance Switching and Routing (HPSR)*, 2001.
13. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. “The Click Modular Router.” *Transactions on Computer Systems*, 2000.
14. S. McKee, R. Klenke, K. Wright, W. Wulf, M. Salinas, J. Aylor, and A. Batson. “Smarter Memory: Improving Bandwidth for Streamed References.” *IEEE Computer*, July 1998.
15. J. Mudigonda, H. Vin, and R. Yavatkar. “A Case for Data Caching in Network Processors.” Under Review. http://www.cs.utexas.edu/users/vin/pub/pdf/mudigonda04case.pdf
16. Network Processing Forum. “IP Forwarding Application Level Benchmark.” http://www.npforum.org/techinfo/ipforwarding_bm.pdf.
17. Network Processing Forum. “MPLS Forwarding Application Level Benchmark and Annex.” http://www.npforum.org/techinfo/MPLSBenchmark.pdf.
18. T. Sherwood, G. Varghese, and B. Calder. “A Pipelined Memory Architecture for High Throughput Network Processors.” In *30th International Symposium on Computer Architecture*, June 2003.