Shangri-La: Achieving High Performance from Compiled Network Applications while Enabling Ease of Programming

Michael K. Chen[1], Xiao Feng Li[2], Ruiqi Lian[3], Jason H. Lin[2], Lixia Liu[2], Tao Liu[3], Roy Ju[1]

1. Microprocessor Technology Labs
Intel Corporation
Santa Clara, CA, USA
{michael.k.chen, roy.ju}@intel.com

2. Intel China Research Center Ltd.
Beijing, China
{xiao.feng.li, jason.h.lin,
lixia.liu}@intel.com

3. Institute of Computing Technology
China Academy of Sciences
Beijing, China
{lrq, liutao}@ict.ac.cn

Abstract

Programming network processors is challenging. To sustain high line rates, network processors have extremely tight memory access and instruction budgets. Achieving desired performance has traditionally required hand-coded assembly. Researchers have recently proposed high-level programming languages for packet processing, but the challenges of compiling these languages into code that is competitive with hand-tuned assembly remain unanswered.

This paper describes the Shangri-La compiler, which accepts a packet program written in a C-like high-level language and applies scalar and specialized optimizations to generate a highly optimized binary. Hot code paths identified by profiling are mapped across processing elements to maximize processor utilization. Since our compilation target has no hardware caches, software-controlled caches are generated for frequently accessed application data structures. Packet handling optimizations significantly reduce per-packet memory access and instruction counts. Finally, a custom stack model maps stack frames to the fastest levels of the target processor's heterogeneous memory hierarchy.

Binaries generated by the compiler were evaluated on the Intel IXP2400 network processor with eight packet processing cores and eight threads per core. Our results show the importance of both traditional and specialized optimization techniques for achieving the maximum forwarding rates on three network applications, *L3-Switch*, *MPLS* and *Firewall*.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications – *data-flow languages, specialized application languages*; D.3.4 [**Programming Languages**]: Processors – *code generation, compilers, optimization*.

General Terms Algorithms, Performance, Design, Languages.

Keywords Packet processing, network processors, chip multiprocessors, throughput-oriented computing, program partitioning, dataflow programming.

1. Introduction

In spite of the time and effort required, network processors like the Intel IXP [16], IBM PowerNP [15], Broadcom BCM1250 [4] and PMC Sierra RM9000 [29] have been mostly programmed using hand-coded assembly. Networks have mostly relied on the widespread use of a few core packet programs. To achieve high line rates, though, a network program usually has very tight memory access and instruction count budgets. In the past, careful hand-optimization of assembly code has been the most effective means of achieving the required performance given the small kernels and difficult resource constraints.

As networks have evolved, so has the code running them, becoming larger, more diverse and complex. More and more network protocols are being developed for specialized applications (e.g. wireless, VoIP, proxies, network-attached storage). Enhancements to base protocols have been implemented to satisfy load balancing, security and reliability requirements. Shipped network hardware must operate correctly in an increasing number of different configurations.

Hand-coded assembly has become a hindrance to developing new network applications and updating existing applications. Reusing common routines written in assembly in different contexts is difficult, debugging assembly code is extremely tedious and maintaining assembly code is a time-consuming effort. Even state-of-the-art tools that abstract some of the assembly programming details still expose to programmers the multi-threaded packet processing cores, heterogeneous memories and custom communication topologies found on network processors.

Shangri-La, which consists of a programming language, a compiler and a runtime system, simplifies development and accelerates performance tuning of network applications. The Shangri-La compiler accepts a program written in Baker, a high-level, domain specific language for designing modular network applications. Aggressive optimizations are applied so that code written in the high-level language can achieve performance comparable to hand-tuned assembly code. Although the compiler currently targets the Intel IXP multi-core network processor, many of the techniques we describe are generally applicable since they deal with the difficulties of targeting heterogeneous multiprocessors and heterogeneous memories, and of optimizing network application constructs.

The primary contributions of this paper are:

- Demonstrated ability to achieve comparable hand-tuned performance on highly resource-constrained network processors from code compiled from a high-level language.
- A complete framework for aggressively compiling network programs using both traditional and specialized optimizations techniques. Hot code paths identified from profiling are mapped across processing elements to maximize packet forwarding rates. Delayed-update software-controlled caches are automatically generated for unprotected, error-tolerant application data structures, useful for network processors that

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.

have no hardware caches. Packet handling optimizations help reduce per-packet memory access and instruction counts. A custom stack model maps program stack frames to the fastest levels of the target processor's heterogeneous memory hierarchy.

- Detailed performance evaluation of compiler generated code on real IXP hardware of three popular network applications, *L3-Switch*, *MPLS* and *Firewall*. The results show the importance of traditional and specialized optimization techniques in achieving maximal packet forwarding rates.

The remainder of this paper details the major components of the Shangri-La system and provides results from running our compiler-generated code on real hardware. Section 2 introduces the Baker network programming language. Section 3 presents the architecture and performance characteristics of Shangri-La's target hardware, the Intel IXP. Section 4 introduces the Shangri-La compilation framework and the runtime support components. Section 5 details important optimization techniques implemented in the Shangri-La compiler. Section 6 presents the results of running three compiled network applications on real hardware. Section 7 describes related work in compiling for network processors and Section 8 summarizes our findings.

2. Baker Programming Language

Currently, most network processors are programmed by hand using assembly. Assembly programming is undesirable for many reasons. It is not portable between different ISAs. Debugging and performance-tuning assembly code is a tedious and time-consuming effort. Finally, assembly programming requires extremely skilled programmers, since it exposes all the hardware resources. An IXP programmer must deal explicitly with heterogeneous memories, inter-processor communication, multi-processing and multi-threading in addition to the usual complexities.

Even in state-of-the-art tools that allow network processor code to be written in a C dialect [17][18], these aspects of the hardware are configured by the programmer through manual insertion of directives and keywords. For example, keywords indicate physical memory levels for referred variables and intrinsics represent accesses to specialized inter-processor communication hardware. Likewise, data packing and alignment optimizations are contingent upon input from the programmer. These language extensions impose significant responsibilities on the programmer and limit portability of the written programs.

Baker [13] is a platform-independent language for network application development. A programmer writes packet-processing applications to an abstract machine with a single level of memory. Baker looks like C, but includes constructs to enable development of large applications from reusable, modular components and to simplify the expression of network programs. Aside from this basic model, a programmer must only understand that the generated code may run on a multi-core processor and must explicitly identify any critical sections.

2.1 Modular framework

Baker programs are structured as a dataflow of packets from receiver (Rx) to transmitter (Tx), as shown in Figure 1. A *module* is a container for holding related *packet processing functions* (*PPFs*), wirings, support code and shared data. *PPFs* contain the C-like code that performs the actual packet processing. *PPFs* can hold temporary local state and access global data structures. Packets enter and exit *PPFs* through channel endpoints. The input and output channel endpoints of *PPFs* wired together form *communication channels* (*CCs*), shown as directed arrows in the figure. *CCs* are asynchronous and can be thought of as queues or FIFOs. Output endpoints are also immediate-release. This means that when a *PPF* places data on an output, the data is released onto the *CC* and is no longer accessible to the *PPF*. A piece of sample Baker code is shown in Figure 2.

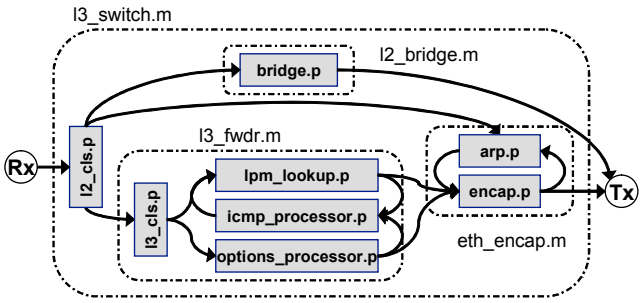

Figure 1 – The L3-Swtich Baker program containing Modules (`.m`), PPFs (`.p`) and CCs (directed arrows). L3-Switch bridges and forwards IP packets.

```
l3_switch.l2_clsfr.ppf( ether_pkt * ph )
{
    int is_arp = ( ph->type == ETH_TYPE_ARP );
    int forward = ( ph->dst ==
          mac_addrs[ph->metadata.rx.port] );
    if( is_arp ){
        channel_put( arp_cc, packet_copy( ph ));
    }
    if( forward ){
        ipv4_pkt iph = packet_decap( ph );
        channel_put( l3_forward_cc, iph );
    }
    else{
        channel_put( l2_bridge_cc, ph );
    }
}
```

Figure 2 – `l2_clsfr` PPF in the `l3_switch` module demonstrating packet (`ph->protocol_field`) and metadata (`ph->metadata.metadata_field`) accesses, use of packet primitives (`packet_decap()` & `packet_copy()`) and placement of packets on CCs (`channel_put()`).

2.2 Packet support

Designing packet and packet protocol support is especially tricky. Protocol developers must be able to add new protocol types easily. There must also be a way to associate state with a packet for use by the network application that is not stored in the packet. Component developers must be insulated from lower-layer protocol encapsulations and packet representations. Finally, manipulations on packets must be computationally efficient.

A packet's data can be thought of as a string of bits. How those bits are interpreted is determined by protocols. A protocol developer creates new protocols by describing them using Baker's `protocol` construct, illustrated in Figure 3. The `demux` pseudo field specifies the `protocol`'s size in a particular packet.

Packet `metadata` can be used to store state associated with a packet, but not contained within a packet. It is particularly useful to a network programmer for storing state associated with a packet generated in one *PPF* and used later by another *PPF*.

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

```
protocol ipv4 {
    ver : 4;
    length : 4;
    ...(fields omitted)
    src : 32;
    dst : 32;
    demux{ length << 2 };
};
```

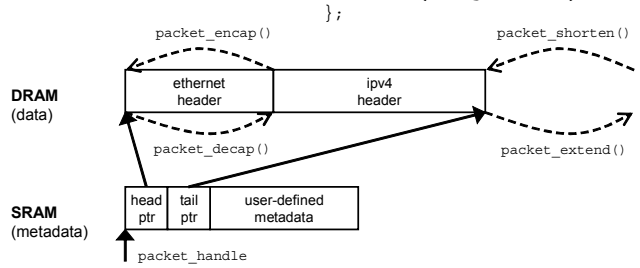

Figure 3 – Example protocols and illustrations describing the effects of packet primitives on a `packet_handle`.

Baker programs manipulate packets through a `packet_handle`. The `packet_handle` points to packet metadata in SRAM. The packet metadata holds user-defined metadata and pointers to the actual packet data in DRAM.

Robust encapsulation primitives make it easy to layer different packet protocols. `packet_decap()`, which internally uses the result of the `demux` field, is used to access the encapsulated payload. Equivalently, one can encapsulate one protocol inside another using `packet_encap()`. `packet_add_tail()` and `packet_remove_tail()` are primitives used to append data to a packet. The effects of these primitives on a `packet_handle` are shown in Figure 3.

There are other features of Baker used to support various packet protocols [13] that are beyond the scope of this paper.

2.3 Language restrictions

Baker imposes several language restrictions to reduce analysis complexity and simplify code generation. Requiring type-alias-free pointers, which prevent explicit typecasts to change the interpretation of a given memory location [11], simplifies alias analysis. Recursion for code within a *PPF* is not supported for two reasons: our survey of network applications indicates recursion is not required; and it would complicate and add overhead to runtime stacks on processors like the Intel IXP with uncached, heterogeneous memories.

3. IXP Network Processor

While the optimizations presented in the paper may be applicable to other network processors, embedded systems or multiprocessors, the Shangri-La compiler currently targets the Intel IXP family of network processors [16]. This section highlights the hardware's many exposed complexities that make it difficult to program or target with a high-level compiler.

3.1 Processor cores

Intel IXP processors are specialized chip-multiprocessors, with one Intel XScale™ core and multiple microengines (MEs) [16]. The XScale processor is used to process control packets, execute non-critical application code, and handle initialization and management of the network processor.

The ME cores are primarily responsible for processing packets. The IXP2400 used for experiments in the paper, shown in Figure 4, has eight MEs. Other IXP processors (e.g. IXP1200 or IXP2800) that have different clock speeds and number of cores can also be targeted by Shangri-La. MEs are lightweight, multi-threaded, pipelined processors running a special instruction set designed for processing packets. Each ME has its own instruction store, independent of other data memory, that holds all the code for threads running on that core. This fast, uncached memory can only hold 4096 40-bit instructions, but it is large enough to hold critical instruction paths for most network applications.

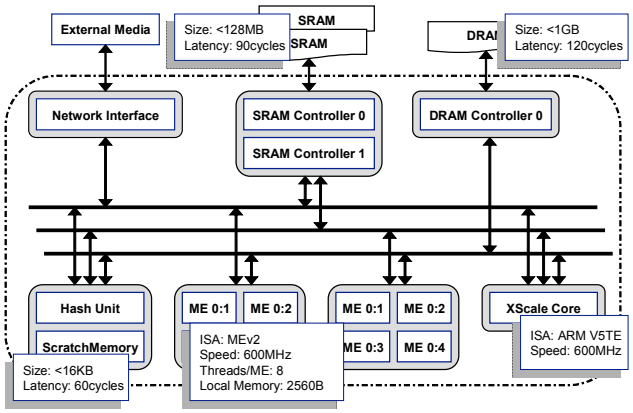

Figure 4 – Block diagram of the IXP2400 showing characteristics of the XScale core, MEs and various memories.

One ME has support for four or eight hardware-assisted threads of execution. Threads on an ME are non-pre-emptive: executing code must give up control explicitly, on a memory instruction or a context switch instruction, before another thread can run. A thread arbiter swaps between ready threads in a round-robin order. This thread model simplifies synchronization within an ME and eliminates additional interrupt-handling hardware, but burdens the programmer with the additional responsibility of handling context switches.

3.2 Memory architecture

Intel IXPs have heterogeneous and uncached memory hierarchies. The four separate levels of memory in the IXP are Local Memory, Scratch Memory, SRAM and DRAM. The sizes and access latencies of the memories are shown in Figure 4. With current IXP programming tools, each different level of memory must be accessed explicitly by the programmer. Memory access instructions include a `ref_cnt` parameter that facilitates wide memory accesses. With the `ref_cnt` parameter, 4B to 32B of Scratch Memory or SRAM, and 8B to 64B of DRAM can be accessed with only one memory instruction. These wide memory accesses are implemented as reads and writes to a contiguous sequence of registers.

Local Memory is treated differently from the other levels of memory. Local Memory is very fast, but is private to a given ME. A subset of Local Memory can be accessed directly from any instruction without a load delay using 8- or 16- word offset addressing. The entire 640 words of Local Memory can be accessed from any thread in an ME with three cycles of latency.

Notably absent in the IXP are any caches for the MEs. In general, network processor architects have used die area and power budgets for multithreading and more processing cores instead of cache memory. This has been motivated by the observation that packets have little or no temporal or spatial memory locality. Additionally, caches are undesirable in embedded systems and in

network applications because they introduce non-deterministic timing effects into a program's behavior. For example, it is difficult to guarantee a minimum line rate for a running network application if it is affected by a cache hit rate.

3.3 Specialized hardware

Although MEs don't contain traditional caches, they do have hardware which can be used to implement a software-controlled cache. Each ME contains a small 16-entry content-addressable memory (CAM). Each entry of the CAM consists of a 32-bit address tag and a 4-bit result. A `cam_lookup` with an address key will return the matching tag's entry number and the 4-bit result on a hit, or the LRU entry number on a miss. An implementation of a software-controlled cache using the CAM will be described in Section 5.2.

The IXP network processor includes many other application-specific features [16] that are beyond the scope of this paper.

4. Shangri-la

Shangri-La is a research project exploring new technologies for improving network processor performance, programmability and usability. Shangri-La consists of the Baker programming language, a compiler, a runtime system and support libraries. Compiler development has focused on innovating and identifying effective optimization techniques, while runtime efforts have explored adaptation techniques to improve performance or reduce power consumption.

4.1 Compiler

The Shangri-La compiler leverages the large code base of the ORC [19][1] project. Extensive changes to the base compiler were made to support the complexities of compiling to the Intel IXP. An overview of the compiler flow is shown in Figure 5.

Right after the source program is converted into nodes of the WHIRL intermediate representation (IR), the *Function Profiler*, which takes a user-supplied packet trace, simulates the network application by interpreting the IR nodes. During simulation, the *Functional profiler* collects global data structure access frequencies, *CC* utilizations and relative *PPF* execution times.

The *Functional profiler* is immediately followed by *Inter-procedural analysis* (*IPA*) *and global optimizer*. *IPA* is primarily responsible for forming *aggregates*, collections of *PPFs* that are mapped to one processing element. The partitioning strategy tries to maximize packet forwarding rates, using *Functional profiler* statistics to identify hot *CC*s to eliminate and frequently executed *PPFs* to duplicate. The aggregation methodology will be described further in Section 5.1. The formed *aggregates* are then dumped into separate WHIRL IR files with each file representing one *aggregate*.

The combined *IPA and global optimizer* is also responsible for managing global memory. While most global application data structures are mapped to SRAM, some data can be placed in Scratch Memory, though, which has about half the latency of SRAM, but is significantly smaller. Possible data structures to be placed in Scratch Memory can be identified with *Functional profiler* access frequency statistics.

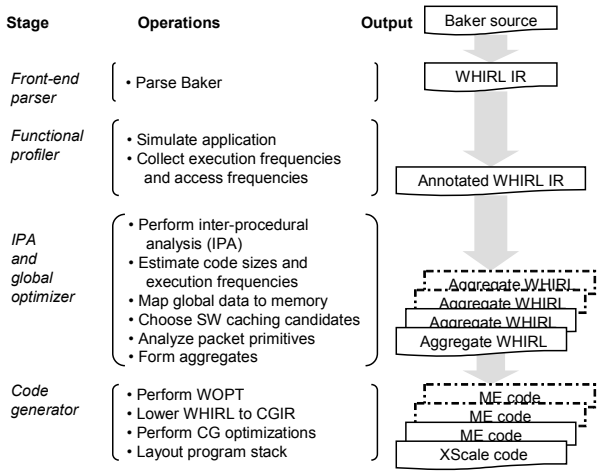

Figure 5 – Overview of Shangri-La compilation stages.

Two different paths in the *Code Generator* target the different IXP cores. Infrequently executed control, management and initialization code is mapped to the XScale core. To generate code for the XScale, the WHIRL nodes are transformed back to C and then compiled by `gcc`. We will not discuss this path further since it is not critical to performance.

Code mapped to the MEs is extensively optimized. First, the compiler applies SSA-based optimizations like dead code elimination, copy propagation and redundancy elimination [7]. Afterwards, the WHIRL nodes are lowered into a code generation intermediate representation (CGIR) adapted for the ME instruction set in which low-level optimizations like global register allocation and instruction scheduling are performed.

Significant effort was spent dealing with tricky aspects of the ME instruction set. For example, the 32 general-purpose registers available to each ME thread are divided into two banks so that instructions with two source operands must have each operand originate from a different register bank. This additional constraint must be considered during global register allocation. Special handling was also required to support wide memory accesses. As mentioned earlier, wide Scratch Memory, SRAM and DRAM reads and writes access a sequence of adjacent 32-bit registers. To properly handle the register dependencies for these memory instructions that read or write multiple registers, the CGIR implements aggregate pseudo-registers which can associate dependencies to individual registers or to the entire set of registers.

4.2 Runtime system

Code generated by the Shangri-La compiler is run on the Intel IXP atop a thin, custom runtime system (RTS). The RTS includes libraries that abstract commonly found network processor hardware resources to facilitate runtime reconfiguration of the system. For example, the RTS can dynamically decide how to map *CCs* to one of many possible hardware-specific implementations (e.g. the IXP has ME next-neighbor registers or Scratch Memory rings for this purpose). The most promising uses of runtime reconfiguration being explored include dynamically turning ME cores off to reduce power consumption when network traffic is low and dynamic code reconfiguration to adapt to changes in processed packet characteristics [36]. This part of Shangri-La is under development and not the focus of this paper.

5. Key Optimizations

At high line rates, network processors have extremely tight instruction and memory access budgets. For OC-48 (2.5Gbps) with minimum size 64B packets, less than 700 instructions can be dedicated to processing a given packet on the IXP2400 (which is designed for the OC-48 configuration) with six MEs (two of the eight MEs are dedicated to Rx and Tx, respectively).

Available memory bandwidth also is a precious resource on IXPs. We performed a simple memory access experiment on our IXP setup to characterize the estimated maximum possible forwarding rates for different numbers of memory accesses per packet, as shown in Figure 6. In this experiment, all six programmable MEs are executing a simple tight loop issuing only memory accesses. The numbers of memory accesses in the tight loop are plotted on the x axis and the achieved forwarding rates are plotted on the y axis. This figure suggests that to achieve 2.5Gbps for minimum sized packets (64B), there can be no more than two DRAM accesses, eight SRAM accesses or 64 Scratch memory accesses for each packet. Also evident in the figure are the fractionally lower forwarding rates that result from issuing wider accesses to a given memory level.

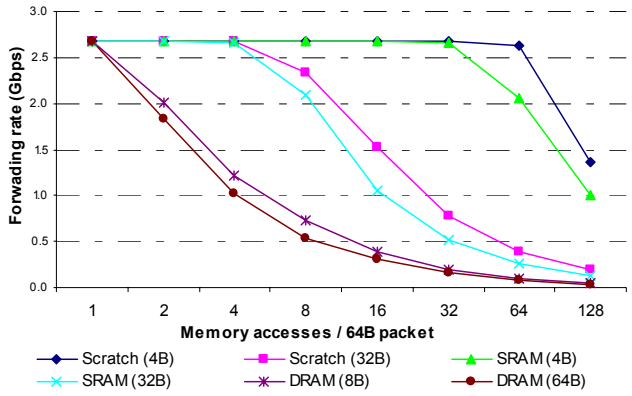

Figure 6 – Illustrates the effects of memory accesses per packet (1-128), memory level (DRAM, SRAM or Scratch) and memory access width (8B-64B) on the forwarding rate of a simple application running on the IXP2400.

While the application of traditional scalar inter-procedural and intra-procedural optimizations are critical to meeting the tight instruction and memory budgets, the optimizations highlighted in this section describe special techniques we applied to meet our performance goals.

5.1 Aggregation

Aggregation attempts to map the application onto available processor resources in order to maximize packet forwarding rates. Aggregation makes use of a throughput model, statistics from the *Functional profiler* and an algorithm for creating *aggregates* from multiple *PPFs*.

$$t \propto \frac{n}{p} \times k \qquad \textbf{Equation 1}$$

The throughput relationship shown in Equation 1 describes packet throughput (t) in terms of the number of MEs (n), the pipeline stage with the lowest throughput (k) and total number of pipeline stages (p). For a given IXP processor, n is fixed, leaving p and k for optimization. k is derived primarily from estimated *PPF* execution times and *CC* overhead statistics collected by the *Functional profiler*. The goal of *aggregate* formation is to maximize packet forwarding rates.

Compared to traditional multi-processor optimization where the goal is to minimize total execution time, this relationship: better reflects how packet pipelines behave; describes the optimization goal according to how performance is evaluated for packet processing; and eliminates system latency from the optimization criteria. The last two points differentiate network applications from normal program optimization in that *latency of a packet through the system can be tolerated in many cases, but minimum forwarding rates must be guaranteed*.

The throughput-driven cost model can result in significantly different code from traditional program parallelization that focuses on latency reduction. When parallelizing a fixed workload, all the system resources are consumed toward a single goal to minimize total execution time. In throughput-oriented systems, the available resources are used to maximize the number of inputs that can be simultaneously processed. Unlike workload optimization, long latency communication and memory accesses are tolerable as long as they can be hidden with work from processing other inputs and memory bandwidth is not saturated.

This cost model is used to drive *aggregate* formation, as shown in Figure 7. To maximize throughput, our compiler can choose to pipeline or duplicate code across multiple processing elements. In pipelining, a packet processing task is divided into *aggregate* pipe stages (each representing one stage in the pipeline) that are mapped to different processing elements connected via *CCs*. During execution, packets are passed between the *aggregate* pipe stages in an assembly-line fashion. Our model correctly indicates that with n fixed, adding pipe stages (increasing p) requires a proportional decrease in k to maintain the same throughput.

```
done ← false
while ! done do
    done ← true
    {dom, next_dom} ← FIND_DOMINATING( aggregates )
    if EXEC_TIME( dom ) >> EXEC_TIME( next_dom ) then
        if DUPLICATE_IMPROVES_THROUGHPUT( dom, target_throughput )
            DUPLICATE( dom )
            done ← false
            continue
    aggregate_pairs ← FORM_PAIRS( aggregates )
    SORT_BY_HIGHEST_CHANNEL_COST( aggregate_pairs )
    foreach pair in aggregate_pairs do
        if MERGE_IMPROVES_THROUGHPUT( pair, target_throughput )
          && MERGE_SATISFIES_CODESIZE_LIMIT( pair ) then
            MERGE( pair )
            done ← false
            break
    if done && NUM( aggregates ) > num_processors then
        RELAX_CONSTRAINT( target_throughput )
        done ← false
foreach aggr in aggregates do
  if( ! SATISFIES_CODESIZE_LIMIT( aggr )
    || INFREQUENTLY_EXECUTED( aggr ){
    MAP_TO_XSCALE( aggr )
    remove aggr from aggregates
  }
duplication_factor ← num_MEs / NUM( aggregates )
MAP_TO_MEs( aggregates, duplication_factor )
```

Figure 7 – Pseudo-code for forming aggregates.

An individual *aggregate* pipe stages or the entire pipeline can also be duplicated to run on multiple processors. Duplicating an *aggregate* pipe stage effectively doubles its throughput. The

duplicated pipe stage can now handle twice as many packets in steady-state flow, even though the latency through it remains unchanged. If the entire packet pipeline is duplicated instead, *floor(n / p)* is the pipeline replication factor.

On real network programs and IXP hardware, the throughput model biases against pipelining and favors duplication for two reasons. Firstly, to maximize throughput of the slowest pipe *aggregate*, work must be evenly partitioned across all pipeline *aggregates*, a challenging task in practice. Secondly, pipelining naturally adds overhead for communicating data over *CCs* between each pipe *aggregate* compared to an equivalent aggregation without pipelining.

Supporting pipelining, though, is necessary for several reasons. MEs have very limited code store. If a network application's critical path cannot fit into the code store of a single ME, there is no choice but to utilize pipeline stages. Pipelining may also have beneficial secondary efforts. Pipelining access to multiple locks might result in less contention than duplicating one *aggregate* with multiple locks. It might also reduce capacity misses for software-controlled caching (Section 5.2).

Aggregates are formed heuristically by merging or duplicating *PPFs* to maximize system throughput. When merging, the goal is to improve throughput by reducing communication costs. Here, pairs of *PPFs* with the highest communication costs are placed on the same *aggregate*. Pipeline *aggregate* duplication is used to improve the throughput of the slowest pipe *aggregate* if its throughput is much less than the other pipeline *aggregates*. After the *aggregates* have been formed, frequently executed *aggregates* representing the core packet processing functions are mapped to the MEs while infrequently executed *aggregates* representing support, control and initialization functions are mapped to the XScale processor.

An important consideration in real-time applications like packet processing is worst case execution time (WCET) analysis. Computing bounds on task execution in the system ensures that the network processor can maintain a minimum line rate. This analysis can be incorporated into our current compilation framework through an iterative compilation design. Results of WCET analysis on code produced by the *Code generator* can be fed back into the *IPA and global optimizer* to modify compilation decisions or to notify the programmer that the current program will be unable to achieve the user-specified minimum performance targets.

5.2 Delayed-update software-controlled caching

Packet processing cores in the Intel IXP do not have hardware caches. The common belief is that packet applications lack enough interesting locality to dedicate die-area for caches. For example, little locality exists in packets stored in DRAM since packets are usually processed by one thread and then leave the system completely. Recent studies, though, have shown packet programs have locality in the application data structures. For example, Baer et al [3] as well as Chiueh and Pradhan [6] demonstrated architectures where caching can improve the forwarding rate of packet route lookups.

The Shangri-La compiler utilizes existing IXP hardware to implement a software-controlled cache that tries to exploit available application caching opportunities without hardware caches. On the IXP, the CAM (Section 3.3) can be used to do fast lookups for available cache entries and cache lines can be stored in an ME's Local Memory which is available to all its threads. To identify good caching candidates, expected hit rates and access frequencies for global data access statistics from the *Functional profiler* are used.

To correctly maintain strict cache coherency, access to software-controlled cache entries would have to be protected by critical sections or the home location would have to be checked on every access, both of which would be expensive and eliminate any caching benefits.

Shangri-La generates a novel "delayed-update" software cache that can be used in error-tolerant applications like packet processing. Suitable caching candidates are frequently read data structures that have high hit rates, but are infrequently written. A frequently found pattern are structures that are frequently read by the packet processing cores, but infrequently written by maintenance, control or initialization code. Updates to these structures are not protected by critical sections in the original code, but rely on the coherency of a single atomic write to guarantee correctness of an update.

A delayed-update cache only checks on every *i*th packet for updates at a cache line's home location, as shown in Figure 8. This significantly reduces the frequency and cost of coherency checks, but causes updates to cached entries to be delayed relative to changes in the home location (e.g. in SRAM).

While incoherency is undesirable in normal applications, delayed updates in network programs only causes packet delivery errors. Fortunately, network protocols are tolerant of packet delivery errors. For example, TCP, used for most connection-oriented internet messages, can request retransmission of lost frames in a stream [34]. Quality of service (QoS) routers explicitly drop frames on selected packet streams to throttle bandwidths, and firewalls drop selected packets to secure internal networks from the internet.

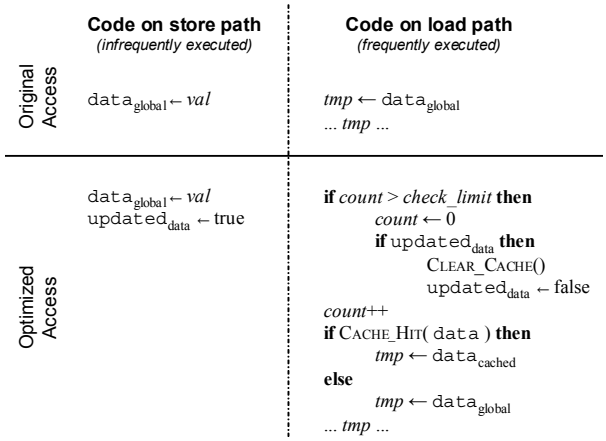

Figure 8 – Code for store and load paths for delayed-update cache to $\texttt{data}_{\texttt{global}}$. Updates to the shared global are detected by changes in the compiler-generated $\texttt{updated}_{\texttt{data}}$.

For a given application, a minimum per-packet load update check rate (r_{load_check}) can be calculated from a user-specified per-packet maximum tolerable packet delivery error rate (r_{error}), the per-packet rate of expected stores to the variable (r_{store}), and the per-packet rate of expected loads to the variable (r_{load}), as shown in Equation 2. As expected, this equation suggests reductions in expected stores or loads can reduce the minimum update check rate.

$$r_{load\_check} \geq \frac{r_{store} \times r_{load}}{r_{error}} \qquad \textbf{Equation 2}$$

5.3 Optimized packet handling

Packet encapsulation (`packet_encap()`, `packet_decap()`, `packet_extend`, `packet_shorten`), packet data access (`data_read`, `data_write`) and metadata access (`meta_read`, `meta_write`) primitives all significantly impact instruction and

memory counts. For example, each packet read and write requires up to {38 + 5 * access_size_in_words} instructions and involves at least one SRAM and one DRAM access. Given the frequency of packet reads and writes in packet processing code, this overhead can be significant relative to the 700 instructions / packet budget for achieving 2.5Gbps on the Intel IXP2400. Given the frequency packet handling operations occur in real application code and the limited per-packet instruction and memory access budget available, optimizing them can result in significant performance improvements.

5.3.1 Packet access combining (PAC)

Network applications are naturally expected to access fields of a packet during processing. Packet data are always stored in DRAM memory on the IXP because packets can be extremely large and in most cases, only the header of the packet is accessed by a network application. According to Figure 6, though, if we simply map every packet access to a DRAM access, packet forwarding rates would be quickly limited by DRAM bandwidth.

To prevent this, an analysis incorporated in the *IPA and global optimizer* and the *Code Generation* stages of the Shangri-La compiler aggressively combines multiple protocol field accesses into a single, wide DRAM access. For example, in Figure 2, the packet fields `dst` and `type` can be accessed together using only one DRAM access. This optimization can also be applied to combine packet SRAM metadata accesses.

Packet accesses to be combined must satisfy three criteria:

- `packet_handles` must be equal.
- The address ranges of the packet data accesses must be adjacent or within a specified bounded range. For the IXP, which is optimized for wide memory accesses, even accesses separated by 32- or 64-bits can benefit from combining.
- The combined data width can not exceed the width that can be accessed by one memory instruction.

Packet access combining is performed in four major steps:

1. Use the criteria above to find the candidates among all packet accesses in an *aggregate*.
2. Compute dominator and post-dominator graph. Packet accesses to be combined must satisfy the dominance relationship (e.g. only dominated reads may be combined).
3. Combined packet accesses must not violate any data dependencies. A data-flow analysis identifies any dependencies between protocol field accesses. In this analysis, a read access is considered a use, and a write access is considered a definition. Two read accesses can be combined if there is no intervening definition to the first field before the second read access. Two write accesses can be combined if there is no intervening use of the first field before the second write access.
4. Combine the packet accesses and eliminate the redundant ones. The remaining packet reads and writes are updated with new memory access offsets and sizes. The removed packet access locations now read temporaries containing the pre-loaded packet data or write temporaries that buffer data to be written out the packet.

5.3.2 Static offset and alignment resolution (SOAR)

Statically determining packet access offsets is almost as important to performance as packet access combining. In network applications, the location and alignment of a given protocol's field is application-context specific. Consider the MPLS over Ethernet packet [28] shown in Figure 9. These packets can have an arbitrary number of MPLS headers attached to the payload (e.g. IPv4 header and data). Consequentially, in applications that process MPLS packets, the locations of the MPLS and IPv4 protocol fields relative to the start of the packet cannot be determined statically. When offsets of protocol fields are not static, the alignment of fields may also be application dependent. Many processor architectures, including IXP, can only perform word-aligned memory accesses.

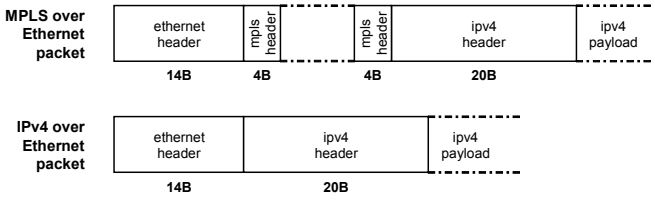

Figure 9 – Illustrates when the offset and alignment of packet fields can (normal IPv4) and cannot (MPLS) be resolved statically.

Handling both unknown field offsets and alignments dynamically at runtime adds significant overheads to packet access primitives. While static alignment resolution can remove only a few instructions, more than half of the 40+ instructions in a packet data access can be removed with static offset resolution. Fortunately, static offsets and alignments can be determined in many instances, but they can only be determined by analyzing the entire packet processing application.

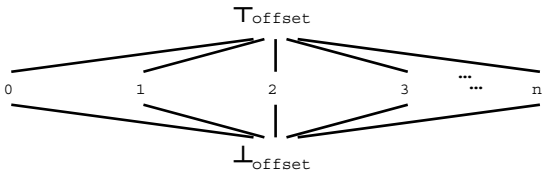

Figure 10 – Lattice for static offset determination (SOD).

We developed a full-program analysis to determine, when possible, static protocol field offsets and alignments for packet accesses in a given packet application. Static offset and alignment resolution (SOAR) is performed in eight major steps. The purpose of the analysis is to statically determine the value of the `head_ptr` (see Figure 3) at all packet access locations:

1. Identify all packet encapsulation (e.g. `packet_encap()` and `packet_decap()`), packet data accesses (e.g. `ph->protocol_field`) and `packet_handle` assignments in the application.
2. Initialize lattice values for static offset determination (SOD). SOD lattice values, shown in Figure 10, correspond to current protocol offset (*`c_offset`*) of a live `packet_handle` relative to the initial `head_ptr`:

   At `packet_handles` entering the receive module (`Rx`),

   $c\_offset \leftarrow 0$

   At all other program locations,

   $c\_offset \leftarrow \top_{offset}$
3. Perform global (inter-procedural and intra-procedural) forward flow analysis of lattice values for SOD. Computed values for *`c_offset`* should be recorded at all packet access program points. The monotonic flow function is described below:

At `packet_encap()`,
$c\_offset_{out} \leftarrow c\_offset_{in}$ − BIT_OFFSET(`packet_encap()`)

At `packet_decap()`,
$c\_offset_{out} \leftarrow c\_offset_{in}$ + BIT_OFFSET(`packet_decap()`)

At control flow joins,

$$c\_offset_{out} \leftarrow \begin{cases} n & \text{if all } c\_offset_{in(i)} = n \mid \top_{offset} \\ \bot_{offset} & \text{otherwise} \end{cases}$$

4. Perform global backward flow analysis of lattice values using the previous flow function, but only apply analysis at program points where $c\_offset = \top_{offset}$. Updated values for c_offset should be recorded at all packet access program points. This backward path is used to propagate static offsets to packets not entering via `Rx` (e.g. at `packet_create()` or `packet_copy()`).

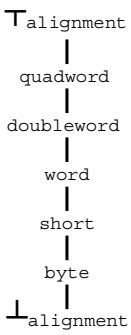

Figure 11 – Lattice for static alignment determination (SAD).

5. Initialize lattice values for static alignment determination (SAD). Lattice values correspond to current protocol alignment ($c\_alignment$) of a live `packet_handle` relative to the `head_ptr`:

At `packet_handles` entering the receive module (`Rx`),
$c\_alignment \leftarrow$ quadword

At all other program locations,
$c\_alignment \leftarrow \top_{alignment}$

6. Perform global forward flow analysis of lattice values for SAD. Computed values for $c\_alignment$ should be recorded at all packet access program points. The monotonic flow function is described below:

At a `packet_encap()`,
$c\_alignment_{out} \leftarrow$ ALIGNMENT($c\_alignment_{in}$ − BIT_OFFSET(`packet_encap()`))

At a `packet_decap()`,
$c\_alignment_{out} \leftarrow$ ALIGNMENT($c\_alignment_{in}$ + BIT_OFFSET(`packet_decap()`))

At control flow joins,

$$c\_alignment_{out} \leftarrow \begin{cases} a & \text{if all } c\_alignment_{in(i)} = a \mid \top_{alignment} \\ \text{MIN\_ALIGNMENT(all } c\_alignment_{in(i)} \text{)} & \text{otherwise} \end{cases}$$

7. Perform global backward flow analysis of lattice values using the previous flow function, but only apply analysis where $c\_alignment = \top_{alignment}$. Updated values for $c\_alignment$ should be recorded at all packet access program points. This backward path is used to propagate static offsets to packets not entering via Rx (e.g. at `packet_create()` or `packet_copy()`).
8. The results of these two dataflow analyses can be used to optimize packet accesses and packet encapsulation in the generated code:

For all packet field accesses with a statically resolved constant offset ($c\_offset \; != \bot_{offset}$), an optimized packet access sequence for a fixed offset can be used in place of a one that must handle unknown offsets:

OFFSET(*field*)
← c_offset + OFFSET_IN_PROTOCOL(*field*)

For all packet field accesses with a statically resolved constant alignment ($c\_alignment \; != \bot_{alignment}$) and an unknown offset ($c\_offset = \bot_{offset}$), an optimized packet access sequence for a fixed alignment can be used in place of one that must handle unknown offsets and alignments:

ALIGNMENT(*field*)
← ALIGNMENT($c\_alignment$ + PROTOCOL_ALIGNMENT (*field*))

For all packet encapsulations (`packet_encap()` and `packet_decap()`) with a statically resolved constant offset, code does not need to be generated to update the `head_ptr` relative to the size of the current encapsulation. Prior to join points where a static offset cannot be resolved ($c\_offset = \bot_{offset}$), code must be inserted that updates the value of `head_ptr` to reflect its current offset. Applying this transformation eliminates instructions and memory accesses resulting from unnecessary updates of `head_ptr`.

5.3.3 Eliminating packet access primitives

In this section, we describe situations where program analysis can be used to identify packet access primitives from a compiled source application that can be completely eliminated in the generated code.

The `metadata` construct is useful for packet processing because it allows state to be attached to a packet as it flows through different *PPFs* and *modules*. For example, in Figure 1, the `l3_fwdr` module can attach a next hop ID to a packet, which the `eth_encap` module uses to find the correct Ethernet header information to encapsulate the packet with. Performance may suffer, though, if metadata accesses are always converted into actual reads and writes of metadata stored into SRAM (see Figure 3). Writes to SRAM are only necessary if the metadata field might be accessed by another ME. In many cases, after aggregation and extensive inlining, a given metadata field may only be live within one *PPF* or *aggregate*. In this case, the metadata access can be simply treated as a local variable.

`packet_encap()` and `packet_decap()` allow arbitrary layering of packet protocols and allow modular packet applications to be written independent of how it may be encapsulated within another application. For example, IPv4 applications can be written to run on Ethernet or to run on any other physical layer protocol. The encapsulation functions update the current `head_ptr` (stored in the packet SRAM metadata) to reflect data prepended to a packet. These encapsulation primitives add memory and instruction overheads. Full support of these primitives is required to enable handling and layering of arbitrary protocols (like the *MPLS* application where an arbitrary number of MPLS packet headers can be prepended), but compiler analysis can identify instances when code generated for the primitive can be completely omitted:

- `packet_encap()` and `packet_decap()` can be eliminated in conjunction with results of the SOAR analysis. If a value of `head_ptr` has been statically determined at a `packet_encap()` and `packet_decap()` location, the `head_ptr` to the current protocol does not need to be maintained and these primitives do not need to be represented at all in the code.
- Paired encapsulation calls (`packet_encap()` → `packet_decap()` or `packet_decap()` → `packet_encap()`) between two protocols can be eliminated if they are paired for every path between them and are called within the same *aggregate*. In this case, the net result relative to other *aggregates* is that the `head_ptr` remains unchanged.

5.4 Stack layout optimization

Implementing a normal program stack is not straightforward due to the Intel IXP's partitioned memory hierarchy and explicit memory instructions for accessing each memory level. Local program variables and spilled register temporaries are traditionally stored in a frame of the program stack. Since Baker does not support recursion and a static call graph can be constructed at compile time, program stack locations could easily be assigned statically to different memory locations.

In Shangri-La, though, the primary goal of stack layout optimization is to allocate as many stack frames as possible to the limited amount of fast memory. Only 48 words of Local Memory are available to each of the eight threads for stack memory (the remaining memory is reserved for other uses like software-controlled caching). To accommodate programs with larger stacks, the stack can grow into SRAM, but its high latency and the consumed bus bandwidth would significantly impact performance if used extensively for the program stack.

Since explicit instructions access each level of memory, the compiler can, for every stack access, either generate instructions and associated control to store in both types of memory, or statically assign it to only one memory level. Since any control overhead would add a significant number of dynamic instructions for every stack access, we opted for the later solution.

In Shangri-La, an *aggregate's* dispatch loop calls *PPFs* (procedures in this discussion) that have packets arriving on its input *CCs* (the procedure's inputs), resulting in a very flat call graph. Given this runtime model, we expect top level procedures in the call graph to be executed most frequently. Hence, the basic stack allocation strategy is to assign Local Memory to procedures higher in the program call graph and assign SRAM memory when Local Memory has been completely exhausted. If a procedure is called from more than one place, its call stack is assigned to the minimum stack location (in Local Memory or SRAM) that will never collide with possibly live stack entries, depending on where it is called from.

Our experiments so far suggest stack locations in SRAM can significantly degrade performance. In initial implementations, the *L3-Switch* application included over 100 dynamic SRAM accesses per packet that came from the stack. Although stack space in Local Memory is small, the call stack in this application also did not exceed 5 frames. It was soon discovered that stack accesses were being mapped to SRAM because the Local Memory stack locations were poorly utilized.

One problem was that initially, to easily accommodate the IXP's offset addressing mode, the stack has a minimum 64B (16 words) frame size. On the IXP, stack entries can be accessed in the same cycle only by using offset-based addressing. In offset-based addressing, the address pointer used to access Local Memory must to be aligned so that the offset can simply be OR'ed to the address pointer (e.g. `$SP[3]` is equivalent to `*($SP | (3 << 2))`. An improved stack layout was implemented that eliminated this minimum stack frame size. Here, the compiler maintains two stack pointers, the physical (`$pSP`) and virtual (`$vSP`) stack pointers, as shown in Figure 12. The physical stack pointer is always properly aligned and the virtual one is sized to the procedure's required minimum. In the final code, only the physical stack pointer is generated, but the virtual stack pointer is used to calculate the correct offset for a stack access relative to the physical stack pointer.

We also confirmed that aggressive inlining improved utilization of the stack. Merging stack frames together eliminates frame boundaries and stack slots reserved for call actual parameters. It also increases global optimization opportunities, which decreases the number of stack slots reserved for temporaries. Both frame size optimizations and aggressive inlining are essential for keeping the runtime stack completely in Local Memory and for achieving good performance on larger network applications.

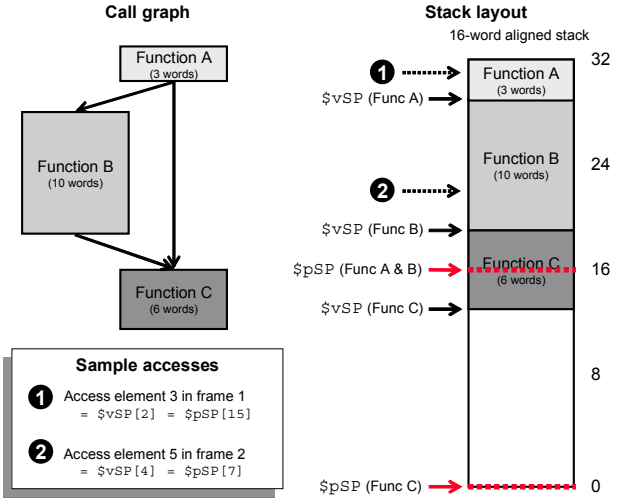

Figure 12 – Illustrates our technique for minimizing call stack frame sizes.

6. Experimental Results

To evaluate code generated from the Shangri-La compiler, we performed experiments on an Intel IXP2400 evaluation board with 8MB SRAM, 64MB DRAM and three 1Gbps optical ports. An IXIA packet generator with three 1Gbps optical ports (to support a maximum of 3Gbps throughput) was used to transmit packets and collect statistics.

6.1 Benchmark applications

Three network applications written in Baker were evaluated, *L3-Switch* (3126 lines), *Firewall* (2784 lines) and *MPLS* (4331 lines).

L3-Switch [27] bridges and routes IP packets. The critical path for *L3-Switch* is the route lookup for the next hop router ID. Next hop IDs are found by traversing a tree data structure (often called a trie) to match the longest matching string of bits from the most significant bits of the destination IP address and retrieving the next hop ID associated with that match.

Firewall sits between an internal network and an external network and prevents selected packets from passing. A classifier attaches flow IDs to packets by matching several packet fields (e.g.

source and destination IPs, source and destination ports, protocol and type of service (TOS)) to an ordered list of user-defined patterns. Selected flow IDs are then dropped by the *Firewall*.

Multiprotocol Label Switching (*MPLS*) [28] routes according to *labels*, instead of destination IPs, attached to packets entering the domain. Routing with *labels* reduces hardware requirements for routing and facilitates high-level traffic control that cannot be achieved by per-hop IP routing.

L3-Switch and *MPLS* were evaluated using NPF packet traces [27][28]. We developed our own packet traces for evaluating *Firewall*.

6.2 Performance Evaluation

Packet forwarding rates and dynamic memory accesses for each application were collected as optimizations were successively enabled. Optimization ordering was done in a way to highlight each optimization, since the benefits of some of the optimizations depend on each other. All optimizations are disabled in the `BASE` configuration, `-O1` adds typical scalar optimizations, `-O2` inlines base packet handling routines, `PAC` enables packet access combining, `SOAR` enables static offset and alignment resolution, `PHR` removes unnecessary packet handling support code and `SWC` enables software-controlled caching.

Table 1 – Dynamic memory accesses per packet.

	Packet			Application		
	Scratch	SRAM	DRAM	Scratch	SRAM	Total
L3-Switch						
+SWC	2.0	3.0	2.0	0.0	8.0	15.0
+PHR	2.0	3.0	2.0	1.0	11.0	19.0
+PAC	2.0	9.0	3.0	1.0	11.0	26.0
+O1	2.0	35.0	29.0	1.0	12.0	79.0
BASE	2.0	35.0	29.0	1.0	22.0	89.0
Firewall						
+SWC	2.0	1.0	1.0	0.3	14.0	18.3
+PHR	2.0	1.0	1.0	0.3	14.0	18.3
+PAC	2.0	5.0	1.0	0.3	14.0	22.3
+O1	2.0	40.6	25.6	0.3	30.8	99.3
BASE	2.0	40.6	25.6	0.6	32.5	101.3
MPLS						
+SWC	2.0	7.0	2.0	0.0	5.0	16.0
+PHR	2.0	7.0	2.0	2.0	9.0	22.0
+PAC	2.0	14.0	3.0	2.0	8.0	29.0
+O1	2.0	23.0	16.0	2.0	9.0	52.0
BASE	2.0	23.0	16.0	2.0	14.0	57.0

For all configurations except three, the program's entire critical packet pipeline was mapped to one ME and then replicated up to five times on the other MEs. The *MPLS* `O1` pipeline and the *L3-Switch* and *MPLS* `BASE` pipelines had to be mapped to two MEs due to ME code size constraints. This pipeline was then replicated two more times on the remaining four MEs. In the most optimized case, we have been unable so far to map the critical path of the benchmark applications to more than one ME. This is due in part to the fact that today, network forwarding applications are still designed to be simple and have short critical paths so that they can handle high packets rates.

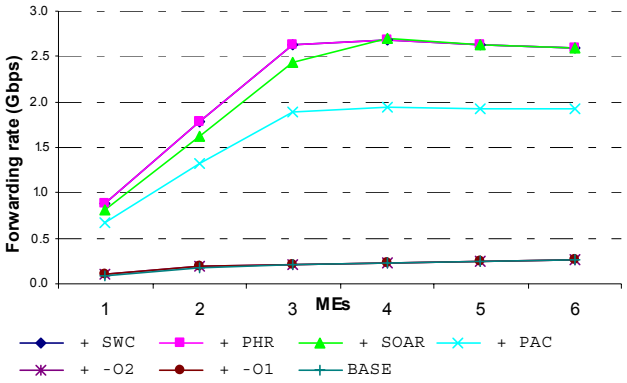

Figure 13 – Packet forwarding rates for *L3-Switch*.

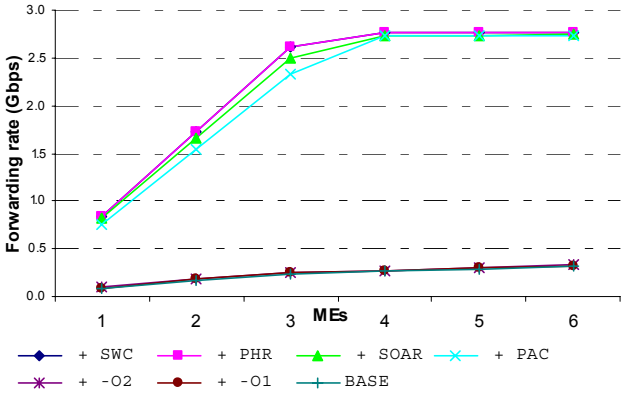

Figure 14 – Packet forwarding rates for *Firewall*.

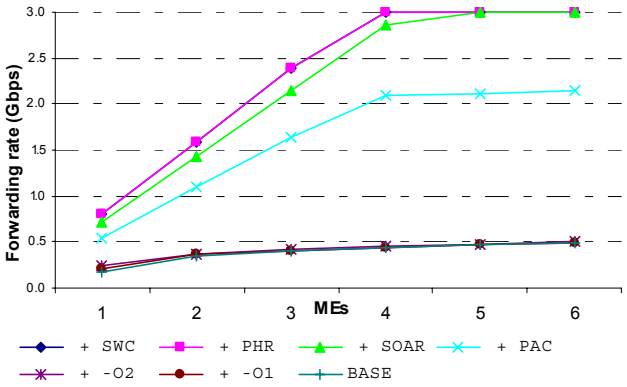

Figure 15 – Packet forwarding rates for *MPLS*.

Table 1 shows the average per-packet dynamic memory accesses for each application as relevant optimizations are enabled (`-O2` and `SOAR` only affect dynamic instruction counts and have no effect on memory access counts). The effects of stack layout optimization described in Section 5.4 are already included in these reported numbers. Without stack layout optimization, even simple programs would generate too many SRAM accesses to achieve respectable packet forwarding rates. The significant impact of `PAC` is evident in this table from the large reduction in packet handling SRAM and DRAM accesses. In the case of *Firewall*, `PAC` even aids the scalar optimizer by exposing additional opportunities to eliminate application SRAM accesses.

Figures 13-15 display packet forwarding rates on minimum

sized 64B packets. Each curve represents the effect of successive optimizations for one to six MEs enabled. Code generated by Shangri-La for all three applications have successfully achieved 100% forwarding rates at 2.5Gbps, which is what the IXP2400 designed for and is the same throughput target achieved by hand-coded assembly versions of the applications written specifically for the processors.

The figures further show that `PAC` improved packet forwarding the most. While PAC eliminates instructions, its largest effect is on reducing DRAM and SRAM accesses. As Figure 6 suggested earlier, having more than just a few DRAM accesses limits the theoretical forwarding rate of the system by saturating the memory bandwidth. This saturation is evidenced by the non-linear increase and flattening in forwarding rates with increasing numbers of MEs enabled for each of the optimization curves. Without memory bandwidth effects, forwarding rates should always increase linearly with more MEs enabled and eliminated instructions should always improve performance with a constant proportion with more MEs enabled.

In the `BASE`, `-O1` and `-O2` configurations, forwarding rate flattening occurs with fewer MEs because there are more memory accesses per packet. The `PAC`, `SOAR`, `PHR` and `SWC` configurations generate fewer memory accesses per packet and only saturate memory bandwidth with at least four MEs enabled.

`-O1` and `SOAR` are important to reducing per-packet instruction counts. Reduced instruction count is important because its effect is multiplied as more MEs are enabled. `-O1` optimizations enable *MPLS*'s and *L3-Switch*'s critical path to fit on one ME instead of two. `SOAR` significantly improves forwarding rates on *L3-Switch* and *MPLS*. Instruction count reductions can be best seen with only one or two MEs enabled on the *L3-Switch* and *MPLS* applications. At these points, memory bandwidth limits are not being hit and improvements in forwarding rates are purely due to instruction count reductions.

The effects of `-O2`, `PHR` and `SWC` appear to have limited effects on application forwarding rates, but `PHR`'s and `SWC`'s ability to reduce dynamic SRAM and Scratch Memory accesses are clearly evident in Table 1. `SWC` successfully caches two small frequently-accessed data structures in *L3-Switch* and *MPLS*.

Our experiments suggest only a rough relationship between the number of memory accesses and the IXP2400's maximum achievable packet forwarding rates. For example, Figure 6 showed that the hardware could achieve 2.5Gbps only if there was 1 DRAM access. Both *L3-Switch* and *Firewall* achieve approximately 2.7Gbps, and *MPLS* achieves 3Gbps, even though all these applications have approximately the same number of DRAM and SRAM accesses in the most optimized configuration.

These inconsistencies suggest that although there is a clear trend between memory accesses and achievable packet forwarding rates, there are also important secondary factors. One secondary factor is the impact of the memory access width on packet forwarding rates, shown earlier in Figure 6. *MPLS* probably achieves higher packet forwarding rates because it issues narrower memory accesses to DRAM than the other two applications (24B vs. 40B). Another possible factor for the discrepancy is the balance between computation and memory accesses. For example, the experiment in Figure 6 had almost no computation, but achieved lower packet forwarding rates than real applications. In this case, the amount of computation and memory access overlap between threads on the same ME may be reduced because all the threads are waiting on memory.

7. Related Work

Click [20] is the most relevant and established academic C++ programming model and environment for building packet processing applications on a single, general-purpose, processor. Baker bears many similarities to Click, especially in regards to its modeling of *communication channels* (*CCs*). The original Click project focused more on the language design than performance: they used a standard C++ compiler and were targeting a general-purpose uniprocessor. Due to architectural and technology differences, it is difficult to make any performance comparison between our system and theirs. Kohler, Morris and Chen [21] later described a source-to-source tool for optimizing Click module configurations. Most of the optimizations they implemented to eliminate modular inefficiencies in a L3-Switch resembled traditional scalar compiler optimizations. The *click-align* optimizer addressed similar packet data alignment issues faced by our system.

Additional work has also been done by other researchers to extend the performance of Click. NP-Click [31] was a project to implement Click on the Intel IXP by replacing code in Click modules with ME instructions. This modularization resulted in a 35% reduction of the packet forwarding rate on minimum sized 64B packets compared to a hand-coded implementation. SMP Click extended the Click runtime system to run an unmodified Click configuration on a SMP [5].

There has been a lot of research recently specifically on programming the IXP, although it has mostly focused on low-level compilation issues. George and Blume [12] developed a network programming framework and a network application language, Nova, but their language is less ambitious and the compiler has mostly focused on scalar optimizations for the IXP. Li and Gupta [24] developed an algorithm that lays out local variables based on access patterns to take better advantage of autoincrement/autoincrement addressing modes available on the IXP. Zhuang and Pande [38] described three different approaches for resolving ME register bank conflicts during register allocation. In a later paper [37], they described how to share registers across threads in a ME to make better use of available architectural registers. Kim et al. [22] described a retargetable compiler infrastructure for network processors, but their target processor was the Paion PPII. Much like our work, they concluded that aggressive reduction of memory accesses is critical in packet processors that do not have caches.

In addition to assemblers, Intel currently has a product toolkit for developing network applications in a C-like language [18]. A newer version of the toolkit is also being developed that supports an auto-partitioning mode that can automatically construct pipeline stages from a program [10]. This compiler achieves similar optimization goals, but assumes a different starting point for the programmer. While the Shangri-La compiler encourages programmers to write small *PPFs*, which are merged or duplicated by the compiler, they assume programmers write large procedures that are partitioned into stages by the compiler.

Both of these commercial compilers [10][18] remove scheduling and register allocation challenges of programming in assembly, but mapping data to memory levels, managing threads and accessing specialized hardware (e.g. hardware queues and CAM) are still the programmer's responsibility. Non-inlined function calls are converted into branches and then registers are globally allocated. Automatic spilling of live registers is supported, but only to one level of memory specified by the programmer [17].

There are a few publications worth mentioning describing work relating to the optimizations highlighted in this paper.

Udayakumaran and Barua [35] also proposed a form of software-controlled caching. In their scheme, the software-controlled cache is used to store register spills to the program stack and prevent it from polluting the hardware data cache. Because our scheme selectively caches global data, it requires a more complex scheme to identify good caching candidates and selectively generate caching code. Additionally, our software-controlled cache also implements a delayed-update coherence mechanism. In comparison, they can completely ignore coherency because they only cache a thread's private stack.

Davidson and Jinturkar [9] described a memory coalescing algorithm for general purpose processors similar to our packet access combining (PAC). This algorithm replaced narrow array access with doubleword accesses in unrolled loops. Memory coalescing implemented extensive profitability checks to factor the realignment costs and limited packing width. Packet access combining is almost always profitable given the high cost of DRAM access on the IXP. Both algorithms perform similar scalar safety checks, but memory coalescing must also handle potential array aliasing. Gupta, Mehofer and Zhang [14], and Stephenson, Babb and Amarasinghe [33] described frameworks for bit-level analysis and optimization that may be useful for analyzing network packet accesses, but neither of these works describe any ideas that bear any resemblance to our optimizations of packet access primitives.

Finally, Avissar, Barua and Stewart [2] discussed techniques for mapping a program stack to heterogeneous memories. Our work is similar to their work in that both have static, not dynamic, mappings to memory levels. In their approach, both global and stack memories are allocated by solving a large linear programming system that incorporates profiling. We also use profiling data for mapping global data structures, but we allocate stack memory and global data separately, and our stack allocation strategy primarily relies on the static program call graph. Avissar, Barua and Stewart's work also does not need to deal with the complexities of stack frame alignment.

8. Conclusions

This paper addresses the challenges of achieving hand-tuned performance on highly resource-constrained network processors on code compiled from high-level languages. We presented a complete framework for aggressively compiling network programs using both traditional and specialized optimizations techniques to aggressively reduce both instruction and memory access counts. Detailed performance evaluations of compiler generated code of three popular network applications on real hardware show the importance of these optimization techniques in achieving 100% packet forwarding rates at 2.5Gbps.

For future work, we will continue with efforts to improve compiled program performance and to try more network applications on our system. We are also considering if some of the highlighted optimizations can be applied to deal with the difficulties of compiling for future general-purpose chip multiprocessors with heterogeneous cores and memories.

9. Acknowledgements

This work would not be possible without significant contributions from Erik Johnson, Aaron Kunze, Steve Goglin, Vinod Balakrishnan, Arun Raghunath and Robert Odell at Intel Communications Technology Lab (CTL); Institute of Computing Technology (ICT) at Chinese Academy of Science; Prof. Harrick Vin and his research group at UT Austin; and our intern Astrid Wang.

10. References

[1] Amaral, J.N., Gao, G.R., Dehnert, J. and Towle, R. The SGI Pro64 Compiler Infrastructure: A Tutorial. In *PACT'00*, Philadelphia, PA, October 2000.

[2] Avissar, O., Barua, R. and Stewart., D. An optimal memory allocation scheme for scratch-pad-based embedded systems. In *ACT Transactions on Embedded Computing Systems (TECS)*, 1(1) pp. 6-26, November 2002.

[3] Baer, J.L., Low, D., Crowley, P. and Sidhwaney, N. Memory Hierarchy Design for a Multiprocessor Look-up Engine. In *PACT'03*, New Orleans, LA, September 2003.

[4] Broadcom Corporation. *The Sibyte BCM1250 Processor.* http://sibyte.broadcom.com/public/index.html

[5] Chen, B. and Morris, R. Flexible Control of Parallelism in a Multiprocessor PC Router. In *USENIX 2001 Annual Technical Conference*, Boston, MA, June 2001.

[6] Chiueh, T. and Pradhan, P. High-performance IP routing table lookup using CPU caching. In *IEEE Infocom'99*, New York, NY, March 1999.

[7] Chow, F., Chan, S., Kennedy, R., Liu, S.M., Lo, R. and Tu, P. A new algorithm for partial redundancy elimination based on SSA form. In *PLDI'97*, Las Vegas, NV, June 1997.

[8] Cooper, K. and Harvey, T. Compiler-Controlled Memory. In *ASPLOS-VIII*, San Jose, CA, October 1998.

[9] Davidson, J. and Jinturkar, S. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. In *PLDI'94*, Orlando, FL, June 1994.

[10] Dai, J., Huang, B., Li, L. and Harrison, L. Automatically Partitioning Packet Processing Applications for Pipelined Architectures. To appear in *PLDI'05*, Chicago, IL, June 2005.

[11] Diwan, A., McKinley, K. and Moss, E. Type-Based Alias Analysis. In *PLDI'98*, Montreal, Canada, June 1998.

[12] George, L. and Blume, M. Taming the IXP Network Processor. In *PLDI'03*, San Diego, CA, June 2003.

[13] Goglin, S., Johnson, E.J. and Vin, H. Baker: A Packet Processing Programming Language for Highly Concurrent Hardware. Under preparation for submission.

[14] Gupta, R., Mehofer, E. and Zhang, Y. A Representation for Bit Section based Analysis and Optimization. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[15] IBM. *The PowerNP architecture.* http://www.hifn.com/products/5np4g.html.

[16] Intel Corporation. *Intel IXP2400 Network Processor: Hardware Reference Manual*. October 2002.

[17] Intel Corporation. *Microengine Version 2 (MEv2): Microengine C Compiler Coding Considerations*. June 2003.

[18] Johnson, E.J. and Kunze, A. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, Hillsboro, OR, April 2003.

[19] Ju, R., Chan, S. and Wu, Chengyong. Open Research Compiler for Itanium Processor Family. Tutorial in *MICRO-34*, Austin, TX, December 2001.

[20] Kohler, E., Morris, R., Chen, B., Jannotti, J. and Kaashoek, M.F. The Click Modular Router. In *ACM TCS*, 18(3) pp. 263-297, August 2000.

[21] Kohler, E., Morris, R. and Chen, B. Programming language optimizations for modular router configurations. In *ASPLOS-X*, San Jose, CA October 2002.

[22] Kim, J., Jung, S. and Park, Y. Experiences with a Retargetable Compiler for a Commercial Network Processor. In *CASES'02*, Grenoble, France, October 2003.

[23] Kulkarni, C., Gries, M., Sauer, C. and Keutzer, K. Programming Challenges in Network Processor Deployment. In *CASES'03*, San Jose, CA, October 2003.

[24] Li, B. and Gupta, R. Simple Offset Assignment in Presence of Subword Data. In *CASES'03*, San Jose, CA, October 2003.

[25] Narlikar, G. and Zane, F. Performance Modeling for Fast IP Lookups. In *SIGMETRICS'01*, Cambridge, MA, June 2001.

[26] Intel Corporation. Microengine Version 2 (MEv2): Microengine C Compiler Coding Considerations. June 2003.

[27] Network Processing Forum. *IP Forwarding Application Level Benchmark.* http://www.npforum.org/techinfo/ipforwarding_bm.pdf

[28] Network Processing Forum. *MPLS Forwarding Application Level Benchmark and Annex.* http://www.npforum.org/techinfo/MPLSBenchmark.pdf

[29] PMC-Sierra. *MIPS-based™ Processors.* http://pmc-sierra.com/processors/

[30] Rosen, E., Viswanathan, A. and Callon, R. RFC 3031 – Multiprotocol Label Switching Architecture. IETF, January 2001.

[31] Shah, N., Plishker, W. and Keutzer, K. NP-Click: A Programming Model for the Intel IXP1200. In 2nd *Workshop on Network Processors (NP-2)*, Anaheim, CA, February 2003.

[32] Shah, N., Plishker, W. and Keutzer, K. Comparing Network Processor Programming Environments: A Case Study. In *2004 Workshop on Productivity and Performance in High-End Computing (P-PHEC), HPCA-10*, Madrid, Spain, February 2004.

[33] Stephenson, M., Babb, J. and Amarasinghe, S. Bitwidth Analysis with Application to Silicon Compilation. In *PLDI'00*, Vancouver, BC, June 2000.

[34] W.R. Stevens. *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley, Boston, MA, 1994.

[35] Udayakumaran, S. and Barua, R. Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems. In *CASES'03*, San Jose, CA, October 2003.

[36] Vin, H., Mudigonda, J., Jason, J., Johnson, E.J., Ju, R., Kunze, A. and Lian, R. A Programming Environment for Packet-processing Systems: Design Considerations. In *3rd Workshop on Network Processors & Applications*, Madrid, Spain, February 2004.

[37] Zhuang, X. and Pande, S. Balancing Register Allocation Across Threads for a Multithreaded Network Processor. In *PLDI'04*, Washington, DC, June 2004.

[38] Zhuang, X. and Pande, S. Resolving Register Bank Conflicts for a Network Processor. In *PLDI'03*, New Orleans, LA, June 2004.