

A Programming Environment for Packet-processing Systems: Design Considerations

Harrick Vin
Jayaram Mudigonda
<vin,jram>@cs.utexas.edu
Department of Computer
Sciences, University of Texas at
Austin

Jamie Jason
Erik J. Johnson
Roy Ju
Aaron Kunze
<jamie.jason, erik.j.johnson,
roy.ju,
aaron.kunze>@intel.com
Intel Research and
Development

Ruiqi Lian
lrq@ict.ac.cn
Institute of Computing
Technology, Chinese
Academy of Sciences,
Beijing, PRC

Abstract

In this paper, we describe the vision and the design of a programming environment, called Shangri-La, aimed at making future generations of packet-processing systems – multi-core, light-weight threaded hardware in general, and network processor (NP)-based systems in particular – as easily programmable as today’s workstations and servers. Our environment consists of: (1) a domain-specific programming language for specifying packet-processing applications, (2) a compiler that incorporates profile-guided techniques for mapping packet-processing applications onto complex packet-processing system architectures, and (3) a run-time system that dynamically adapts resource allocations to create systems that are robust against attacks and that optimize performance and power consumption for the current network conditions. We justify our design and articulate the challenges in designing each of these components.

1. Introduction

The design of packet-processing systems is required to meet two, often-conflicting, requirements: (1) support a large number of high-bandwidth links, and hence large system throughputs, and (2) offer a wide range of services as varied as conventional forwarding functions, VPN, intrusion detection, differentiated services, and overlay network processing. To simultaneously meet these requirements, a new breed of processors, referred to as *network processors (NPs)*, has emerged [1][2][3][4][5][6]. Network processors, much like general-purpose processors, are programmable and include mechanisms—such as multiple processor cores per chip and multiple hardware contexts per processor

core—that enable them to process packets at high rates. These mechanisms foreshadow a more general trend towards the design of multi-core, multi-threaded architectures targeted for high-throughput computing environments. NPs, when combined with general-purpose processors, fixed-function coprocessors and other reconfigurable logic elements, create a powerful hardware platform for designing packet-processing systems.

Unfortunately, the methodologies needed to map packet-processing applications onto NP-based packet-processing systems are unfamiliar to many programmers, and the tools to perform such mappings are in their infancy. Today, programmers are often required to manually partition each network application into components at design time and map these components onto different types and instances of processors within a packet-processing system. The different types of processors available within a packet-processing system (and even within an NP) are often programmed using separate programming environments. In most cases, general-purpose processor cores included in packet-processing systems are programmed using conventional programming languages (e.g., C), compilers (e.g., the GNU C compiler), and operating systems (e.g., Linux). Programming environments for special purpose cores in NPs, on the other hand, typically have their own tools and development methodologies that usually expose hardware details within the programming language and provide limited, if any, run-time systems for managing resources. The programmer, in turn, is required to develop hand-tuned code that carefully manages a variety of resources while ensuring that the system can process incoming traffic at the line rate.

Such hand-tuned resource mapping decisions are made at design time and are based on the performance expectations of the application, expected workload, and

exact hardware configuration of the system. Consequently, when an application is ported from one platform to another, the performance rarely scales as expected due to mismatches between the mappings, workloads, and the new hardware.

Even recent attempts at complete programming environments for NPs expose most of the hardware details to the programmer and involve the programmer in mapping applications to packet-processing system resources [8][18]. We predict that future generations of packet-processing systems will support a larger number and more types of processors, more diverse memory hierarchies, and more complex processor interconnects; hence, the difficulty in programming packet-processing systems will only increase over time.

In this paper, we describe our vision and design of a programming environment aimed at making future generations of packet-processing systems as easily programmable as today’s workstations and servers. Our programming environment consists of: (1) a domain-specific programming language for specifying packet-processing applications, (2) a compiler that incorporates profile-guided techniques for mapping packet-processing applications onto complex packet-processing system architectures, and (3) a run-time system that dynamically adapts resource allocations to create systems that are more robust against attacks, achieve higher performance, and consume less power than current systems on similar hardware. The resulting environment facilitates rapid development of portable, high-performance packet-processing applications on programmable packet-processing systems.

The rest of the paper is organized as follows. In Section 2, we expose the characteristics of packet-processing applications and describe some of the common architectural features found in most NPs and packet-processing systems. In Section 3, we describe the overall design of our programming environment with Section 4 providing details of each component of our design, focusing particularly on the key research challenges and requirements. Finally, Section 5 summarizes our contributions and current status.

2. Problem Domain

Our objective is to design a programming environment that facilitates rapid development of portable, high-performance packet-processing applications on multi-core, light-weight threaded packet-processing systems in general and current NP-based systems in particular. In what follows, we first expose the characteristics of packet-processing applications and describe some of the common architectural features found in most NPs and packet-processing systems. We argue

that NP and packet-processing system architectures are qualitatively different from general-purpose systems and have a more complex processor structure. Further, packet-processing applications are structurally different from other applications where high performance (throughput) is required.

2.1. Packet-processing Applications

Packet-processing applications receive, process, and transmit data units—generally referred to as packets or cells¹. To understand the requirements these applications impose on system architectures and programming environments, we have analyzed several packet-processing applications and derived their fundamental characteristics.

- Packet-processing applications can be described using cyclic data-flow graphs [5][25]. Most of these graphs possess the following characteristics:
 - Many packet-processing functions (i.e., data-flow actors) are *stateful* – they maintain per-flow state. The state is accessed and updated while processing packets belonging to the flow. The statefulness of functions when combined with burstiness of traffic ensures that the number of packets belonging to different flows and of different types processed by the application can vary greatly over time; hence, the system does not reach a steady-state of operation.
 - The execution of processing functions in the data-flow graph is triggered by packet arrivals, timers, or other hardware events (e.g., link failure). The sequence of functions executed for a packet depends on the packet’s type (determined based on its header and content) and the state of the system.
 - Packet-processing functions can be represented as *M-in, N-out* data-flow actors. On processing a packet, some functions generate multiple output packets (e.g., multicast routing); some functions process multiple packets and generate one output packet (e.g., IP defragmentation); while some other functions generate output packets (e.g., keep alive) without consuming any packets.

¹ In this paper, the term packet is used to refer to any unit of network data transfer, including, but not limited to datagram, frame, cell, and packet.

- In most applications, there is little or no dependence between packets belonging to different flows². Hence, packet-processing applications exhibit a high-degree of parallelism while processing packets from different flows. However, within flows, many packet-processing applications require—because of shared flow state or other ordering constraints—packets belonging to a flow to be processed and transmitted in a particular order.
- Throughput, as opposed to delay, is the primary performance metric for many packet-processing applications. Further, individual functions in most packet-processing applications are often not compute-intensive; their performance is dominated by memory-access latencies (resulting from operations such as compression table lookup, routing table lookups, etc.). Emerging packet-processing applications often make hundreds of memory accesses per packet and hiding those memory access latencies to achieve better throughput is paramount.

2.2. Network Processor and System Architectures

To exploit the inherent flow-level parallelism present in this class of applications, NPs support multiple cores for processing packets. Many NPs also include general-purpose processors to support infrequent but complex operations, and special-purpose units to perform operations such as encryption and checksum computation.

To hide memory access latencies, NPs often support *hardware multi-threading*. With hardware multi-threading, when a thread blocks, on a memory access for example, another thread within the same processing unit takes over and processes a different packet. Further, to reduce the memory access latencies and contention (and thereby limit the number of hardware threads needed to utilize processor cores fully), NPs often support a multi-level memory hierarchy.

To enable multiple processor cores to communicate effectively, NPs typically contain at least one, but usually many, forms of inter-processor communication (IPC) mechanisms. For example, inter-thread signaling mechanisms are generally fast, but convey only a coarse granularity of information, whereas atomic shared memory operations can be quite fine grained, but have longer access latencies. NPs often also provide hardware support for rings, queues, and blocking and non-blocking forms of synchronization. NP architectures from multiple

vendors instantiate these architectural features [1][2][3][4][5][6].

Depending on the types of applications to be supported, a packet-processing system may combine one or more NPs, general-purpose processors, co-processors, and programmable logic elements. Thus, from a programmer's perspective, NPs and packet-processing systems contain processors with heterogeneous capability—some processors may be fully programmable (e.g., the micro-engines and the Intel XScale[®] processors in the Intel IXP2800 network processor), some may only support configuration (e.g., a classification co-processor), while some others may be fixed-function (e.g., a hash/crypto unit). Processors may communicate with each other using several different communication mechanisms (e.g., shared memory and message passing) with different bandwidth and latencies of communication. Finally, the memory system may support specialized functional capabilities (e.g., atomic increment/decrement operation, ternary CAM).

2.3. Solution Requirements

The unique characteristics of packet-processing applications, when combined with the sophistication of NP hardware, have resulted in a lack of good tools to automatically map such applications onto NPs. Today, application designers map applications to hardware manually at design time. The complexity of such mappings and their variability across applications, architectures, and workloads makes programming NPs complex, tedious, and error-prone. Furthermore, when applications developed for one platform are ported to other platforms the performance does not scale as expected, and packet-processing systems so designed cannot adapt to dynamic changes in traffic conditions.

These observations lead to five requirements for designing a programming environment for packet-processing systems.

1. The programming environment should abstract away architectural details from the programmers. It should allow programmers to specify entire packet-processing applications (control- and data-path functions) without partitioning them across different types of processors.
2. The programming environment should automate the allocation of processor, memory, and communication resources available in packet-processing systems to packet-processing functions.
3. The programming environment should support mechanisms for dynamic adaptation of resources allocated to applications. These mechanisms should allow packet-processing systems to

² Flows refer to a sequence of related packets (generally transmitted between two application end-points).

provide performance guarantees to flows under fluctuating traffic conditions as well as ensure robustness of the system under attacks.

4. The programming environment should enable packet-processing systems to achieve performance comparable to hand-tuned system.
5. The programming environment itself should be retargetable and extensible.

In what follows, we describe our approach to designing a programming environment for packet-processing systems.

3. Shangri-La: A Programming Environment for Packet-processing Systems

We apply the classic *divide-and-conquer* methodology of system design to develop a programming environment that meets the above requirements. The architecture of our overall system, referred to as *Shangri-La*, is illustrated in Figure 1.

To facilitate the development of packet-processing

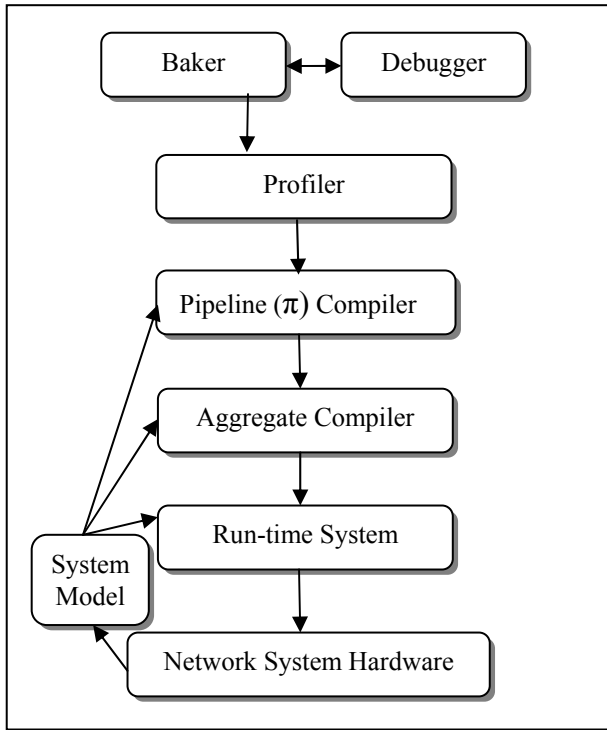


Figure 1: System architecture

applications, Shangri-La defines *Baker*, a modular, domain-specific programming language that allows a network application to be specified as a composition of *packet-processing functions (PPF)*. Baker allows unified specification of entire packet-processing applications (control and data path). A Baker application is compiled

to an abstract machine (captured by an Intermediate Representation (IR)), and is fed into the compilation system of Shangri-La.

The compilation system of Shangri-La consists of three components: the profiler, pipeline compiler (or π -compiler), and aggregate compiler.

The profiler (1) derives code and data structure profiles—such as the locality properties of data structures, frequencies of executions for different PPFs, the amount of communication between each pair of PPFs, etc.—by emulating the execution of the network application under representative traffic conditions; and (2) annotates the IR with the resulting profiles. It is important to note that the profiler emulates the abstract machine without any knowledge of the target packet-processing system architecture; hence, it can only derive a functional profile, and not cycle-accurate performance profile for the applications. The annotated code is subsequently fed to the π -compiler.

The π -compiler addresses the question: how should the application (code and data structures) be organized to use the available packet-processing system resources effectively? First, it uses the data structure profiles to derive a strategy for explicit management of the memory hierarchy. This may involve mapping or prefetching data structures to specific levels of the memory hierarchy, as well as identification of data structures that should be managed using different memory management policies (e.g., selecting different caching policies for different data structures). Second, the π -compiler clusters PPFs into appropriate pipeline stages (referred to as *aggregates*), determines the allocation of resources for each aggregate (e.g., whether or not an aggregate needs to be replicated), and derives an initial mapping of aggregates onto the multiple processors available in the packet-processing system.

The aggregate compiler uses the aggregate definitions and memory mappings from the π -compiler and produces an optimized binary for each of the target processor types available in the packet-processing system. The aggregate compiler performs traditional compiler tasks, machine-dependent optimizations, and domain-specific transformations.

Finally, the *run-time system (RTS)* includes facilities to load and execute aggregates, monitor traffic fluctuations, system performance, and power consumption, and adapt the resources allocated to aggregates such that application performance and energy consumption targets can be met even in the presence of dynamic fluctuations in traffic conditions. Additionally, the run-time system includes a resource abstraction layer that exports high-level interfaces for a wide range of hardware resources, while hiding the details of their implementation (e.g., exporting communication channel

as an abstraction while hiding the details of how the channel is realized on the hardware). The run-time system binds the interface to an appropriate implementation at load/run-time. This facilitates dynamic adaptation of resource allocations to aggregates without requiring a recompilation.

Observe that architectural properties—such as the number and types of processing cores, properties of different levels of the memory hierarchy, etc.—need to be exposed to the π -compiler, the aggregate compiler, and the run-time system. Shangri-La captures these architectural details in a *system model*; thus allowing the π -compiler, the aggregate compiler, and the run-time system to be parameterized for different packet-processing system hardware configurations by altering the model.

4. Design Details and Challenges

Each of the three major components of the Shangri-La architecture—language, compiler, and run-time system—poses its own set of research challenges and key design considerations. The following sections describe these challenges along with the key design considerations and tradeoffs we have made for Shangri-La.

4.1. Baker: A Domain-specific Programming Language

A language in Shangri-La must play two roles: (1) as the interface to the programmer for expressing the application semantics, and (2) as an interface to the compiler to enable the generation of efficient executables on the target system. While the first role of a language is undeniably important, for the Shangri-La architecture and the Baker language, it is the second role that is crucial. The design of Baker, thus, has primarily focused on the second role by asking the question: ‘what does the Shangri-La compiler complex require to perform its task of automated mapping and efficient code generation?’ Secondary to this question, we have tried to answer the question: ‘how does the programmer efficiently express the application?’ Fortunately, in most instances, both questions can be satisfied by creating domain-specific constructs from which the compiler can infer concurrency information and using which the programmer can easily express application semantics.

4.1.1. Baker Language Features. The Baker language is syntactically similar to C, but with data-flow and other packet-processing domain extensions. Like all data-flow languages, Baker has actors, called *packet-processing functions (PPFs)*, inter-actor communications conduits,

called *channels*³, and data appropriate for usage on channels, called *packets*. These constructs are illustrated

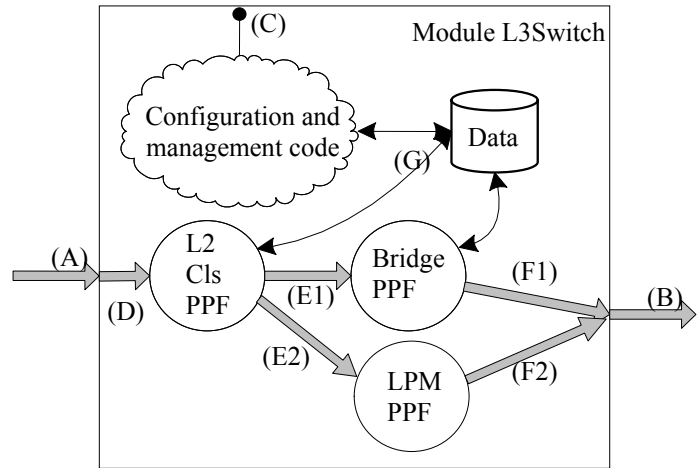


Figure 2: Baker data-flow constructs

in Figure 2 and the subsequent code segment, and explained in more detail below.

```

1. // "Protocol" definition
2. protocol Ethernet {
3.   dest : 48;
4.   src  : 48;
5.   len_type : 16;
6.   snap : anyof {
7.     { len_type > 1500 } : 0;
8.     default: LLCSNAP; // Not shown
9.   }
10.  demux { (len_type < 1500) ? 22*8 : 14*8 };
11. };

12. // L2 Classifier process function
13. void L2Cls.process(Ethernet_packet_t* p) {
14.   IPv4_packet_t* ip;
15.   if (p->len_type < 1500) processSnap(p);
16.   else
17.     if (p->len_type == 0x800 &&
18.         isInIfaceList(p->dest)) {
19.       ip = packet_decap(p); //Uses demux value
20.       channel_put(forward_chnl, ip);
21.     } else
22.       channel_put(bridge_chnl, p);
23. }

24. module L3Switch {
25.   ppf L2Cls; //forward declarations of
26.   ppf Bridge; //required PPFs
27.   ppf LPM;

```

³ Or communications channels

```

28. channels {
29.   input Ethernet_packet_t input_chnl; //(A)
30.   output packet_t output_chnl; //(B)
31. }
32. wiring {
33.   //equate this module's external channels
34.   input_chnl = L2Cls.input_chnl; //(A=D)
35.   output_chnl = Bridge.output_chnl; //(B=F1)
36.   output_chnl = LPM.output_chnl; //(B=F2)
37.   //bind internal PPFs channel endpoints
38.   L2Cls.bridge_chnl -> Bridge.input_chnl; //(E1)
39.   L2Cls.forward_chnl -> LPM.input_chnl; //(E2)
40. }
41. //module's data
42. iface_table_t iface_tbl;

43. //module's interface
44. void add_interface(iface_t r);
45. void del_interface(iface_t r);
46. };

```

An Ethernet bridging and IPv4 forwarding application (an “L3 switch”) could contain PPFs for route lookup and bridging. It also contains configuration and management code that adds and removes interface information from an interface table (e.g., up, down, MAC addresses, etc.).

PPFs contain the actual packet-processing code in an application. This code is expressed in much the same way a C function is expressed (lines 13-23). The inputs to the function represent packets from the input channel endpoints of the PPF. The function explicitly places packets on its output channel endpoints (lines 20, 22), and the data structures accessed are those available within the scope of the PPFs compilation unit. Currently, Baker allows only a single instance of any PPF in an application. One interesting topic for further exploration is how to instantiate PPFs without sacrificing the efficiency of PPF’s accessing per-instance data structures.

Channels carry data, typically packets, between the output and input channel endpoints of PPFs. Channels represent a wiring of the data-flow through the PPFs of an application. PPFs can have passive input channel endpoints, in which data arrival implicitly invokes the processing code of the PPF (e.g., although not shown, L2Cls.input_chnl is passive and so the L2Cls.process function is invoked implicitly). PPFs can also have active inputs, in which retrieval of data from the channel is explicit in the packet-processing code. Active inputs enable the programmer to express such PPFs as a scheduler in a quality-of-service application.

In Baker, packets are declared to be of a new, first-class data type, `packet_t`. Packets can be accessed

through a protocol specification and associated with meta-data, as follows:

- Protocol specifications, which are written by Baker programmers, enable packets to be viewed according to a particular layout (lines 1-11). For example, a bridging PPF may choose to view a packet through an Ethernet protocol whereas a routing operation may choose to view the same packet through an IPv4 protocol. Protocol specifications insulate the programmer from packet memory layout and alignment issues; the compiler and RTS may decide to represent packets in the most appropriate manner for the target hardware--for example, as a contiguous block of memory, or as a chain of buffers--without the Baker programmer knowing.
- Meta-data is used to convey per-packet information, such as input and output port, and flow identifiers, between PPFs. Meta-data is user-defined in Baker, is created by one PPF and consumed by another, and is carried with a packet through the channels of Baker.

We are currently exploring how Baker can specify flows of packets. Our current approach uses meta-data to define flows such that packets within the same flow can then be ordered and serialized according to the specification of the programmer. For example, all of the packets entering a order-sensitive PPF could be ordered according to flow-ID-based piece of meta-data as well as a monotonically increasing sequence number (e.g., a received packet number). However, more work remains as this is a key area for a packet-processing language.

Finally, although not strictly part of a data-flow model, Baker defines modules (lines 24-46), which represent a namespace for other modules, PPFs (lines 12-23), channels (lines 28-31), shared data (line 42), and configuration and management code (lines 44-45). Modules may also contain input and output channels of their own (e.g., A, B) that are wired to the input and output channels of their contained PPFs (e.g., A wired to D). Constructs such as module and configuration code are necessary for the expression of a complete packet-processing application (including control-plane processing), as well as being convenient for the organization of a programmer’s code. Similar to PPFs, only one instance of a module can currently exist in a Baker application, and an interesting area of further research is how to instantiate modules efficiently.

4.1.2. Using Baker Language Features. Just saying Baker is a data-flow language is not sufficient for the programmer to properly understand how the compiler will extract the parallelism of the application. PPFs and channels must not only be able to express the packet-

processing application characteristics stated earlier (e.g., statefulness, flow-specifications, etc.), the programmer must decompose the application into these constructs (which may be a non-trivial task in itself) while understanding how this decomposition may affect the final performance and correctness of the generated code. To this end, Baker defines an *implicit threading model*. A programmer must assume that any PPF may be replicated, and hence execute concurrently on multiple threads. The programmer cannot create or destroy threads, however. Instead, the programmer must assume an implicit threading model as illustrated in Figure 3. In this threading model, each input can conceptually be thought of as an independent thread. In this sense, channels are akin to queues; however, Baker does not strictly enforce this property of PPF channels. Instead, while the programmer thinks of channels as queues, the compiler may implement channels through either function calls or queues (but still must preserve the queue-like semantics of course). As we describe in the following section, this non-strict definition of channels is important for the compiler to maximize throughput.

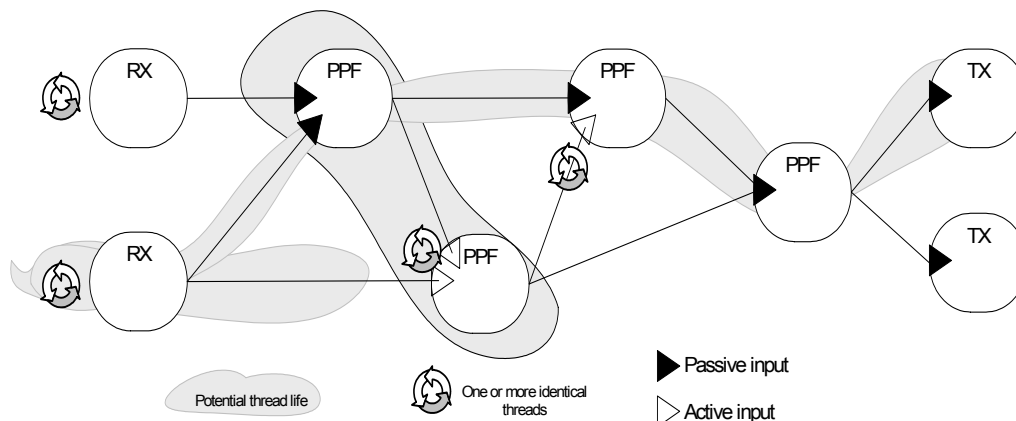


Figure 3: Baker's implicit threading model and passive and active inputs

4.1.3. Implications of Baker Language Features: a Compiler's Perspective. In order to generate code that can achieve high throughput on an NP-style parallel system architecture, a compiler must be able to derive and exploit the inherent parallelism in the application's *code* and *data*. Baker enables the Shangri-La compiler complex to understand those functions that are independent (i.e., PPFs), as well as the data on which those functions can be replicated (i.e., packets and their ordering constraints).

However, while Baker does enable the compiler to extract the inherent parallelism of the application, it is equally important that Baker does not enforce any constraints that could limit the compiler's ability to exploit such parallelism. The implicit threading model and non-strict definitions of channel implementations of

Baker are examples of where we made a conscious decision in the language to not restrict the choices of the compiler and run-time system. These two language features enable the compiler to decide exactly how to map the PPFs to processing cores, which is important because this mapping may depend on workload and the hardware architecture as well as the application itself. For example, the workload may change the locality properties of data, which affects where queues should be placed in the processing pipeline—something possible because of the non-strict definitions of channel implementations in Baker. As for hardware architecture considerations, the exact number of threads within a processor dictates the relative compute-to-I/O ratios of code that should be executed on those processors, and hence the implicit threading model of Baker enables the compiler to control exactly onto how many threads a PPF is replicated.

While no one existing language meets our requirements for Shangri-La and Baker, some parts of existing languages contain useful concepts that we have borrowed. The most notable of these is the data-flow concepts from Click [25]. However, while Click, as well as languages for other extensible router frameworks—

Genesis [26], NetScript [35], NetBind [12], VERA [23], Scout [28], Router Plugins [15][27], and PromethOS[24]—support creation of network applications through composition of modular components [19], most of these languages utilize C/C++ or other general-purpose programming languages for developing the modules; hence, it is difficult for the compiler

to extract the concurrency information for efficient mapping of applications onto packet-processing system architectures. In many of these systems, the mapping of components onto hardware resources is performed by hand, or the hardware platform assumed is uniprocessor, or the languages restrict the choice of mappings so as not to account for workload or hardware architecture variations.

One solution to extracting parallelism from general purpose languages is through language extensions such as OpenMP and related work [30]. Although these solutions ease the burden of extracting parallelism in programs, they tend to introduce too much overhead (e.g., explicit fork/join) or don't lend themselves to the type of

parallelism inherent in packet processing (i.e., pipelined functions as opposed to loops and vector operations).

Finally, languages, such as microC from Intel [5] or picocode from IBM [3], expose hardware details to the programmer, and hence fail to meet our basic requirements of portability. Similarly, languages such as Network Classification Language (NCL) [5] and Functional Programming Language (FPL) [7] offer only limited expressibility; programs expressed in these languages do not completely describe all of the packet-processing operations.

4.2. Profile-guided, Automated Mapping Compiler

The compiler complex of Shangri-La consists of the profiler, π -compiler, and aggregate compiler.

4.2.1. Profiler. In the Shangri-La architecture, the runtime characteristics of a network application—such as the locality properties of data structures, frequencies of executions for different PPFs, the amount of communication between each pair of PPFs, etc.—are used to guide the allocation of processor, communication, and memory resources of packet-processing systems to applications. Such profile-driven compiler optimizations are not new; code layout, for example, has previously been improved through profile-guided optimizations. However, in most previous profile-guided optimizations, the profile data has been derived by first compiling the code without profile information, then executing this code with instrumentation to gather the profile information, and finally recompiling the code with the newly gathered profile information. In the Shangri-La architecture, we do not believe this approach to be feasible because it requires a reasonable first compilation and mapping of the application without profile information. In addition, collecting profile information in hardware requires intrusive instrumentation of the code, or is restricted to those statistics available through hardware-based performance monitoring units.

Instead, the Shangri-La profiler derives code and data structure profiles by emulating the execution of the network application using the IR produced by the Baker parser. In addition to the IR produced from the Baker language, to profile the run-time characteristics of a network application, the profiler needs: *Application state* that contains any persistent state that the network application uses to determine actions performed on packets—this includes, for instance, a route table, flow-classification data structures, and any per-flow state, and a *packet trace* that identifies a representative mix and arrival pattern for packets at the target packet-processing system. The profiler derives statistics for the properties of

interest through a functional emulation of the application under these representative conditions. Because the profiler emulates the abstract machine with sample packet traces, the compilation time is certainly increased but this cost is expected to be justified with a gain in runtime performance. Examples of profiling information include execution frequencies of code sequences, amount of data communicated through channels, access frequencies of data objects, etc. This information can guide a variety of program transformations and code optimizations. For example, execution frequency can influence code layout, and memory access frequency can help determine the layout of data objects across the levels of memory hierarchy. Given that the profiler is invoked at an early stage of compilation, the abstract machine emulates based on the programming model at the source language level and does not assume much knowledge of target processors. Therefore, the abstract machine is not expected to provide accurate performance information.

Although conceptually straightforward, the design and implementation of the profiler poses one primary challenge: scalability. Any limits in the scalability of the profiler are due to at least two factors: (1) complexity of functional simulation, and (2) difficulty in dealing with packet traces. To control the complexity of the simulation environment requires a parameterized simulator, wherein the level of detail can be refined selectively and progressively. Further, since the refinements may depend on the traffic, the profiler needs to be self-tuning. We are exploring the design of a profiler that allows controlled, progressive refinement of the profile studies. While designing the profiler, a key challenge will be to identify specific properties of interest and then derive appropriate sampling techniques that can reduce the profiling complexity considerably.

4.2.2. Pipeline Compiler. The pipeline compiler (π -compiler) partitions a packet-processing application into a series of tasks (called aggregates), which form the processing stages in a pipeline. On IXP-based NPs, for example, these pipeline stages can be mapped to multiple chained microengines (MEs) as well as the Intel XScale® processor. The π -compiler has two primary functions: (1) it manages the memory hierarchy to minimize average memory access times; and (2) it groups packet-processing functions into *aggregates* such that these aggregates, when mapped onto the multiple processor cores, can maximize the overall throughput.

While the π -compiler derives aggregates, it is important to have a well-engineered cost model to consciously guide each aggregation step. The cost model includes factors such as the cost of communication, synchronization overhead, memory access latencies, CPU execution times, and code size. Although it may sound

appealing to simply minimize the processing time of the dominant stage in the partitioned tasks to maximize the rate of packet processing in the pipeline, this tends to split the PPFs of an application into too many aggregates, increasing communication cost and the number of processor cores allocated in the pipeline. Since the pipelined tasks can be replicated as multiple threads on one or more MEs to process multiple packets in parallel, it is important to balance the rate of pipelined tasks (i.e. the number of packets processed in the pipeline within a given time) and the available amount of parallelism to concurrently process the packets in replicated pipelines. The ultimate objective is to maximize the amount of packets processed in the complete system within a given period of time.

There is a large body of parallel programming research on designing algorithms for mapping computation onto multiprocessors [11][21][31][42][40][29][32]. The research can be broadly classified into two categories. The first category of research focuses on the problem of mapping parallel (data- and task-parallel) computations on multi-processors [11][21][31][42]. Most of these techniques derive a mapping such that the execution time of a single instance of the program is minimized. For packet-processing applications, on the other hand, the optimization criterion is maximization of average or worst-case packet-processing throughput. The second category of research addresses the problem of mapping pipelined computations (e.g., streaming and DSP applications) onto multi-processors [20][38][39]. This work is more closely related to the problem at hand. However, most of the prior work makes assumptions that all the units of work go through a single sequence of pipeline stages, and pipeline stages are performing computationally intensive tasks (hence, when two pipeline stages are fused to create a new pipeline stage, its execution time requirement can be estimated simply as the sum of the execution time requirement of the component stages). These assumptions do not hold for packet-processing applications. As we have argued earlier, at any instant, a packet-processing system may process multiple packets, each of which may execute a different sequence of functions. Therefore, in this work, we are investigating novel algorithms for clustering, allocation, and mapping of packet-processing applications onto heterogeneous, multi-processor architectures.

Although it is widely known that packet data structures (e.g., packet header, payload, packet meta-data) have little locality, we have shown that application data structures (e.g., per-flow state such as a meter, header compression state, a trie used to organize IP route tables into an efficiently searchable structure) in packet-processing applications exhibit considerable locality of access. Because of the inherent differences in their

locality properties, these different types of data structures often interfere with each other and thereby lower the effective hit rate of the memory subsystem. Hence, a single hardware-based mechanism for managing the cache hierarchy is ineffective for packet-processing applications. In our system, the π -compiler will use the access frequency, object size, and other object and data locality properties collected or derived by the profiler to determine an appropriate memory hierarchy management policy. This may involve allocating data structures at different levels of the memory hierarchy, distributing data structures across memory banks at the same level of the hierarchy for load balancing [9], and using controlled pre-fetching.

The π -compiler represents an entire packet-processing application as a PPF graph, where each node represents a PPF and each edge represents a communication channel. The inputs to the aggregation and memory mapping functions in the π -compiler are: the PPF graph, a high-level representation of code sequences, and the symbol tables. We extend an existing framework of inter-procedural analysis to perform a set of analyses across functions to characterize objects, computation, communication, and instruction stores. These analyses provide essential information to each step of aggregate clustering, allocation of various types of resources, placement of aggregates to MEs, and mapping of data structures to memory hierarchy. These decisions not only determine the quality of code produced by the rest of the compiler but could also influence the adaptation performed by RTS. Annotations on how aggregates are placed and replicated as modeled by the π -compiler are passed to RTS to allow efficient mapping, while RTS is free to adapt resource allocation and mapping by observing the system load.

Given a set of aggregates, an aggregate construction phase in the π -compiler generates the necessary glue code to tie together the PPFs within an aggregate as well as across aggregates. For instance, since each aggregate executes continuously and processes a stream of packets, the aggregate constructor maps each aggregate to a thread and introduces the code necessary to dispatch packets to the appropriate PPF upon their arrival. Similarly, if an aggregate can receive packets from multiple aggregates, the construction phase incorporates the appropriate scheduler to ensure that different types of packets don't interfere with each other's performance.

The whole compilation infrastructure of Shangri-La incorporates an iterative compilation feature. This provides the system with opportunities to refine the decisions made in an earlier compilation and with a higher chance to approach an optimal solution. In an iterative compilation framework, it is important to identify the type of events and statistics to be monitored

and fed back with proper mapping and to design a robust feedback loop to guide subsequent compilations toward a better solution. However, an iterative compilation framework still requires high-quality cost models and heuristics in the compiler to guide the optimizations during each iteration of compilation. An iterative framework with sloppy heuristics may never converge to an optimal solution.

4.2.3. Aggregate Compiler. The aggregate compiler receives from the π -compiler a set of aggregates, their mappings to hardware processing cores, as well as a policy for managing the memory hierarchy. The aggregate compiler performs both machine dependent and machine independent optimizations with the objectives of maximizing performance and throughput of each aggregate. For each aggregate mapped to a target processing core, the aggregate compiler produces the output in the form of assembly or object code along with a set of annotations used by the RTS. It is common for an NP to contain multiple types of processing cores. The aggregate compiler needs to generate multiple versions of aggregate code in different ISAs for each aggregate that may be mapped to multiple types of processing cores.

Many compiler analysis and optimization techniques developed for general-purpose compilation are applicable to compiling packet-processing applications on NPs. For example, inter-procedural analysis performs various types of analysis across functions to provide sharpened analysis results to many subsequent optimizations. Memory optimizations, such as placing data prefetches and reordering data layout or object fields, can hide the latency in memory accesses or improve the spatial locality of accessed data items. Full or partial redundancy elimination can remove redundant computation and memory references appearing on all or some execution paths.

Most of the Shangri-La components introduced thus far are independent of target hardware. However, the code generation (CG) component, where native code is generated and many processor dependent optimizations are performed, is expected to vary significantly from one processing core to another, since the different types of processing cores on each NP often have dramatically different ISAs and micro-architectural implementations. On the IXP NPs, the Intel XScale[®] processor is a general purpose processing core and has been adopted in the designs of various embedded systems. Hence, we leverage existing technologies and tools to generate code for the aggregate mapped to the Intel XScale[®] processor. On the other hand, new technologies are being developed to optimize the aggregates on MEs because they have a major impact on the overall throughput of packet-processing applications running on IXP NPs.

Furthermore, the design of the MEs, which target the efficient processing of packets, poses a number of challenges to compilation. We discuss several of these challenges below:

1. Fragmented memory hierarchy: On the MEs, the memory hierarchy is divided into a number of levels including local memory (LM), scratchpad, SRAM, and DRAM. LM is local to each ME, whereas the rest are shared by all MEs and the Intel XScale[®] processor. Unlike a traditional cache structure managed by the hardware, the address spaces on different levels of the memory hierarchy are distinct, and require different types of instructions and register classes to access different memory levels. No operating system or support of a single virtual address space exists on the MEs.

As in the majority of high-level programming languages, procedure invocation is supported in Baker for the sake of programmability and modularity. A call stack is a typical means to support general procedure invocation. However, implementing a stack on fragmented memory hierarchy is non-trivial since the compiler needs to track whether the stack has outgrown the allowed space at a particular memory level. Generating runtime checks to select among multiple code sequences for the different memory levels is inefficient in both performance and code size. We are investigating an inter-procedural stack management framework to allow a statically determined mapping from a stack location to a particular memory level.

Another issue due to fragmented memory hierarchy is on pointer dereferences. If a pointer may point to different memory levels, there is no easy way to generate efficient instructions to dereference the pointer. One way to address this issue is to force the objects that may be pointed to by the same pointers to be allocated onto the same memory level. A congruence-based points-to analysis may help partition objects into congruence classes to map the object allocation to different levels of the memory hierarchy.

2. Register aggregates and partitioned register classes: The ME ISA allows a number of registers to be used implicitly in an instruction through register aggregates or indexed registers. This adds complexities to several phases in the CG. The IR in CG has to be capable of representing the implicit registers while maintaining both time and space efficiency. Instruction scheduling needs to capture the

dependencies among all explicit and implicit operands while reordering instructions. Register allocation needs to perform liveness data flow analysis and color those register operands inferred from a compact representation.

The registers on a ME are divided into a number of classes. For example, each level of memory hierarchy has its dedicated register classes to move data in and out of its memory. The class of general purpose registers on a ME is further divided into bank A and bank B. Each instruction often has constraints on the legal combinations of register classes or banks for its operands. This poses a challenge on allocating proper registers to each live variable while minimizing the number of moves among different register classes and banks.

3. **Architecture Irregularity:** In addition to the constraints on allowed register classes and banks for each instruction, there are other irregularities in the ME ISA. For example, the registers on ME can be accessed using a context-sensitive mode or an absolute mode, where the former treats each register local to a thread and the latter provides a means to access any register across all threads on an ME. Although the absolute mode is an effective way to communicate among different threads on an ME, not all instructions can refer to registers using the absolute mode. Hence, this limits the use of absolute mode.

Figure 4 shows the block diagram of the ME code generator. The phases and their ordering are similar to other code generators, but the ME CG has to answer the design

challenges mentioned above. The code selection phase translates from a high-level IR to CG IR, which has a typical one-to-one mapping to ME instructions. The memory optimizations generate efficient code sequences to access memory, e.g. to combine multiple loads that access adjacent data fields into one load with a register aggregate target. The Control Flow Optimization (CFO) and Extended Basic-Block Optimization (EBO) phases perform simplification on control flow and classical optimizations on the scope of extended basic blocks, respectively. Loop optimizations, such as loop unrolling, focus on loop structures. The global scheduling may reorder instructions to reduce critical schedule lengths across basic blocks. Register allocation colors all virtual registers with proper register classes and banks. If there are register spills, local instruction scheduling is invoked to reschedule the instructions in the affected basic blocks. The code emission phase emits assembly code as well as the annotations passed to RTS.

The system model abstracts architectural and micro-architectural details, such as latency, instruction opcode, the number of MEs, the size of each level of memory hierarchy, etc. into a separate module. Such information is used throughout the entire compiler. The code size guard regularly checks the current usage of instruction store. When the current code size approaches its physical limit, many optimizations that may increase code size, such as loop unrolling and scheduling with code duplication, are restricted, while optimizations that reduce code size, such as redundancy elimination, are made more aggressive. The heuristics to make intelligent trade-off between code size and performance remain an important subject to explore.

4.3. Run-time System

Although it may be possible to acquire packet traces that are broadly

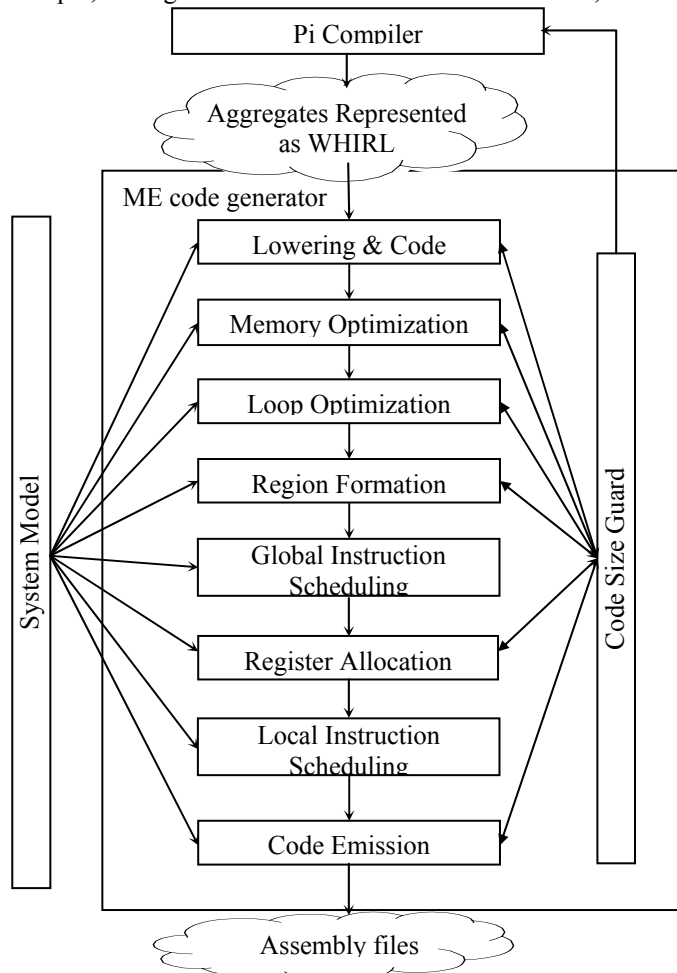


Figure 4: Microengine code generator phases

representative of the workloads that are presented to a packet-processing application, it is likely that these traces will differ from the workloads presented to the application when it is deployed in the field. After all, network applications are deployed in extremely diverse environments. A wireless access point may be deployed in a small business environment with file and printer sharing being the dominant application. The same model of wireless access point could also be deployed in a residential environment where web surfing may be the dominant application. It would be impossible to come up with a set of packet traces that would accurately represent both of these environments.

Even if the packet traces used for profile-driven compilation are accurate with respect to the actual workload, such workloads are rarely static over time. Network traffic characteristics change from hour to hour, minute to minute, and second to second. If a packet-processing application is to keep up with performance demands when confronted with workloads that differ from those used during profiling, it may need to adapt the allocation of resources to software constructs at run time.

Besides performance, other benefits may also come from the ability to adapt. For example, if a network device could dynamically power off or reduce the clock frequency of unused or underused hardware resources, the average power consumption and heat dissipation of the device could be reduced. Also, in an environment where those with malicious intent may try to deny service to legitimate users by hoarding critical resources, the ability to adapt resource allocations at run time may allow a device to prevent such denials of service.

Supporting run-time adaptation requires a run-time system with two important properties: resource-awareness and dynamic resource adaptation.

In this context, resource-awareness means that the run-time system must know which resources are being used by the application, and how effectively they are being used. The RTS provides resource-awareness through a *resource abstraction layer (RAL)* that is linked to the application code at run-time. For each resource type, the RAL defines an interface and includes one or more implementations of the interface. For instance, the RAL supports a *queue* resource type with enqueue and dequeue methods. A RAL implementation may support multiple queue implementations (e.g., on the IXP2800 network processor, queues can be implemented using

next-neighbor registers, on-chip scratch memory, off-chip SRAM or DRAM). This decomposition allows the run-time system to select the most-suited implementation based on the mapping of aggregates onto processors (e.g., for aggregates mapped to neighboring microengines, the run-time system selects the queue implementation that uses next-neighbor registers; while for other cases, the run-time system selects the scratch memory or SRAM/DRAM implementations). Since the RAL interfaces are linked at run time, the resource allocations can be modified at run time without recompiling the code.

Designing an abstraction layer for a network device is a challenge for two reasons. First, although the packet-processing domain is a narrower domain than general computing, different applications found in this domain still require quite varied hardware services. For example, an Ethernet switch may require the ability to compute

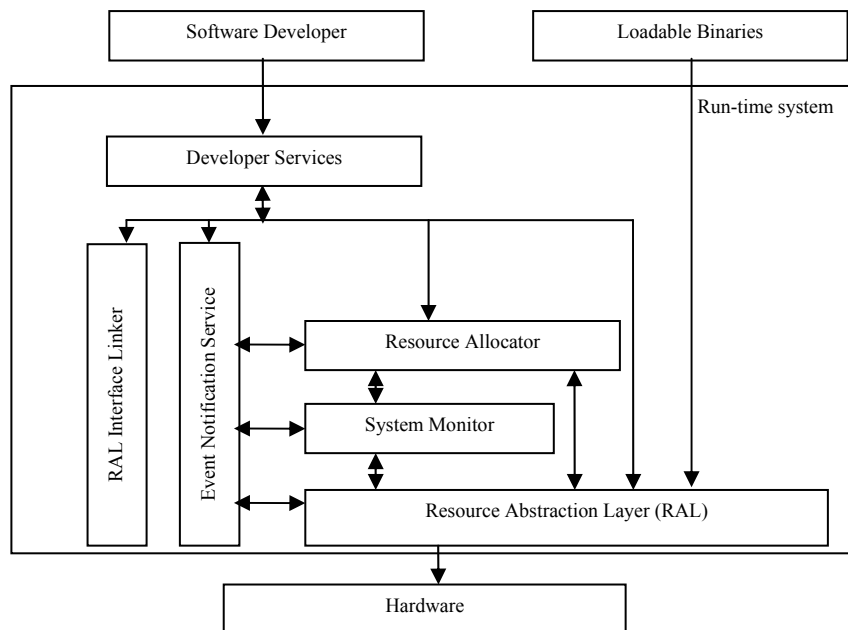


Figure 5: Run-time system decomposition

hashes very efficiently, whereas a VPN offload device may require the ability to perform encryption and decryption very efficiently. Second, the spectrum of hardware platforms used in these applications is also broad, many requiring different methods of performing the same computational tasks.

To support dynamic resource adaptation, the run-time system monitors system performance and traffic conditions and adapts resource allocations across aggregates. The system monitor allows users or higher layers of the system to define triggers based on predicates defined over run-time measures. At run time, the monitor receives—either using a polling interface or through

asynchronous event notifications—system statistics (e.g., queue lengths) from the resources, evaluates the predicates, and generates events if any of the predicates are satisfied. Based on the performance requirement of the application and the current traffic conditions, the resource allocator determines and enforces the new allocation.

The design and implementation of such a distributed monitoring and resource adaptation framework is challenging for two reasons. First, the performance-sensitive nature of packet-processing applications means the monitoring infrastructure as well as the run-time system should impose as little overhead as possible. Second, the inherent heterogeneity and widely-varying capabilities of the resources available in NP-based systems makes the task of determining an optimal mapping of pipeline stages to processor resources complex.

When a packet-processing pipeline is mapped onto multiple processors, a packet migrates from one processor to another, with each processor providing a portion of the total service requested by the packet. Further, each processor may simultaneously service packets belonging to multiple flows. Providing performance guarantees in such distributed, shared environments requires sophisticated techniques for coordination and scheduling of multiple resources [13][36]. These techniques determine (1) the mapping of flows to resource instances, in the event that multiple resources with the same functional capability but with different performance characteristics are available in the packet-processing system; (2) the relative priority for processing packets from different flows at each resource; and (3) guidelines for gracefully degrading system performance (or at least the performance observed by some flows) in the presence of persistent overload. Such a resource allocation framework is essential to construct packet-processing systems that are robust to denial-of-service attacks.

In addition to supporting run-time adaptation, the RTS supports features necessary for running and debugging code on an embedded network device. These features include the ability to load code, run code and debug code.

The design decomposition of the RTS is shown in Figure 5.

The literature contains operating systems designs for multi-processor systems [22][34][43], real-time systems [17][33][37], extensible systems [10][14][16][17], and pipelined systems [25][30][41]. We plan to leverage many concepts from the prior work. The realization of these concepts in packet-processing systems with stringent resource and timeliness constraints poses several problems that we plan to explore.

5. Concluding Remarks

The programming environments—languages, compilers, and run-time systems—for NPs are in their infancy. At the same time, NPs represent a much larger trend in the processor industry: multi-core, light-weight threaded architectures design for throughput-driven applications. Once this trend hits the mainstream programming marketplace, the need for a programming environment that is as easy to use as the programming environments for today's workstations and servers will become universally important to programmers. The Shangri-La architecture represents a complete programming environment for the domain of packet processing on multi-core, light-weight threaded architectures in general, and NPs in the specific. Shangri-La encompasses: (1) A language that exposes domain constructs instead of hardware constructs, keeping the programmer and code separate from architectural details. (2) A sophisticated compiler complex that uses profile information to guide the mapping of code to processors and data structures to memory automatically. (3) A run-time system to ensure maximum performance benefits in the face of fluctuating traffic conditions—both natural and malicious.

We are currently working on two major tasks: creating a prototype implementation of the proposed architecture, and researching the more difficult questions that we will face as development proceeds.

The prototype system builds on the Open Research Compiler infrastructure and targets the Intel IXP2400 network processor. It includes implementations of each component shown in Figure 1, with simplified algorithms in some of the components. This prototype system will provide a platform for further research and development, and should be available in the first half of 2004.

Our current research tasks cover a wide spectrum, and our progress includes published work in language design [44] and run-time adaptation of resource allocations [45].

6. References

- [1] *AMCC's nP7xxx series of Network Processors*, <http://www.mmcnetworks.com/solutions/>.
- [2] *Agere's PayloadPlus Family of Network Processors*, http://www.agere.com/enterprise_metro_access/network_processors.html.
- [3] *IBM PowerNP Network Processors* http://www-3.ibm.com/chips/techlib/techlib.nsf/products/IBM_PowerNP_NP4GS3.

- [4] *iFlow Family of Processors*, Silicon Access, <http://www.siliconaccess.com>.
- [5] *Intel IXP family of Network Processors*, www.intel.com/design/network/products/npfamily/index.htm.
- [6] *The Motorola CPort family of Network Processors*, <http://www.motorola.com/webapp/sps/site/taxonomy.jsp?nodeId=01M994862703126>.
- [7] *Agere Functional Programming Language*, www.agere.com/enterprise_metro_access/docs/PB02014.pdf.
- [8] *TejaNP*: A Software Platform for Network Processors*, www.teja.com.
- [9] R. Barua, *Maps: A Compiler-Managed Memory System for Software Exposed Architectures*, PhD dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 2000.
- [10] B. Bershad, S. Savage, P. Pardyak, E.G. Sirer, D. Becker, M. Fiuczynski, C. Chambers and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of ACM Symposium on Operating Systems Principles*. December 1995.
- [11] S. Bokhari, *Assignment Problems in Parallel and Distributed Computing*, Kluwer Academic Publishers, 1987.
- [12] Campbell, S. Chou, M. Kounavis, V. Stachtos, and J. Vincente, "NetBind: A Binding Tool for Constructing Data Paths in Network Processor based Routers", in *Proceedings of Fifth International Conference on Open Architectures and Network Programming (OPENARCH'02)*, New York, NY, June 2002
http://www.comet.columbia.edu/~mk/papers/openarch02_netbind.pdf.
- [13] Chandra, M. Adler, P. Goyal and P. Shenoy, *Surplus Fair Scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors*, In *Proceedings of 4th Symposium on Operating System Design and Implementation (OSDI) 2000*.
- [14] G. Coulson and G.S. Blair. *Architectural Principles and Techniques for Distributed Multimedia Application Support in Operating Systems*. *ACM Operating Systems Review*. October 1995.
- [15] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner, *Router Plugins, A Software Architecture for Next Generation Routers*, in *Proceedings of SIGCOMM'98*, 1998.
- [16] D.R. Engler and M.F. Kaashoek. *Exokernel: an operating system architecture for application-level resource management*. In *Proceedings of ACM Symposium on Operating Systems Principles*, December 1995.
- [17] W.M. Gentleman, S.A. MacKay, D.A. Stewart and M. Wein. *An Introduction to the Harmony Realtime Operating System*. In the *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems*, Summer 1988.
- [18] L. George and M. Blume, *Taming the IXP Network Processor*, To appear in *Proceedings of PLDI'03*, San Diego, California, 2003.
- [19] Y. Gottlieb and L. Peterson, *A Comparative Study of Extensible Routers*. In *Proceedings of OpenArch '02*, 2002.
- [20] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, *A Stream Compiler for Communication-Exposed Architectures*, in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October, 2002.
- [21] R. Gupta, S. Pande, K. Psarris, and V. Sarkar, *Compilation techniques for parallel systems*, *Parallel Computing*, Vol. 25, No. 13-14, Pages 1741-1783, 1999.
<http://citeseer.nj.nec.com/150142.html>.
- [22] G.C. Hunt and M.L. Scott. *The Coign Distributed Partitioning System*. In *Proceedings of 3rd Symposium on Operating Systems Design and Implementation*. February 1999.
- [23] S. Karlin and L. Peterson, *VERA: An Extensible Router Architecture*, *Computer Networks*, 2002.
- [24] R. Keller, L. Ruf, A. Guindehi, B. Plattner, *PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing*, in *Proceedings of Fourth Annual International Working Conference on Active Networks*, 2002.
- [25] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, *The Click Modular Router*, *Transactions on Computer Systems*, 2000.
- [26] M. E. Kounavis, A. T. Campbell, S. Chou, F. Modoux, J. Vicente, and H. Zhang, *The Genesis Kernel: A Programming System for Spawning Network Architectures*, *IEEE Journal on Selected Areas in Communications (JSAC), Special Issue on Active and Programmable Networks*, Vol. 19, No. 3, pp. 49-73, March, 2001 <http://comet.ctr.columbia.edu/genesis>.
- [27] F. Kuhns, J. DeHart, A. Kantawala, R. Keller, J. Lockwood, P. Pappu, D. Richards, D. Taylor, J. Parwatikar, E. Spitznagel, J. Turner, K. Wong, *Design of a High Performance Dynamically Extensible Router*, in *Proceedings of DARPA Active Networks Conference and Exposition '02*, 2002.
- [28] D. Mosberger. *Scout: A Path-based Operating System*. PhD Dissertation, Department of Computer Science, University of Arizona, July 1997.
- [29] S. Orlando and R. Prego, *Scheduling Data-Parallel Computations on Heterogeneous and Time-Shared Environments*, *Proceedings of European Conference on Parallel Processing*, Pages 356-366, 1998
<http://citeseer.nj.nec.com/orlando97scheduling.html>.

- [30] M. Philippsen, A Survey of Concurrent Object-oriented Languages, in *Concurrency: Practice and Experience*, volume 12 Pages 917-980, 2000.
- [31] V. Sarkar, *Partitioning and scheduling Parallel Programs for multiprocessors*. The MIT Press. 1989.
- [32] K. Sevcik, Characterizations of Parallelism and their use in Scheduling, in *Proceedings of 1989 ACM SIGMETRICS Conference*, pages 171-180, 1989.
- [33] K.G. Shin, D.D. Kandlur, D.L. Kiskis, P.S. Dodd, H.A. Rosenberg, A. Indiresan. A Distributed Real-Time Operating System. *IEEE Software*, September 1992.
- [34] W. Shu. Chare Kernel: A Runtime Support System for Parallel Computations. *Journal of Parallel and Distributed Computing*. Volume 11, number 3, 1991.
- [35] S. Silva, Y. Yemini, and D. Florissi, "The NetScript Active Packet processing system", *IEEE Journal on Selected Areas in Communications (JSAC)*, Vol. 19, No. 3, Pages 538-551, March 2001.
- [36] Srinivasan, J. Anderson, Efficient Scheduling of soft real-time applications on multiprocessor, Submitted to the *23rd International Conference on Distributed Computing Systems*, 2003.
- [37] D.B. Stewart and P.K. Khosla. Chimera Real-Time Operating System. <http://www-2.cs.cmu.edu/~aml/chimera/chimera.html>.
- [38] J. Subhlok and G. Vondran, Optimal Mapping of Sequence of Data Parallel Tasks, in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
- [39] J. Subhlok, J. M. Stichnoth, D. R. O'Hallaron and T. Gross, Exploiting Task and Data Parallelism on a Multicomputer, *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.
- [40] R. Subrahmanian, I. D. Scherson, V. L. M. Reis and L. M. Campos, Scheduling Computationally Intensive Data Parallel Programs, <http://citeseer.nj.nec.com/256010.html>.
- [41] M. Welsh, D. Culler, and E. Brewer, SEDA: An Architecture for Well-conditioned, Scalable Internet Services, In *Proceedings of Symposium on Operating Systems Principles (SOSP-18)*, October 2001.
- [42] T. Yang and A. Gerasoulis, PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors, Proceedings of the *1992 ACM International Conference on Supercomputing*, Washington DC, 1992.
- [43] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black and R. Baron. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. In *Proceedings of 11th Symposium on Operating Systems Principles*, November 1987.
- [44] S. Goglin, D. Hooper, A. Kumar, R. Yavatkar, Advanced Software Framework, Tools, and Languages for the IXP Family. *Intel Technology Journal*. <http://developer.intel.com/technology/itj/2003/volume07issue04/>, November 2003.
- [45] R. Kokku, T. Riche, A. Kunze, J. Mudigonda, J. Jason, H. Vin, A Case for Run-time Adaptation in Packet Processing Systems, in *Proceedings of the Second Workshop on Hot Topics in Networks*, November 20, 2003.
- * Other brands and names are the property of their respective owners