



# Intel<sup>®</sup> Technology Journal

Communications Processing

**Advanced Software Framework, Tools,  
and Languages for the IXP Family**

# Advanced Software Framework, Tools, and Languages for the IXP Family

Stephen D. Goglin, Corporate Technology Group, Intel Corporation  
Donald Hooper, Intel Communications Group, Intel Corporation  
Alok Kumar, Intel Communications Group, Intel Corporation  
Raj Yavatkar, Intel Communications Group, Intel Corporation

## ABSTRACT

The IXP network processor [1] architecture is designed to process packets at high rates where inter-arrival times between packets are sometimes less than the individual memory access latencies. To keep up with such high arrival rates, IXP presents a novel architecture consisting of a collection of multithreaded, multiprocessors with an explicit memory hierarchy. Such an architecture poses interesting challenges when an application developer wishes to develop software for a wide range of services that can easily be ported from one IXP processor to another. To make it easy to develop portable, modular software on IXPs, we currently provide a combination of development tools and a software framework consisting of libraries, run-time infrastructure, and APIs to facilitate the creation of application building blocks. This paper describes the future directions in IXP programming tools, software frameworks, languages, and compilers.

We first identify the limitations of the current IXA software framework [3], describe extensions to remove such limitations and discuss how these extensions also help in the development process.

Secondly, we provide an overview of the Developer Workbench tool suite and its features, which provide significant advancements over traditional development environments. We highlight features that enable architectural task modeling, allow debugging using bandwidth analysis, thread history, and operand tracing, and provide a novel packet-centric analysis based on extensible protocol packet generation, packet status, and a graphical view of the packet dataflow.

And finally, to move the developers beyond a chip-centric view of programming, we also investigate domain-specific languages and adaptable run-time environments. We describe and justify one such research project under development, called *Baker*, to develop a domain-specific language for network processing.

## INTRODUCTION

In this paper, we discuss three new ways that the software development infrastructure for programming network processors is being improved: the existing IXA software framework is being extended, new features are being added to the Developer Workbench, and research in developing a domain-specific language and compiler for packet processing is being pursued. We start with a brief introduction to each topic.

### Framework Extensions

Firstly, network processors are being targeted at a wide range of applications with varying packet-processing and throughput requirements. The programmability and flexibility of a network processor make it suitable for applications ranging from Voice over IP to content-based routing with data rates spanning OC-3 to OC-192. In such an environment, the investment made in software development by equipment manufacturers is increasingly significant. Preserving this investment and leveraging it across multiple projects are key considerations when choosing a network processor. The IXA software framework [3] provides the necessary software infrastructure to help develop modular, reusable software building blocks for network processors of the IXP family.

The IXA Portability Framework is now widely adopted for software development on the IXP family. However, the framework is unable to fully support some classes of applications. These applications require support for specific features such as multicast forwarding, handling packet fragmentation, and running at low rates (OC-3 to OC-12). The IXA framework, therefore, must be extended so that it can support all applications efficiently and easily.

The main shortcomings of the existing software framework are (i) lack of support for the creation of multiple packets from a packet in the packet processing stage, (ii) lack of support for efficient manipulation of

packet data buffers when they are sent to multiple interfaces, and (iii) the inability of microblocks that were built independently to fit together when run on different (not the same) threads within a microengine.

## Development Tools

Secondly, with each new generation of IXP network processors, the hardware architecture becomes more complex: there are additional microengines, memory controllers, and hardware acceleration features. Therefore, the need for advanced tools that simplify the design and debugging processes is greater today than ever before.

In this paper, we describe the current method of debugging IXP applications, then introduce tool enhancements that enable a new, packet-centric, development methodology. We discuss new architectural planning, packet list, history event capture, conditional breakpoint, and instruction operand tracing features.

## Next-Generation Compilers

Finally, we are conducting research on a domain-specific programming language, called *Baker*, for developing applications using network processors. Because of the unique hardware features and structure of the network processor platform, conventional, procedural languages are forced to expose many of the details of the hardware to the developer. Exposing low-level details may allow a developer to produce high-performance object code but the development is complex requiring unique skills, and porting an application (or its parts) from one IXP to another is difficult. The Baker language exposes the domain-specific features of packet processing rather than the hardware-specific features. As a result, the code is easier to write and port, and, at the same time, the compiler has sufficient information to be able to produce efficient binaries.

We assume that the reader is familiar with the Intel Network Processor Family [1], previous versions (SDK 3.5 or earlier) of the IXA Software Development Toolkit [2] and the IXA portability framework [3].

## FRAMEWORK EXTENSIONS

There are two problems in the existing IXA software framework [3]:

1. The IXA software framework provides the software infrastructure to help develop modular, reusable software building blocks for IXPs. It assumes the model in which a packet traverses from one microblock to another, where each microblock [3] can modify the packet content or the state associated

with the packet. In this model, it is not possible to generate multiple packets from a single packet in a microblock. The generation of multiple packets from a single packet in a microblock is required for the support of packet multicast and IP packet fragmentation.

2. The IXA software framework provides the flexibility to port an application on IXPs ranging from the high-end IXPs running at very high data rates to the low-end IXPs running at lower data rates, without any changes to the microblocks. The low-end IXPs have a fewer number of microengines compared to the high-end IXPs. Therefore, several microblocks that may run on separate microengines on a high-end IXP may have to be combined to run on a single microengine when ported to a low-end IXP. However, if these microblocks use microengine *Content Addressable Memory* (CAMs) [1] for achieving efficient mutual exclusion and synchronization, then combining microblocks while they run on different threads in a single microengine is not possible.

We now describe the extensions designed to solve the above two problems.

## Support for Multicast and Packet Fragmentation Features

### Problem Definition

In the current framework, packet data are stored in DRAM, and every packet has a one-to-one relationship with the meta-data associated with it that is stored in SRAM. Information about a packet is passed across microblocks using a fixed meta-data structure. This framework does not apply in a case where a multicast packet is processed requiring multiple copies of an incoming packet to be sent over different outgoing links. The existing framework requires generating a new packet buffer for each multicast copy. Copying of packet data for multicast is undesirable because it adds to memory bandwidth and latency requirements. Similarly, to handle IP fragmentation, an incoming packet must be divided into parts that are individually sent out as separate packets. The current framework infrastructure would again require copying of data buffers, something that is equally undesirable. Our goal is to extend the framework infrastructure to support new functionality without requiring additional copying of data buffers.

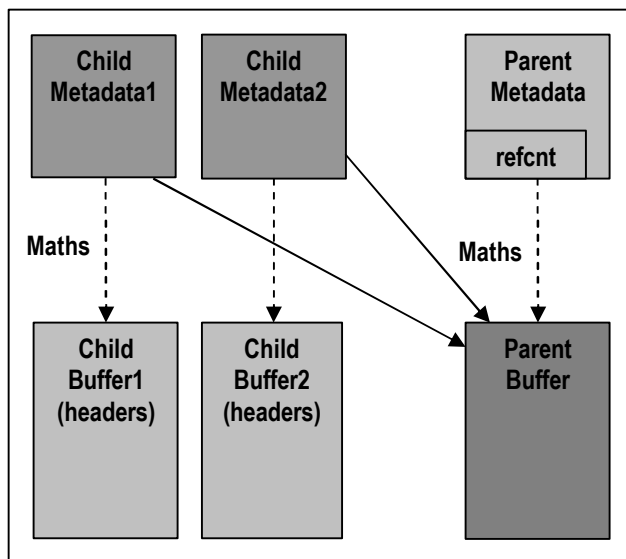
To be precise, we must meet the following requirements to achieve efficient support for multicast and packet fragmentation features:

- A packet data buffer should not have to be copied. Only the packet headers, which are different for all the copies, should be copied when a packet is multicast.
- Only minimal changes should be made to the packet-processing microblocks. Packet-processing microblocks should not have to distinguish among unicast, multicast, or fragmented packets in terms of buffer and meta-data management except when a microblock needs to process the multicast or fragmented packets differently from the unicast packets.
- There should be no performance impact for unicast or non-fragmented traffic. Support for multicast and fragments should not impact the performance of unicast or other traffic requiring no fragmentation.

### Extensions to the Infrastructure

When a multicast packet is to be sent out with multiple copies, the only change in each copy of the packet is to the modified header; the payload is left unchanged. The same thing is true when an incoming packet is fragmented and each piece is sent out as a separate packet. Therefore, we extend the framework by adding the concept of *child* and *parent* buffers with the explicit goal of avoiding buffer copies, but still being able to transmit a distinct copy of each packet. Each multicast packet data buffer is a parent buffer. All packets generated from the original packet have a child buffer, containing a new packet header and a pointer to the original parent data buffer.

Zero copy of packet data, except packet headers that are distinct for each packet copy, can be achieved by having the parent buffer that has data buffer and the child buffers, storing only the packet headers, for every copy of the multicast/fragmented packet. The child buffers point to the parent buffer. The child buffers store modified headers for every multicast/fragmented packet. The parent buffer stores the common data buffer for all copies of the multicast packet. The parent buffer contains a *refcnt* (reference count) representing the number of child buffers pointing to it. [Figure 1](#) shows the structure of a multicast/fragmented packet.



**Figure 1: Data layout for a multicast/fragmented packet**

Every buffer has a corresponding meta-data. In the current framework, every unicast packet has a corresponding meta-data. We propose that every copy of the multicast packet also has a corresponding *child meta-data*. After a packet is multicast, the *child meta-data* is used by all packet processing blocks and is used for queuing of the packet. This extension needs to be implemented in the architecture in such a way that a child buffer for a multicast packet would not look any different than a parent buffer for a unicast packet. This is achieved by making the child meta-data the same as the parent meta-data except for the fields that are not used by the packet processing blocks. Since most of the packet processing blocks work only on packet headers, they can use a child buffer in the same way as they use a packet buffer for a unicast packet. Thus, there are no changes to the previously written packet processing blocks: multicast or fragmented packets are treated the same as unicast, non-fragmented packets.

As a result of these changes, the *Packet Transmit* block of the packet processing pipeline must change. Before freeing up a buffer, the block first reads the *refcnt* from the parent meta-data, decrements it, and frees the parent buffer only if the *refcnt* is zero. The reading of the *refcnt* adds one extra dependency to the packet transmit code. However, since all packets are stored in the local memory and then processed later, the reading and checking of the *refcnt* can easily be overlapped with the existing memory accesses, after storing a packet in the local memory. This ensures minimal changes to the packet transmit block and makes it possible to meet the line rate in all the cases.

## CAM Sharing Extension

### Problem Definition

Mutual exclusion is a well-known problem for multithreaded software. For example, in a network processor, several threads may be accessing the same memory location at the same time and updating the value at that location. These accesses and modification to the same location have to be mutually excluded to maintain the correct semantics. Therefore, a network processor needs to have a mechanism to support *mutual exclusion* across these threads. *Folding* provides a mechanism to support this in the IXP2xxx family of network processors. *Folding* uses the microengine CAM to provide mutual exclusion on a location across threads running in a microengine without throttling the performance. *Folding* is briefly described in the next few paragraphs.

A *critical section* is a part of software that involves a sequence of *read* – *modify* – *write* operations on a memory location. The *read* operation reads the value of the memory location into the microengine. The *modify* operation is performed after the *read* operation is complete and modifies the data read from the memory location. The *write* operation writes back the modified data to the memory location. Any other thread can perform the *read* operation on the same location only when the *write* operation of the previous thread is complete. Due to the non-parallel nature of “critical section” code, it becomes a challenge to hide the latencies involved in reading and writing the critical data from/to the memory. *Folding* solves this problem by making sure that the critical data are in the microengine when the second thread needs them. The operations *read* – *modify* – *write* are modified in *folding* to allow parallelism. Threads perform all these operations in strict order. For example, thread 0 performs the *read* operation and sends a signal to thread 1 to perform the *read* operation and so on. Thread 0 starts the *modify* operation after all the threads have completed the *read* operation. Similarly, *modify* and *write* operations are performed in strict order.

During *read* operation, a thread does a CAM lookup to determine if the data from the desired memory location are already in the microengine. In the case of a *CAM miss* (data are not in the microengine), the thread issues a read for the data from the memory and updates a free CAM entry to reflect that the data will be available in the microengine. In the case of a *CAM hit* (data are already in the microengine), it does not issue the read from the memory. When every thread is done with the *read* operation, the first thread modifies the corresponding data in the microengine and stores the modified data in the microengine. It then signals the next thread to start

the *modify* phase. This ensures that the next threads use the data that have already been modified by the first thread. Hence, only one thread modifies the data at one time and not more than one thread works on the same data at any one time. This ensures mutual exclusion on the data across all threads. A thread performs *write* after it is done with *modify*, only if it is the last thread working on the data. From the above description we see that even for a *critical section*, *folding* allows *reads* and *writes* to be performed in parallel, while allowing mutual exclusion.

Least Recently Used (LRU) is a standard cache replacement policy. In traditional caches that use LRU, when a new entry is to be fetched in the cache, then the entry in the cache that has not been used for the longest period of time is expunged. This scheme of *replacing the cache entry that has not been used for the longest period* is called LRU. In the state of the art, we use *folding* in conjunction with the LRU method to solve the mutual exclusion problem. When we have a CAM miss, we find a free CAM entry using the LRU policy implemented by the CAM hardware to store the new data that are going to be read. We use LRU as a replacement policy in *folding* because LRU is supported by the CAM hardware and the LRU policy meets the cache replacement requirement of *folding* as long as only one microblock is using *folding* at any time within the entire microengine (across all threads).

However, with the addition of more functionality, especially on medium to low-end performance NPs, more than one microblock runs on different threads in a microengine. These microblocks run independently on different threads and are not synchronized across each other. More than one of these microblocks may need to use the *folding* technique and, therefore, simultaneously use the CAM for the purpose. We show how a single LRU policy across the entire CAM supported by the hardware is not sufficient when the CAM is shared by two or more microblocks running independently in parallel. The single LRU policy across all the CAM entries has the following two problems.

1. *LRU might return a CAM entry that is in use by another microblock.* Suppose, there are two microblocks A and B running independently on a single microengine. Assume microblock A is running on the first four threads and microblock B is running on the last four threads with no synchronization across each other. When needed, the hardware LRU would give the single LRU entry among all the CAM entries in the microengine, even though that entry might be in use by a different microblock running on another thread. We cannot

protect the two different CAM entries, one used by microblock A and the other by microblock B. For example, microblock A may execute its *read – modify – write sequence* for an arbitrarily longer period compared to the one executed by microblock B. Therefore, a thread running microblock A may need to keep a CAM entry, *e1*, much longer than the CAM entries used by the threads running microblock B. Therefore, while entry *e1* is used by a thread running microblock A, threads running microblock B can run much faster and use every other entry, making all of them more recently used compared to entry *e1*. This makes *e1* the LRU even though it is still in use. As a result, the hardware victimizes the entry *e1* and makes it available to a thread running microblock B even though the original thread is still using *e1*. This leads to unpredictable and undesirable results.

2. *A thread running a microblock does not know how to evict an entry that has the data of other microblocks.* Suppose, we have two microblocks A and B running independently and microblock B gets a free entry that has some data previously used by microblock A. Microblock B has to flush the data stored in that entry before it can use the entry for its own data. Microblock B has no idea about the data structures of microblock A and their actual location in the memory. Therefore, it is impossible for microblock B to flush the old data.

In addition to the problem of using a single hardware LRU when two independent microblocks share the CAM in a microengine, the microblocks also need their keys to be unique to ensure that any key of microblock A does not match a key of microblock B. In the following section, we propose a solution to the two problems we have outlined.

### Extensions to Support CAM Sharing

A solution to the problem of ensuring uniqueness of keys across microblocks is to assign keys so that a key of a microblock does not match a CAM entry that stores the same key of another microblock. Our proposal is to append a unique microblock ID to each key so that keys from different microblocks can never be the same. We propose that the unique ID be a four-bit constant assigned to each microblock by the dispatch loop [3]. This ID is assigned to the most significant four bits of a key. That leaves 28 bits for microblocks to use to perform CAM searches. The number of bits required for the unique ID can be changed without affecting any other component of this proposal. However, we believe that it is highly unlikely that more than 16 (maximum number represented using 4 bits) microblocks using CAMs

independently will run on a single microengine in the future. We also believe that a 28 bit-wide key is sufficient to perform searches.

The solution to the second problem is to always yield a free CAM entry when there is a CAM miss. It should be noted that in *folding*, you do not need the least recently used entry on a CAM miss; you only need one free CAM entry that no other thread is using<sup>1</sup>. A free entry can be identified by maintaining a simple bit vector of free CAM entries. We propose use of the following data structures:

1. For every CAM entry, we maintain a reference count representing the number of threads using the CAM entry. The reference count for a CAM entry is incremented when a thread starts using the CAM entry. The reference count is decremented when the thread exits using the CAM entry.
2. We maintain a bit vector, *free\_CAM\_vector*, with each bit corresponding to a CAM entry. A bit corresponding to a CAM entry is set if and only if the CAM entry is free. A CAM entry is free if and only if the reference count corresponding to the CAM entry is zero.

Identifying a free CAM entry is now as simple as finding a CAM entry with the corresponding bit set in the *free\_CAM\_vector*. This operation can be easily done using the *ffs* [1] instruction of the microengine. Whenever there is a CAM miss, instead of using the LRU to find a free entry, we find the free CAM entry using the *free\_CAM\_vector*. We can partition the *free\_CAM\_vector* among the microblocks running in parallel. For example, if microblock A runs on four threads and microblock B runs on the other four threads, then microblock A always searches for free entries among the first eight entries of the CAM while microblock B searches among the last eight entries of the CAM. This partition can be easily achieved by masking out the bits not corresponding to entries allocated to a microblock.

Assuming that every thread can only use one CAM entry at a time<sup>2</sup>, every microblock should be allocated a number of CAM entries at least equal to the number of threads running the microblock. In general, if every thread running a microblock can use a maximum of *n* CAM entries at a time, the microblock needs a number of

<sup>1</sup> When we say that a thread is *using* a CAM entry, we imply that the thread is in one of the “read,” “modify” or “write” operations for the data pointed to by the CAM entry.

<sup>2</sup> This is true in all the reference microblocks we have.

CAM entries =  $n * (\text{number of threads})$  running the microblock. Using this method, one can easily see that, every time there is a CAM miss, there will always be a free CAM entry among the CAM entries allocated to the microblock. For example, assume that we allocate each microblock the maximum number of CAM entries ( $n * (\text{number of threads})$ ) needed across all the threads running the microblock. Therefore, every time there is a CAM miss, the thread asking for a free entry must not be using all the  $n$  CAM entries allocated to it. Therefore, less than  $n * (\text{number of threads})$  entries must be in use leaving at least one free CAM entry. If more than  $n * (\text{number of threads})$  are allocated to a microblock, it can rotate the *free\_CAM\_vector* to use all the CAM entries allocated to it to achieve more caching. However, a minimum of  $n * (\text{number of threads})$  entries are required for the *folding* algorithm to work.

We just described the extensions for the support for *packet multicast and fragmentation* and *CAM Sharing* in the IXA Portability Framework that eases the software development on the IXPs. To further ease the development of the software on the IXPs, we have several advanced features in our development tools and a new programming language. We discuss them in the next sections.

## ADVANCED TOOLS FEATURES

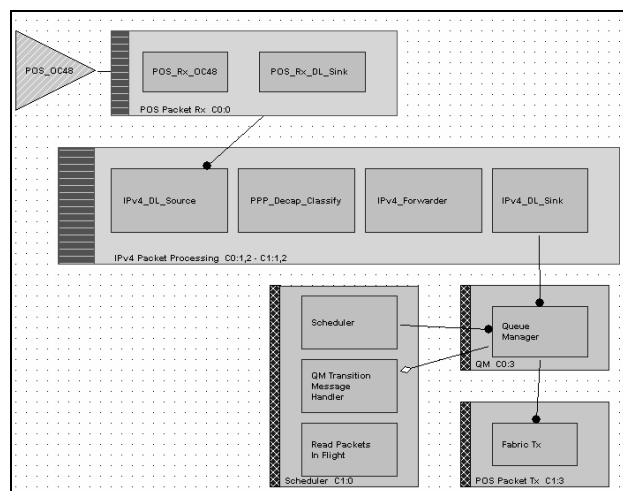
The Developer Workbench was introduced three years ago with the first IXP generation, the IXP1200. It has since been upgraded to support the most recent generations of IXP chips, the IXP2400, the IXP2800, and others. The Developer Workbench represents a major step in debug technology, with a patented graphical user interface for the presentation of multiple hardware execution threads, simulation history, and simulation status. It is a tool suite that integrates project software organization, chip configuration, microcode assembly, C compilation, linking, packet generation, cycle-accurate simulation, source-level debug, queue status monitoring, and simulation thread history. The tool suite has established a reputation as the leading development environment for network processors.

So far, our tools have provided a view of the software executing with no special features added to take into account the application domain. However, with network processors, the objective is clear. The application is working on packets or cells, forwarding data in a network environment. With this in mind, we are making the tools more application friendly, to make debugging more *packet-centric*.

## Architecture Tool

When deciding which IXP chip to use, or whether to even select one of Intel's IXP processors for a network processor-based system, there are many issues to consider. Will the application code fit in the microengine? What processing stage partitioning will likely meet the line rate requirements? Will the desired memory accesses per packet fit within the bandwidths supported by the chip? How many microengines will be needed to support the desired line rate? These questions and many more can be answered by the new IXP Architecture Tool.

[Figure 2](#) shows the packet processing stage block diagram for a POS OC-48 design. Using the IXP Architecture Tool, the user can lay out the partitioning of the design, specify data structures and task flow, and finally run analysis to determine whether the design "fits."



**Figure 2: OC-48 design using the architecture tool**

The Architecture Tool allows one to determine the feasibility of a design, before embarking on the full-blown project. With the major partitioning done, programmers can be assigned in parallel to work on coding the various blocks. In addition, the Architecture Tool views of code partitioning and data structures make it easy to determine where previous code can be reused (for example, microblocks from the IXA Software Framework), and where new code needs to be written.

Once feasibility has been proven, the Architecture Tool can be used to create a skeletal *Developer Workbench* project. With this as the implementation starting point, code can be written, compiled, and readied for debug.

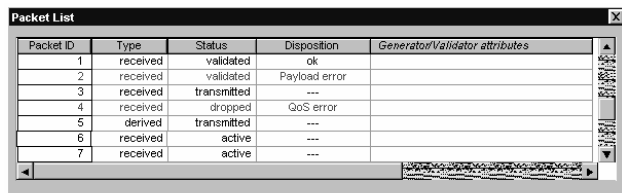
## Packet Generation

We have developed and added a new packet generator to the Developer Workbench, called *PacketGen*. PacketGen provides the capability to continually source and validate packets, according to the specification of plug-in protocol modules that define protocol types and a traffic specification of packet grouping into flows. The flows correspond to connections in ATM or TCP/IP. The headers can be layered for a flow in any order, and flows can be bundled within other flows. Sequencing algorithms include round-robin, token-bucket, and random order. There are many choices for setting the rate on a flow, from simple peak rate to the detailed parameters of an ATM TM4.1-based flow specification.

## Packet Status View

Previously, finding a packet in a simulation was a laborious effort unless you were the actual designer of the code, in which case you would have known where to set the breakpoints to stop execution at key packet processing milestones. Even so, debugging and tuning the design involved many reruns of simulation and many attempts to capture data that would lead to the cause of a problem. It was far more difficult to debug another's code. To speed up the debugging and performance tuning tasks of the project, a packet profiler has been added to the Developer Workbench. The packet profiler provides the capability of tracing packets from packet generation, through many events in the IXP chip simulation, all the way to validation of output packets after transmit.

[Figure 3](#) shows the new packet status view. This view is the starting point for packet-centric debugging. It shows the state of a packet, such as whether it has been received into the chip, transmitted, derived from another packet, or dropped. If it was dropped, the reason is displayed. If validation caught an error, such as an invalid header format, the symptom information is displayed.



Packet ID	Type	Status	Disposition	Generator/Validator attributes
1	received	validated	ok	
2	received	validated	Payload error	
3	received	transmitted	---	
4	received	dropped	QoS error	
5	derived	transmitted	---	
6	received	active	---	
7	received	active	---	

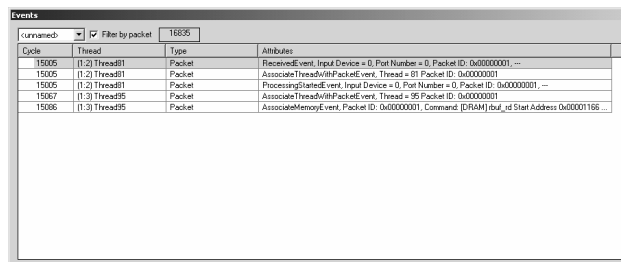
**Figure 3: Packet status view**

When the user sees a problem in the packet status view, a line can be highlighted to flag the “packet of interest.” At this time, a right click will take the user to the code associated with a major event, such as received (first code to work on that packet), or transmitted (last code to work on that packet).

## Event View

A packet may actually be in the chip for several thousand cycles. Having two major events (transmitted and received) four thousand cycles apart narrows it down somewhat, but there is still much tracing to do.

The *filtered event view* is shown in [Figure 4](#). Normally, this is a very big list of all history events, including memory reads/write, processing started on a piece of code by a thread, processing started on a microblock, packets received and transmitted, and other events. By selecting the *Filter by Packet* choice in the events dialog box, the number of events is now trimmed to just show those events involving the “packet of interest.”



Cycle	Thread	Type	Attributes
15005	[1-2] Thread01	Packet	ReceivedEvent_InputDevice = 0, Port Number = 0, Packet ID: 0x00000001, ...
15005	[1-2] Thread01	Packet	AssociateThreadWithPacketEvent, Thread = 01, Packet ID: 0x00000001, ...
15005	[1-2] Thread01	Packet	ProcessingStartEvent_InputDevice = 0, Port Number = 0, Packet ID: 0x00000001, ...
15007	[1-3] Thread05	Packet	AssociateThreadWithPacketEvent, Thread = 05, Packet ID: 0x00000001, ...
15006	[1-3] Thread05	Packet	AssociateMemoryEvent, Packet ID: 0x00000001, Command [DRAM] buf, rd Start Address 0x00001166, ...

**Figure 4: Packet events–filtered**

Now the user can right click to go to the exact places in the code associated with these events, and can then verify major checkpoints along the packet's life as it is being processed by the application.

## Packet Dataflow View

Previously, the thread history view was the primary debug window for watching the behavior of threads. This provides horizontal lines, one per thread, for up to 128 threads. The user could scan backward and forward in time and observe memory access events from request to completion. Also, the user could place labels in code to signify important code points and select them for display on the thread line of the thread history window.

Taking the history window paradigm further, [Figure 5](#) illustrates the packet dataflow view concept. Instead of threads running horizontally in the window, there are packets displayed. In addition to memory references, the data values of the memory reference are shown. Finally, the microblock partitioning is front and center, showing the flow of code for a given packet.

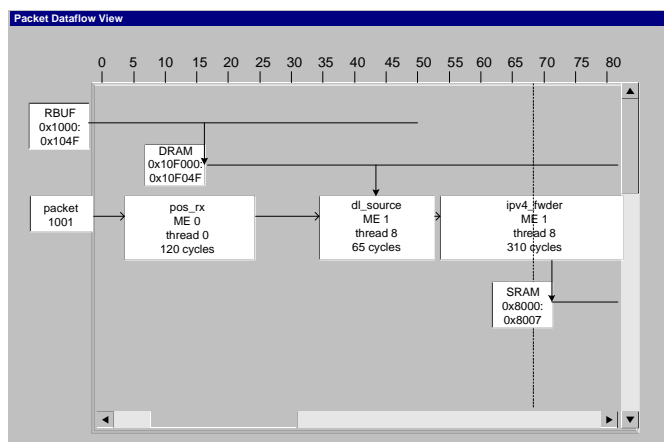


Figure 5: Packet dataflow view

The interaction of more than one packet can also be shown, such as access to the shared data in critical sections.

Using the packet dataflow view, the user can see at a glance the actual design running as envisioned in the Architecture Tool, with tasks and data structures. Erroneous data in data structures, such as bad packet data, incorrect table entry, or wrong inter-thread communication will be shown in this view, thus eliminating many steps of finding and looking up data values.

### Conditional Breakpoint

Previously, only unconditional breakpoints could be placed on specific lines of code. The GUI supported a selection of microengine contexts for the breakpoint to be applied. This typical debug procedure was a time-consuming process of manually stepping through breakpoints until the desired condition occurred. [Figure 6](#) shows the *breakpoint editor enhancement*, which provides a true conditional breakpoint capability.

The editor provides a function wrapper for the breakpoint. From within the function, the user can access and set all simulation states, including microengine register variables and memory values. It is even more powerful, in that registered console functions for any foreign model can also be called. It acts just like a function defined in the command line C-Interpreter. In this example, the breakpoint function accesses the IP destination address variable, tests whether it is within a range of values, and breaks only if the condition is satisfied.

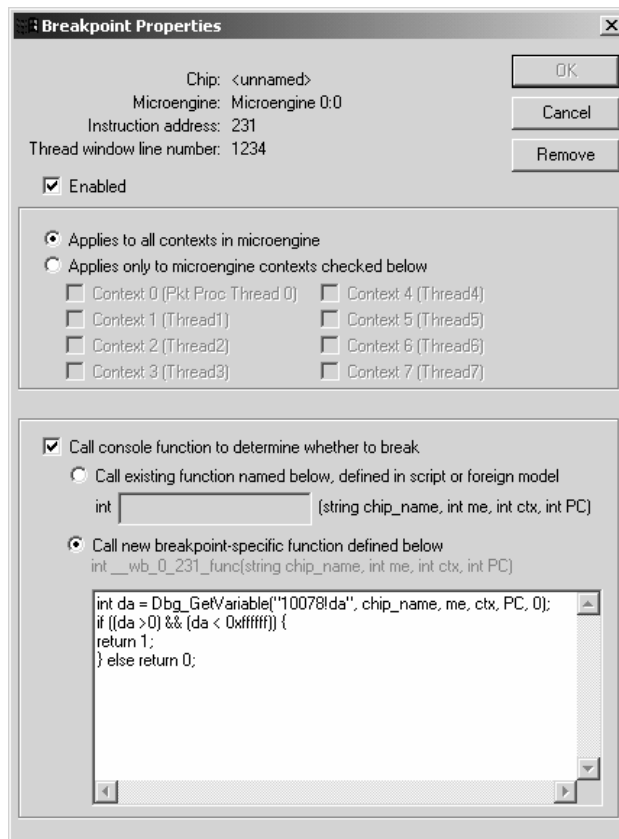


Figure 6: Breakpoint properties dialog

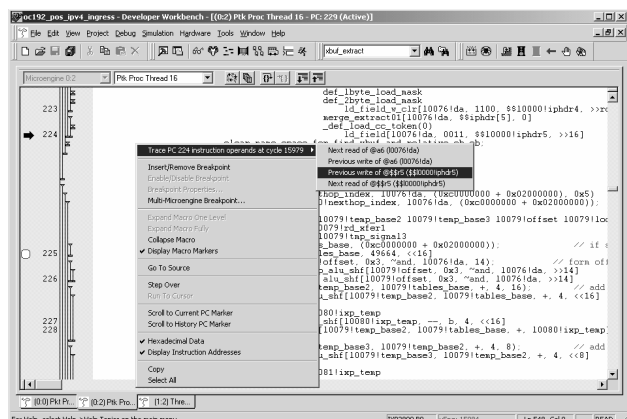
The conditional breakpoint can be used to mark packet events as well, such as the creation of derived packets for multicast, the notification for a dropped packet, or it can tag that a packet processing stage has been entered. These events will show up in the packet events view. Major events, such as packet derived or dropped, will show up in the Packet Status View.

As we described earlier, when the event has appeared in packet status or events view, tracing a bug is a simple matter of highlighting the “packet of interest” in either of these two windows, and right clicking to go to the code that caused the event.

### Instruction Operand Trace

Previously, when simulation had stopped, the user would scan through the code in *thread list view*, hovering over variables to find one that was not “right.” Sometimes this could be accelerated by using the editor’s *find* feature, with the guess that the previous setting of a register was in the up direction and in the same context and microengine. To speed up the tracing of variable values, the instruction operand trace feature is added to the Developer Workbench.

[Figure 7](#) shows the use of instruction operand tracing in the thread list view.



**Figure 7: Instruction operand tracing**

When the user selects the backward trace to the previous write of an operand, the thread list goes to that code line, and the current “cycle of interest” is changed for all event and history views. If another microengine wrote the operand (e.g., a next-neighbor write), the thread list view will bring up the thread in the other microengine and scroll to the line of code that initiated the write. This saves a couple of steps in the search; plus, it takes the user quickly to the cause or the result.

Considering the hundreds of times a user tries to trace operand values during design test and validation, the cumulative effect of instruction operand trace feature results in a significant time saving.

## Tool Enhancements Summary

The tool enhancements provide a major step forward in network processor tool technology. Collectively, they shift the focus of development toward the application domain. They provide a packet-centric approach to the design and debug of network processor applications. In this paper we presented the following tool enhancements:

- *Architecture Tool*, for top-level partitioning and feasibility analysis.
- *Advanced Packet Generation*, for continuous generation and validation according to extensible protocol plug-ins.
- *Advanced Packet Profiler*, with packet list view summarizing packet status and providing hooks to other graphics views.
- *Comprehensive Event History*, with filtering and instant jump to associated code.

- *Packet Dataflow View*, extending the Packet Profiler with presentation of packet code flow and associated data structures.
- *Conditional Breakpoint*, with full access to simulation states and simulator console functions.
- *Instruction Operand Tracing*, with fast jump forward or backward in simulation history.

So far, we have discussed incremental extensions or enhancements to the IXA software framework and development tools. However, we have also been investigating some long-term directions in simplifying the software development by developing a next-generation, packet processing language.

## NEXT-GENERATION PACKET PROCESSING LANGUAGES

### Why Build A New Language?

Network processing (NP) systems are designed to meet two, often conflicting, requirements: support of a large number of high-bandwidth links, and hence large system throughputs and, at the same time, offer a wide range of services. To meet these requirements simultaneously, NPs support mechanisms such as multiple processor cores per chip and multiple hardware contexts per processor core that enable them to process network packets at high rates.

Currently, the languages used for programming these systems expose much of the hardware details to the programmer. Hence, to develop high-performance packet processing applications on such systems, programmers are required to become intimately familiar with the hardware and develop hand-tuned code for managing NP resources.

One solution to the above problem is to take a general-purpose language such as C and enhance it with new constructs and programmer-specified annotations to allow representation of independent modules and their interactions. In this approach, the compiler uses the information gleaned from the use of new constructs and program annotations to automatically map a sequential program onto multiple processing cores and threads. A programmer is still responsible to map various data structures in different parts of the memory hierarchy. IXP C is such an extension to the standard C language that has been described elsewhere [7].

In this paper, we describe yet another approach that hides almost all the hardware details by taking an existing general-purpose programming language and extending it with domain-specific features, to enable both the

programmer and compiler to build efficient network systems. Here we describe the vision of such a language called Baker.

### What is Baker?

Baker is the programming language for a network application development environment currently being researched, called *Shangri-la*. Baker is a domain-specific programming language in that, although it is based on C, it has many packet-processing specific features added to it. These features not only make the job of the programmer easier but they also narrow the scope of possible applications the compiler needs to handle, and expose critical information enabling the compiler to make informed decisions and generate efficient binaries.

### Domain-Specific Features

The domain of packet processing has several key features that differentiate it from other problem domains. In the packet-processing domain, the code and data are usually highly independent. NPs have been designed with multiple cores and multiple hardware threads in order to take advantage of this independence, and to be able to execute much of the code in parallel. This results in one of the most difficult jobs of the developer: partitioning the program among the cores and threads.

Baker exposes this concurrency of the data by explicitly representing packets in the language, and exposes concurrency in code through a data flow model.

### Packets

The majority of the work done in NP systems involves working with the data and metadata of packets. Baker treats packets as first-class types in order to identify to the compiler data that is independent, and hence parallelizable. This enables the compiler to optimize code working with this data. It also gives the programmer a higher level view of them, simplifying the code.

### Data-Flow

One of the challenges of programming multicore multithreaded systems is determining how to partition the application in such a way that it can be executed over the cores and threads to take advantage of the highly threaded hardware and still keep the code reusable. Much research has gone into compilers that can automatically partition programs written in general-purpose languages, but this is generally ineffective in this domain because these compilers must make conservative assumptions about the independence of the code and data. Baker's approach to this problem is to use a data flow model. In this model, a programmer can make it

known to the compiler which pieces of the application are reasonably independent and therefore can be run in parallel. With the addition of knowledge of packets and their independence, the compiler can then lay these pieces out on the cores and threads as it sees fit.

In a data flow model, many functional units, or *actors*, are linked together to form a greater application. These actors process their input and produce outputs, analogous to the *pipe and filter model* in a UNIX shell. Baker uses the concepts of Packet Processing Functions (PPFs), and *channels* for describing the data flow, which are heavily influenced by the language Click [4]. The programmer writes several reasonably independent PPFs to do the work of a network application and connects the PPFs via channels in a module. A module is a collection of PPFs and channels. A PPF is an actor in the data flow model and contains the data and functionality for a small piece of a network application. PPFs can have any number of inputs or outputs that are later connected to other PPFs by channels. Channels are typed, unidirectional interconnects that may or may not be synchronous. Because the definition of a channel is very open, the compiler is free to replace them with whatever construct it deems optimal for a particular system.

Figure 8 illustrates a simple L3 switching module described in a data flow model, containing three PPFs, and the following code snippet represents how its definition would be written in Baker.

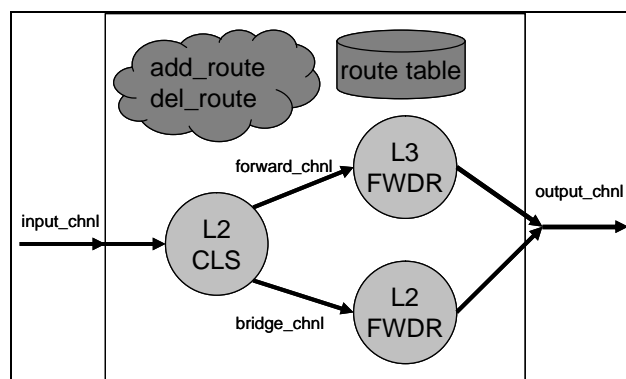


Figure 8: An L3 switch module

```
// L3 switch module
module L3Switch {
  ppf L2Cls; //forward declarations of
  ppf L2Fwdr; //required PPFs
  ppf L3Fwdr;

  channels {
    input packet input_chnl;
    output packet output_chnl;
  }
  wiring {
    //equate this module's external channel
    //endpoints to internal PPFs'
    input_chnl = L2Cls.input_chnl;
    output_chnl = L2Fwdr.output_chnl;
  }
}
```

```

output_chnl = L3Fwdr.output_chnl;

//bind internal PPFs channel endpoints
L2Cls.bridge_chnl -> L2Fwdr.input_chnl;
L2Cls.forward_chnl ->L3Fwdr.input_chnl;
}

//module's data
route_table_type route_table;

//module's interface
void add_route(route_t r);
void del_route(route_t r);
};

```

In this example, the *L3Switch* declares the PPFs that it contains and binds their channel endpoints together. This binding defines the processing pipeline for this module. The *L3Switch* also exposes two channel endpoints of its own. These endpoints are equated to endpoints of internal PPFs, allowing the module to be used by another developer without knowing its internal details. Any packets sent to the module's input channel will be sent to the L2Cls PPF. This wiring concept is borrowed from the nesC [5] language developed for TinyOS [6].

## Consequences of Partitioning a Baker Application

Once the application is written, the Baker compiler uses knowledge about the packets and data flow (actors and channels) to partition the code among the processors and threads of the NP.

Many PPFs are capable of being replicated and run on many cores simultaneously as long as the data that they are working on is reasonably independent. Baker assumes that all packets are completely independent, and thus all PPFs may be replicated. If packets worked on by two or more PPFs are interdependent, then a programmer must specify ordering criteria for the packets in the definition of the input channel of the PPF.

Given that the code of the network application may be run in parallel in a way that is completely determined by the compiler, the programmer must understand at all times that any code may be run by any number of threads simultaneously. Therefore, a Baker programmer must take precautions, such as using locks, to protect any data that may be shared to avoid problems.

## Portability

One advantage of a domain-specific language such as Baker, is that it can aid in the portability of applications. Specifically, by abstracting the key aspects of the hardware, the programmer no longer becomes concerned with the exact processor configuration in the NP.

To this end, Baker presents an abstract hardware model to the programmer to simplify both the programming and porting of applications. The basic model is one that is multithreaded with a flat, shared memory. Although the model is multithreaded, the programmer has no control over or knowledge of the threads. This lack of control is important to avoid burdening the compiler with the task of automatically synchronizing access to shared data. Rather than try to solve the hard problem of building a compiler that can analyze code to automatically find any synchronization issues on its own, Baker leaves that work to the programmer. Because the programmer knows the system is multithreaded, all global or shared data must be protected via locks.

The flat memory model presented to the programmer simplifies coding as the developer no longer has to place variables in one of any number of memory types and locations, or write the accompanying specialized accessing code. The compiler can automate the task of mapping data structures to memory types, for example, using knowledge such as which memory accesses are for packets and which are not.

As an example of portability, let's consider how Baker enables programmers to work with packets without dealing with different memory types or packet buffer data structures. Baker allows the programmer to define protocol header templates, and then access packet data using protocol fields and several built-in functions. A protocol header template is much like a struct in C, except that there are no types to each field, only sizes, and some special features have been added to handle special cases, such as optional fields.

An example of a protocol definition for EthernetSNAP, and the subsequent packet access through that protocol is given below:

```

// The LLC and SNAP subfield
field LLC SNAP {
    DSAP : 8;
    SSAP : 8;
    Control : 8;
    Vendor : 24;
    LocalCode : 16;
};
protocol Ethernet2 {
    dest : 24;
    src : 24;
    len : 16;
    snap : anyof {
        { len > 1500 } : 0;
        default: LLC SNAP;
    }
    demux { (len<1500)?len:14 };
};

```

Here we have defined two structures, one a protocol and the other a sub-field of a protocol. Elements of each are given a name and a length in bits. The ordering is important, as each field is found in the packet by the total

number of bits used up before it. The compiler generates the correct code to access the bits in the actual packet when the protocol is used by the programmer, no matter how the packet is represented in memory. Protocols are allowed to define pseudo-fields, which do not affect the placement of the regular fields, but give a tool to the protocol developer to ease the burden on the programmers using it. The *anyof* directive is a way for a protocol to have optional fields. Here, the field *snap* takes on the properties of either a zero length field or of an LLC SNAP, depending on the evaluation of the cases of the *anyof*. The *demux* pseudo-field is a required field for all protocols which allows the compiler to generate the code necessary to find the next encapsulated protocol. Depending on the value of *len*, the next protocol header can be found either *len* bytes or 14 bytes after the beginning of this header.

```
//example function using the above protocol
void process(Ethernet2_packet_t* p)
{
    IPv4_packet_t* ip;
    if (p->len < 1500)
        processSnap(p)

    ip = packet_decap(p);
    if (ip->version == 6)
        processIPv6(ip)
    else
        processIPv4(ip)
}
```

Notice the new types, *Ethernet2\_packet\_t* and *IPv4\_packet\_t*. Once the compiler has parsed a protocol definition, it creates new types for them, which are subtypes of the built-in packet type. The fields are accessed just as the fields of a struct would be, which makes coding very simple, and hides any implementation details. The *packet\_decap* function uses the required *demux* field to find the next header and returns a pointer to the protocol found there. Notice that whether or not there was a SNAP header, the code to decapsulate is exactly the same.

Given that no details are provided to the programmer, the compiler is free to make any optimizations to the handling of packets and data. And if anything changes, or the code is targeted to a system that handles packets completely differently, then all that is required is a simple recompile.

## CONCLUSION

We first identify the limitations of the current IXA software framework and development tools, and then describe the incremental enhancements as well as some breakthrough research in providing much better tools to an IXP developer. We describe simple extensions to the framework to overcome these limitations without affecting performance. These extensions facilitate the

development of new applications (such as multicast, fragmentation) using the IXPs while supporting seamless portability between very high-end and low-end IXPs.

In addition, we describe features of the advanced tools. Collectively these features shift the user's focus toward the application domain of packet-centric processing, and away from the hardware details. Rather than having to remember the relationship between application code and microengine/thread assignments, the user can now focus on the progress of test packets and major application events. The user can instrument code and create special events that can also be traced.

The packet-centric approach enables instrumentation of code without inserting hardware-specific hooks. Therefore, the instrumentation portion itself is also portable. The new packet-centric events and dataflow views also help in debugging the same design that has been ported across IXP versions.

Finally, we discussed future directions in packet processing languages that provide benefits to both programmer and compiler writer, and gave a brief overview of how to realize some of these directions in the Baker language. Only a few of the advantages and tools it provides for easing the development of packet processing applications are discussed here. Although much research remains to be done on Baker and its compiler environment, Shangri-la, we already see promising results in related technologies such as the IXP-C [7].

## ACKNOWLEDGMENTS

The work presented here resulted from the contribution of many members of our team. Authors are especially grateful to Uday Naik, Prashant Chandra, Erik J. Johnson, and Eric Walker for in-depth discussions and original technical contributions.

## REFERENCES

- [1] Matthew Adiletta et al., "The Next Generation of Intel IXP Network Processors," *Intel Technology Journal*, Volume 6, Issue 03, August 15, 2002, pp. 6-18.
- [2] Intel® IXP2400/2800 Software Development Toolkit Version 3.5, Intel® IXP2400/2800 Network Processors Development Tools User's Guide, September 2003.
- [3] Uday Naik et al, "IXA Portability Framework: Preserving Software Investment in Network Processor Applications," *Intel Technology Journal*, Volume 6, Issue 03, August 15, 2002, pp. 50-60.

- [4] E. Kohler, et. al., "The Click Modular Router," *Technical Report Laboratory for Computer Science*, MIT, 2000.
- [5] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," in *Proceedings of Programming Language Design and Implementation (PLDI) 2003*, June 2003.
- [6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System Architecture Directions for Networked Sensors," *Architectural Support for Programming Languages and Operating Systems*, pp. 93–104, 2000.
- [7] Intel Corporation, "Introduction to Auto-Partitioning Programming Model," *Literature number 254114-001 at* <http://www.intel.com/design/network/papers/254114.htm>

## AUTHORS' BIOGRAPHIES

**Stephen D. Goglin** is a network software engineer in the Communications Technology Lab of CTG at Intel Corporation. He received his M.Sc. degree in Computer Science from the University of Victoria in 1999. Stephen's interest includes graph theory, code generation, and his wife and two children. His e-mail is [stephen.d.goglin@intel.com](mailto:stephen.d.goglin@intel.com).

**Donald Hooper** is a senior software architect in the Network Processor Group. He has led many projects including Logic Synthesis, Video Servers, MPEG 2 and DAVIC Standards, IXP1200 Software Tools and Libraries, IXP2800 Proof of Concept Designs, and NPG Coding Standards. His professional interests include networking, artificial intelligence, and object-oriented languages. He attended four colleges with cumulative B.S.E.E. degree credits finishing at UCLA. He resides in Shrewsbury, Massachusetts, and can be reached via e-mail at [donald.hooper@intel.com](mailto:donald.hooper@intel.com).

**Alok Kumar** is a software architect in the Software Architecture Group of the CIG GTO at Intel Corporation. He has worked on new algorithms for IPv6 forwarding, packet classification and software framework for IXP2xxx. His interests are in the areas of high-speed programmable routers, quality of service, algorithms and computer graphics. He received his B.Tech degree in Computer Science from the Indian Institute of Technology, Delhi, in 1999, and his M.S. degree in Computer Science from the University of Texas at Austin in 2001. His e-mail is [alok.kumar@intel.com](mailto:alok.kumar@intel.com).

**Raj Yavatkar** is currently the chief software architect for the Internet Exchange Architecture at the Intel Communications Group. Raj Yavatkar received his Ph.D. in Computer Science from Purdue University in 1989. He is a co-author of the book "Inside the Internet's Resource reSerVation Protocol (RSVP)" published by John Wiley in 1998. He currently serves on the editorial board of the IEEE Network magazine. His e-mail is [raj.yavatkar@intel.com](mailto:raj.yavatkar@intel.com).

Copyright © Intel Corporation 2003. This publication was downloaded from <http://developer.intel.com/>.

Legal notices at <http://www.intel.com/sites/corporate/tradmarx.htm>.

For further information visit:

[developer.intel.com/technology/itj/index.htm](http://developer.intel.com/technology/itj/index.htm)