Supporting Run-time Adaptation in Packet Processing Systems

Vinod Balakrishnan[†], Ravi Kokku[‡], Aaron Kunze[†], Harrick Vin[‡], Erik J. Johnson[†]
[†]Intel Research and Development [‡]University of Texas at Austin
[†]{vinod.k.balakrishnan,aaron.kunze,erik.j.johnson}@intel.com
[‡]{rkoku,vin}@cs.utexas.edu

Abstract:

Implementors of packet-processing applications on multi-core processors must balance two requirements: (1) adapt processor allocations dynamically to reduce the overall resource provisioning requirement for the system, achieve robustness to traffic fluctuations, and reduce energy consumption; and (2) utilize, for each application stage, resources (e.g., memory levels, inter-processor communication mechanisms, etc.) closer or local to the processors on which the stages are mapped to achieve the highest possible throughput. In this paper, we describe the design and implementation of a run-time adaptation system that can meet these two requirements simultaneously. Our design allows each application stage to utilize local resources whenever possible in the steady state. Upon adapting the allocation of processors to stages, the run-time system (1) binds each resource usage within a stage to a new resource instance; and (2) checkpoints and migrates the state from the previous resource instance to the newly-bound resource instance. We describe the design and implementation of our adaptation system in the context of a packet processing system designed using the Intel® IXP2400 network processor. We show that our design has little impact (14%) on the steady-state throughput of the system. We further show that our design is able to perform resource adaptation for a real application in less than 100ms, allowing processor allocations to be adapted at a very fine time-scale.

1 Introduction

Packet processing systems are optimized to process network packets efficiently. These systems are required to support high-bandwidth links (and hence, high packet processing throughput) as well as a set of complex packet processing applications (e.g., protocol conversion, firewall, Secure Socket Layer, intrusion detection, and virus scanning). Together, these requirements yield scenarios where the time to process a packet exceeds the inter-arrival time of packets at a system. To support complex applications in high-bandwidth environments, modern packet processing systems (and in particular, network processors) utilize multi-threaded, multi-processor architectures, sophisticated memory hierarchies, and specialized inter-processor communication mechanisms [6]. For instance, the Intel® IXP2400 network processor includes eight RISC microengines (ME) each with eight hardware threads and an Intel XScale® core, a multi-level memory hierarchy with memory local to individual microengines, on-chip memory shared across all processor cores, interfaces to external SRAM and DRAM, as well as next-neighbor rings and multiple hardware queue implementations for efficient data movement between processor cores. Such network processor architectures fore-shadow a more general trend toward the design of multi-core, multi-threaded architectures targeted for high-throughput computing environments.

Packet processing applications are often implemented on such multi-core, multi-threaded architectures using the *pipelined model* of computation. In this model, application functionality is partitioned into a graph of packet processing stages connected by communication channels (or queues); each stage is then mapped onto one or more processors [17, 31, 15, 13]. The sequence of stages invoked for processing a packet depends on the type of the packet (determined based on the header/payload of the packet) [14, 17]. For example, a Secure Socket Layer [10] termination application processes three packet types—setup packets (that create per-flow state), outgoing packets (that involve encryption), and incoming packets (that require decryption).

To implement such pipelined applications on parallel architectures, designers must balance two requirements. First, to achieve the highest possible throughput, each pipeline stage should utilize resources (e.g., memory levels, inter-processor communication mechanisms, etc.) closer or local to the processors on which they are mapped; accessing local resources incurs smaller latencies and minimizes the use of shared global resources. Second, because of the traffic fluctuations inherent in packet networks, the workload seen by each pipeline stage may vary significantly over time. Hence, the allocation of processors to pipeline stages needs to be adapted over time. Adapting processor allocations dynamically reduces the overall resource provisioning requirement for the system, achieves robustness to traffic fluctuations, and reduces energy consumption. In fact, it has been shown that a device that can adapt–without incurring any adaptation overheads–the mapping of pipeline stages to processors at run-time needs up to 50% less processing resources to process the same packet traces, and can reduce its average power consumption by up to 80% [18]. Realizing these benefits, however, requires the system to adapt processor allocations at fine time-scales.

These two requirements—adapting processor allocations at fine time-scales and achieving high throughput—can be contradictory. Whereas allocation of local resources to pipeline stages is desirable for achieving high throughput, doing so complicates adaptation on systems with diverse hardware resources (such as the IXP2400 network processor). Since many of these hardware features have restrictions on how and when they can be used, it is difficult to consistently use the most efficient hardware resources when processor allocation is adapted dynamically. Adaptation can be simplified if pipeline stages use only globally accessible resources; however, doing so can decrease significantly the throughput supported by the system. Hence, a key challenge is to design a packet processing system that can support adaptation and achieve high throughput simultaneously.

In this paper, we describe the design and implementation of a run-time adaptation system that meets this challenge. Our design allows each pipeline stage to utilize local resources whenever possible in the steady state. Upon adapting the allocation of processors to stages, the runtime system (1) binds each resource usage within a pipeline stage to a new resource instance, and (2) checkpoints and migrates the state from the previous resource instance to the newly-bound resource instance. We show that by exploiting the features of packet processing applications one can reduce significantly the overhead of run-time adaptation. We describe the design and implementation of the checkpointing/state-migration mechanism and the resource binding mechanism in the context of a packet processing system designed using the Intel® IXP2400 network processor. We show that our design has little impact (14%) on the steady-state throughput of the system. We further show that our design is able to perform resource adaptation for a real application in less than 100ms, allowing processor allocations to be adapted at a very fine time-scale.

The rest of the paper is organized as follows. In Section 2, we formulate the problem of adapting resource allocations for packet processing applications, and discuss the design alternatives for the check-pointing/state-migration and resource binding mechanisms. In Section 3, we describe our implementation. We present experimental results from our prototype in Section 4, and discuss optimization opportunities in Section 5. Section 6 compares our work with related research, and finally, Section 7 summarize our contributions.

2 Problem Formulation

Efficient use of resources (e.g., memory and communication channels) local, or closer, to processor cores is critical for achieving the best possible performance for applications running on network processors. In such environments, dynamically adapting allocation of processor cores to application stages offers some unique challenges. In this section, we first motivate the use of local resources for achieving high throughput in the context of the Intel® IXP2400 network processor (Section 2.1), and then describe the challenges in designing an adaptive system on this platform (Section 2.2).

2.1 A Case for Using Local Resources

The Intel® IXP2400 network processor includes eight RISC microengines each with eight hardware threads and an Intel XScale® core. It offers multiple implementations of packet channels to facilitate communication of packets between pipeline stages in the application, as well as multiple implementations of locks to provide synchronization between threads. These implementations have different performance characteristics and different restrictions on when they can be used. For example, the next-neighbor registers in the IXP2400 network processor can be used to move packets from a microengine *only* to its neighboring microengine. In addition to next-neighbor registers, the IXP2400 network processor offers globally available queuing mechanisms that can be used to move packets between *any* pair of processors including (1) a limited number of scratchpad rings implemented using on-chip memory, and (2) off-chip SRAM memory that can be used to implement packet channels in large quantities but with lower performance. When the mapping of application pipeline stages to processors changes, the set of available hardware resources may change as well and hence the performance of the application depends on the ability to adapt to this changing resource availability. Adaptation can be simplified if pipeline stages use only globally accessible resources (e.g., scratchpad rings or SRAM rings); however, since using non-local resources results in unnecessary usage of internal and external bus bandwidths and incurs larger latency, the throughput supported by the system can decrease significantly.

To quantify the impact of inefficient usage of hardware resources, consider the bus structure of the IXP2400 network processor. The various internal buses in IXP2400 network processor used by packet channel implementations are shown in Figure 1. The IXP2400 network processor has four internal buses that connect the microengines to SRAM controllers and scratchpad memory. These buses are called the S-Push and S-Pull buses. Each cluster of microengines shares an S-Push bus for sending data to devices and an S-Pull bus for receiving data from devices. There are two microengine clusters on the IXP2400 network processor, each with four microengines. The IXP2400 network processor has two SRAM controllers, each with a read and write bus connected to external SRAM memories.

The impact on the available bus bandwidths for different channel implementations is shown in Tables 1 and 2. The SRAM ring implementation is a pure software implementation that can be used to communicate packets between any two processors. The scratchpad ring implementation can also be used for any pair of processors, but there are only sixteen such rings in the system. The next-neighbor

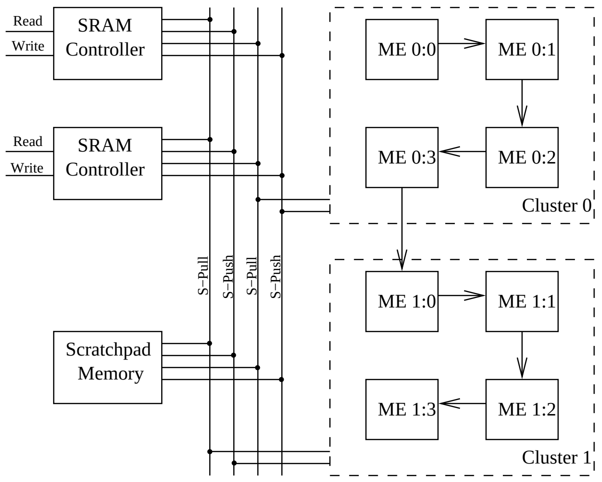

Figure 1: IXP2400 network processor internal buses

ring implementation can only be used between adjacent microengines. Table 1 shows the number of bytes transfered on the S-Push and S-Pull buses for a single packet that is successfully transmitted over an instance of the different packet channel implementations. Table 2 shows the impact of using the various packet channel implementations on the SRAM buses. In both tables, the bandwidth percentages show the amount of bus bandwidth utilized for supporting a single packet channel processing 1Gb/s of 64-byte Ethernet packets from one microengine to another. The data in these two tables are based on a 600MHz IXP2400 network processor[23]. Note also that, for both the internal and external buses, the impact of using SRAM rings with statistics shown in Tables 1 and 2 represent the best-case statistics. The SRAM ring implementation uses spin-locks in SRAM to provide synchronization; hence, the bus overhead would be even greater in the presence of higher lock contention.

The differences in bus bandwidth usages in Tables 1 and 2 result in differences in packet processing throughput. We evaluated the throughput impact of using different packet channel implementations by comparing the forwarding rates achieved by the IPv4 switching and forwarding application shown in Figure 2. The L2 classification, L3 forwarding, and Ethernet encapsulation pipeline stages were each mapped to only one microengine. We varied the packet channel implementation used by channels 2 and 5 between next-neighbor ring implementations, scratch ring implementations and SRAM ring implementations and measured the forwarding rate of the application in the different configurations. The application input traffic consisted of packets destined to the L3 forwarding pipeline stage. We measured the forwarding rate while varying the packet size from 64 bytes to 256 bytes. The result of this experiment is shown in Figure 3. This result was obtained using the experimental setup described in Section 4. Since this application is compute-intensive and not bound by internal bus bandwidth, it does not show a throughput benefit in using next-neighbor rings over scratchpad rings, but we don't expect this to be true for all applications. In either case, the results clearly show that using on-chip resources for communicating packets between pipeline stages offers significant performance improvements.

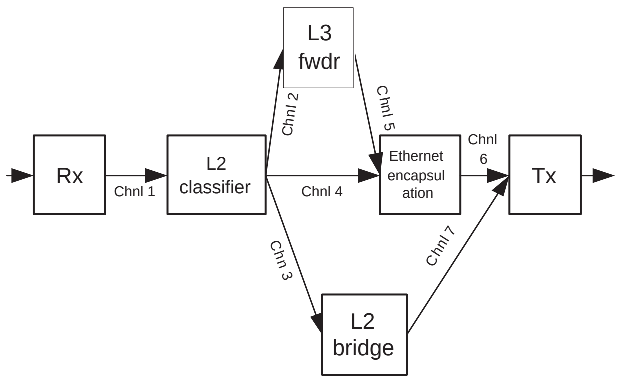

Figure 2: Layer 3 switching and forwarding application

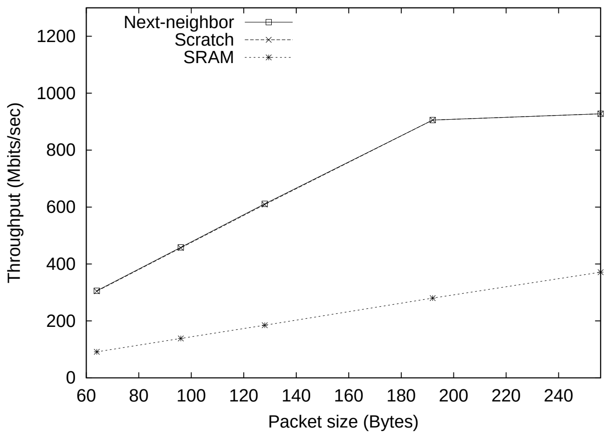

Figure 3: Throughput benefits of packet channel adaptation(Source:Intel)

Lock resources used for synchronization also consume unnecessary bus bandwidth when the most efficient implementation is not used when it is available. For example, using a spin-lock in microengine local memory provides synchronization between microengine threads without using any bus bandwidth, whereas using a spin-lock in scratchpad memory or SRAM uses internal and/or external bus bandwidth. The impact on system performance due to spin-locks in scratchpad memory or SRAM will vary greatly based on lock contention.

Clearly, using the most efficient packet channel and lock implementations available to the application in its current processor mapping is critical for maximizing system per-

Implementation	# of S-Push Bytes	% of S-Push Bandwidth	# of S-Pull Bytes	% of S-Pull Bandwidth
Next-neighbor	0	0%	0	0%
Scratchpad Ring	4	0.47%	4	0.47%
SRAM Ring w/ Stats	68	7.9%	68	7.9%

Table 1: Internal bus usage for packet channel operations on one channel carrying 1Gb/s of 64-byte packets

Implementation	# of SRAM Read Bytes	% of SRAM Read Bandwidth	# of SRAM Write Bytes	% of SRAM Write Bandwidth
Next-neighbor	0	0%	0	0%
Scratchpad Ring	0	0%	0	0%
SRAM Ring w/ Stats	76	13%	68	12%

Table 2: External bus usage for packet channel operations on one channel carrying 1Gb/s of 64-byte packets

formance. Hence, in the face of adaptation, a mechanism for rebinding pipeline stages to resource implementations is needed in order to maintain high throughput on an adaptive packet processing system.

2.2 Adaptation Mechanisms

Our design allows each pipeline stage to utilize local resources whenever possible in the steady state. Upon adapting the allocation of processors to stages, the run-time system (1) binds each resource usage within a pipeline stage to a new resource instance, and (2) checkpoints and migrates the state from the previous resource instance to the newly-bound resource instance. In what follows, we describe the challenges in implementing these mechanisms.

2.2.1 Resource Binding

Packet-processing environments offer some difficulties for the design and implementation of binding mechanisms. First, given the extreme performance requirements of many packet-processing applications, the solution must have very little impact on the run-time performance of the system. Second, since the workloads in a packet-processing environment fluctuate frequently and since packet-processing is often done in an embedded environment, the solution must be simple enough to allow for rebinding to occur on a very fine time-scale. Finally, the solution must work well in a system with fixed-size code stores, such as the IXP2400 network processor microengines.

A few alternatives for binding mechanisms exist. For example, run-time checks could direct packet-processing code to use the correct hardware resource. So, when a pipeline stage wants to perform an operation on a packet channel or a lock, the code could check a variable in memory that identifies the resource instance the code should use. The variable could be an enumeration or a set of function pointers. We call this mechanism “run-time binding”. Run-time binding has the advantage that it does not require any modification of the application binary to adapt the binding. When the rebinding occurs, the system just rewrites a variable in memory to make the rebinding take effect. One disadvantage of run-time binding is that it incurs a run-time performance penalty. Every time a resource is accessed, the indirection needed to implement run-time binding uses valuable processor/memory resources. Run-time binding has another disadvantage in a fixed-size code store environment such as an IXP2400 processor microengine. In this environment, all of the resource implementations would need to fit in the code store, which has very limited space.

Another alternative for binding, at the other end of the complexity spectrum, is recompilation. Resource implementation could be compiled into the pipeline stage, and when the resource is rebound, the pipeline stage could be re-compiled. This has the advantage of incurring no run-time performance penalty, as well as perhaps leveraging compiler optimizations to gain extra run-time performance. On the other hand, re-compilation is a complicated operation that would severely limit the frequency at which adaptation could be done.

We designed and implemented a solution in between these two points on the complexity spectrum. Our solution has the advantages of having very little run-time performance penalty, being sensitive to limited instruction store processors, and being much less complicated than re-compilation.

The implementation details of the mechanisms used to perform resource binding are described in Section 3.

2.2.2 Checkpointing and State Migration

To adapt processor allocations, a mechanism is needed to adjust the mapping of pipeline stages to processors at run-time and to migrate state between them. This state migration involves migrating code between processors and migrating any data being manipulated by the code. It also involves migrating data being managed by other allocated resources, such as packet channels. This mechanism must be designed to meet the following requirements:

- Allow for adding and removing processing resources to pipeline stages and adaptating other resources at run-time without losing any intermediate state in the running pipeline stages. For example, if a packet is being processed by a pipeline stage or stored in a packet channel, the memory buffer storing the packet should not be lost when the resource mapping of the stage is changed. This is complicated on processor architectures with resources local to individual processors that is not easily accessible from other processors.

- Allow migration of pipeline stages between different processor architectures, including processors with fixed-size instruction stores (such as IXP2400 network processor microengines) and processors with instruction caches (such as the Intel XScale® core).

- Be sensitive to the performance requirements of packet processing code. In this environment, packet inter-arrival rates are on the same timescale as memory latencies, which means the mechanisms must have as little run-time performance impact as possible.

Existing mechanisms used in symmetric multiprocessing systems and parallel processing systems do not meet these requirements. For example, the Linux kernel does not allow for the safe addition or removal of processing resources to code at run time, or for the migration of processes between processor architectures, and its mechanism for migrating processes depends on an instruction cache on each processor[22]. Checkpointing–a method used to migrate processes in the parallel processing domain–could be adapted to fit these requirements, but it imposes an unacceptable run-time performance penalty [20] and does not allow for adding and removing processing resources.

To address these requirements, we have designed a mechanism that leverages the unique properties of packet-processing applications. We describe this mechanism in detail in Section 3.

3 Design and Implementation

We implemented a run-time system (RTS) on the IXP2400 network processor that meets the requirements for the run-time adaptation mechanisms listed in the previous section. In this section we explain the design choices we made for the adaptation mechanisms that we implemented in the RTS. We also describe the important steps that the RTS performs in adapting from one processor mapping configuration to another.

3.1 Design/Implementation of resource binding

We implemented an adaptation-time linking solution for the resource binding mechanism. To enable this mechanism, a pipeline stage binary uses resources, like packet channels and locks, through a software interface definition (a set of methods) to the pipeline stage binary. Each interface can have multiple implementations that utilize different hardware resources. The adaptation-time linking solution links in the appropriate implementation into the pipeline stage binary.

This approach has the advantages of not requiring a recompilation of the pipeline stage binary every time the binding of a software construct needs to change and incurs less run-time performance and code store size overheads than a run-time binding solution. Less code store is required since only the implementations used are linked in the final binary. In the remainder of this section we describe the software interface abstracting the resources and then describe the adaptation-time linking solution.

3.1.1 Resource Abstraction

Resources are exposed to a pipeline stage binary by the Resource Abstraction Layer (RAL). The RAL provides a set of interface definitions which we call *RAL interfaces*. We call a method in a RAL interface a *RAL method*. Each RAL interface can be realized using different hardware resources and we call each realization of a RAL interface a *RAL implementation*. A pipeline stage binary can have multiple instantiations of RAL interfaces and we call these *RAL instances*.

Figure 4 shows the important hardware resources in a simplified representation of the Intel IXP2400 network processor. The different resources have different constraints on their usage and different performance characteristics as described in Section 2.1. The RAL interfaces required for adaptation and the different hardware resources used by their implementations are given below:

- Packet channel: A packet channel provides a conduit for transferring packets between two pipeline stages. The packet channel interface provides methods to enqueue and dequeue packets into the packet channel. The RAL currently provides three packet channel implementations that use different hardware resources: (1) channels that use SRAM (2) channels that use the Hardware-supported packet rings in on-chip scratchpad memory and (3) channels that use next-neighbor registers.

- Locks: The lock interface provides synchronization methods to acquire and release a mutex. The RAL currently provides two lock implementations: (1) locks that use the local memory in an ME to provide synchronization between the threads in an ME, and (2) locks that use a spin lock in scratchpad memory, which is available to all processors.

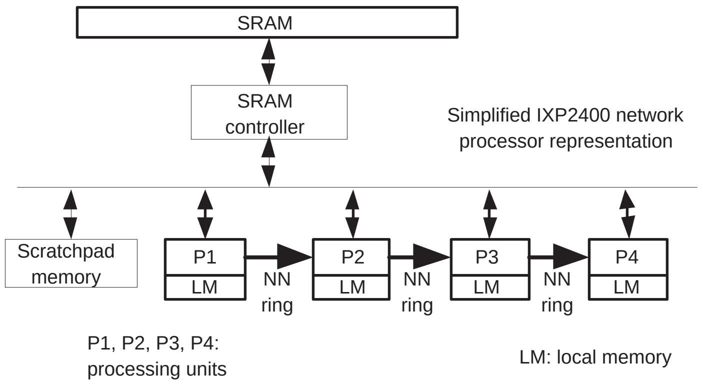
SRAM
SRAM controller
Simplified IXP2400 network processor representation
Scratchpad memory
P1
P2
P3
P4
LM
NN ring
P1, P2, P3, P4: processing units
LM: local memory

Figure 4: Simplified representation of the IXP2400 network processor hardware

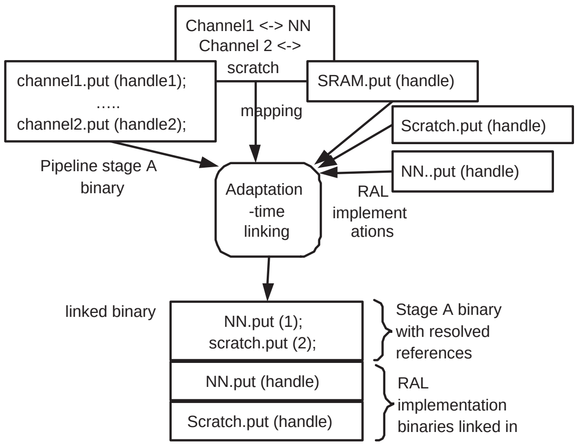

Figure 5: Linking illustrated

3.1.2 Adaptation-time linking

We now describe how a RAL instance in a pipeline stage is bound to a RAL implementation.

Adaptation-time linking (Figure 5) takes as input:

- The binary type, either an ME binary or an Intel XScale® core binary
- A pipeline stage binary containing unresolved references to the RAL methods
- A list of RAL instances used in the binary along with their implementation mapping and methods invoked on each RAL instance and handle values of each RAL instance.
- The RAL implementation binaries

The adaptation-time linking links in the RAL implementation binaries in the pipeline stage binary and resolves the references to the RAL methods in the binary.

The adaptation-time linking also resolves references to RAL handles. A RAL instance handle is introduced to allow sharing of the same RAL implementation code between all of the RAL instances using the RAL implementation. This works as follows: each RAL instance is given a unique instance handle value. This value is passed as an argument to the RAL methods of the RAL interface. The RAL implementation can then distinguish between the multiple instances on a per-call basis. In order to enable binding two RAL instances of the same interface to different RAL implementations, the name of each RAL method is qualified with the RAL instance name in the binary.

3.2 Design/Implementation of state migration

State migration mechanisms are required to change the mapping of a pipeline stage to a processing unit and ensure that none of the application or processor state is lost. The RTS currently only supports a one-to-many mapping of pipeline stages to processing units. State migration entails three operations: checkpointing and data migration, resource state migration, and code migration. These three operations are described in detail in the following sections.

3.2.1 Data migration/Checkpointing

Packet processing code has some unique characteristics that make the design of a data migration mechanism much simpler than general-purpose data migration mechanisms designed before.

First, packet processing code is typically written in a single infinite loop, as diagrammed in the pseudo-code below.

```
while (1) {
    dequeue a packet from an input channel;
    process the packet;
    enqueue packet(s) on output channel;
}
```

The second characteristic of packet processing code is that the loops shown above iterate at very high speeds. This is necessitated by the rates at which packets must be processed in packet processing systems. The final characteristic is that at the beginning of this loop, the packet processing code has no local state. The compiler [30] that we use to compile the application does not allow for thread-local memory, it ensures that the stack is empty of useful information, and does not allow the programmer to acquire locks across loop iterations. These characteristics allow for the creation of a light-weight checkpointing mechanism that can be used to stop packet processing code very efficiently without losing any important state.

When code needs to be stopped, checkpointing is done in two simple steps. First, the RTS tells the running code to stop before starting another iteration of the top-level loop. Second, the run-time system waits until all of the threads running the affected code have stopped at the top of the loop. At this point, the code may be safely stopped without losing any state and restarted on another processor, even if the processor has a different architecture.

Checkpointing on the ME involves stopping the ME execution, writing a thread halt instruction at the checkpoint location (top of the loop) and restarting the ME. This causes all the threads in the ME to halt at a safe location. The ME checkpointing incurs no run-time performance penalty.

On the Intel XScale® core, before starting a new iteration, the Linux kernel thread checks a flag and exits if the flag is set. The RTS sets this flag when it wants to migrate the pipeline stage that is mapped to that thread. The Intel XScale® core checkpointing incurs the run-time performance overhead of checking a flag.

3.2.2 Resource state migration

We implement resource state migration for packet channels. Packet channel state migration is the mechanism used to save the packet handles in a packet channel implementation and initialize a new packet channel implementation with those packets. This mechanism is required to prevent packet resource leakage, when a packet channel instance is remapped from one implementation to another.

A possible solution to packet channel state migration consists of checkpointing the source pipeline stage and allowing the channel to be drained before checkpointing the next pipeline stage that the channel is used to communicate with. This method breaks down if the data-flow graph has loops. Consider an application (Figure 6) with 3 stages A, B and C. This application has a loop between the stages A and B consiting of the channels:channel1 and channel3. If we have to migrate both channel 1 and channel 3, then checkpoint ordering A, B or B, A will not suffice since with either option, there could be a packet channel with packet handles in it after the checkpointing is completed, resulting in leakage of the resources associated with those packet handles. Since data flow graphs with loops occur in some network applications, we need to implement a packet channel state migration mechanism.

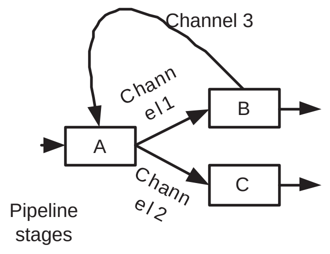

Figure 6: Example application with a loop

We implement channel state migration by first checkpointing both the source and sink pipeline stages and then copying the packet handles from the original packet channel into the new packet channel.

3.2.3 Code migration/Loading code

Migrating code requires associating a processing unit with a pipeline stage binary and starting execution of the code on that processing unit. The RTS provides two types of processing units: (1) Linux kernel threads on the Intel XScale® core (the cardinality of which is bounded by the memory in the system) and (2) ME, of which there are 8 on the IXP2400 network processor, and each ME can have 4 or 8 hardware threads.

Each pipeline stage is associated with two binaries of different formats, one for ME and the other for the Intel XScale® core. This enables RTS to migrate code for a pipeline stage between the two types of processing units as well as between processing units of the same type. The MEs have a fixed-size code that is loaded with the instructions that need to be executed and started by writing an enable register on the ME. On the Intel XScale® core, loading code on a processing unit associates a function entry point for a Linux kernel thread. For the Intel XScale® core, starting a processing unit means creating and starting a Linux kernel thread and passing it the function pointer that was associated with the processing unit in a previous load command.

3.3 Adaptation steps

To explain our adaptation mechanisms, in this section we consider the example application in Figure 7 running on the simplified version of the IXP2400 network processor as shown in Figure 4, and describe the steps performed by the RTS in adapting from one processor mapping configuration to another. A processor mapping configuration consists of the pipeline stage to processor mapping along with the new RAL instance mapping.

Initially (Figure 8), stage A is mapped to processor P1, stage B is mapped to processor P2 and stage C is mapped to processors P3 and P4. Stage A communicates to stage B using a local packet channel implementation (next-neighbor ring) since they are mapped to adjacent processing units

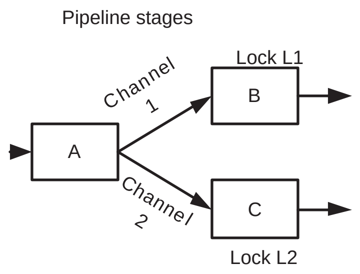

Figure 7: Pipeline stages in an example application

and stage A uses a global packet channel implementation (scratch ring) to communicate with stage C. In the example, stage B is able to use a local lock implementation for L1 since it is mapped to only one processor whereas stage C uses a global lock implementation since it is mapped to multiple processors. In order to migrate the system from the initial to the final mapping (Figure 8) the following changes must be made:

- Stage B migrates from executing on one processor P2 to executing on two processors (P3, P4).
- Stage C migrates from executing on two processors (P3, P4) to executing on (P2) only.
- Channel 1 needs to use a global ring implementation instead of a NN ring implementation.
- Channel 2 can now use a local NN ring implementation instead of a global ring implementation.
- Lock L1 needs to use a global lock implementation instead of a local lock implementation.
- Lock L2 can use a local lock implementation instead of a global lock implementation.

The RTS performs the following steps to adapt from an initial mapping to a final mapping.

1. Checkpointing: In our example, P1, P2, P3, P4 would be checkpointed.
2. Packet channel state migration: In our example, packets in the NN ring implementation of channel 1 are saved and after the new global ring implementation has been allocated, it is initialized with the saved packets.
3. Binding: In our example, in the stage A binary channel 1 is rebinded to use a global ring implementation and channel 2 is rebinded to NN ring implementation. In Stage B binary, channel 1 is rebinded to a global ring implementation and lock L1 is bound to a global lock implementation. Similarly in stage C, channel 2 is rebinded to a NN ring implementation and lock L2 is bound to a local lock implementation.
4. Loading: In our example, processing units P1, P2, P3, P4 are all loaded with the new binary and restarted. An interesting optimization in the loading process would be to determine the order of loading the processing units to minimize packet loss. For example, in the final mapping in our example (Figure 8), pipeline stage B is mapped to two processing units P3, P4, while stage C is mapped to P2 only. In this case, the loading order P3, P2, P4 results in less system disruption time (defined as the time the system is not processing packets) than the loading order P3, P4, P2.

4 Results

Our RTS implementation was done on a RadiSys, Inc. ENP-2611* with a single 600 MHz IXP2400 network processor, running MontaVista* Linux*. Our evaluation was primarily done on this hardware platform with one evaluation done on the cycle accurate simulator for the IXP2400 network processor provided in the Intel®Internet Exchange Architecture (IXA) Software Development Kit (SDK) Workbench version 3.5.

To understand and evaluate our RTS adaptation mechanisms, we used a set of application-agnostic microbenchmarks and application-specific microbenchmarks. We also measured the cumulative effects of the RTS adaptation mechanism using two metrics: total time to adapt and total time a processor is down.

We used a 37.5 MHz hardware-based timer on the IXP2400 network processor to measure the time taken for the different operations. We measured the overhead of invoking the method used to read the timer value as 0.53μs from both Linux user and kernel space.

In the following sections we outline the experimental setup and the results of these studies. The measurements were taken in the Linux kernel on the Intel XScale® core unless otherwise mentioned.

4.1 Application-agnostic microbenchmark

4.1.1 Cost of checkpointing

We used the following metrics to evaluate the checkpoint mechanisms:

- time to inform the processing unit to stop at the beginning of the loop
- time to check if the threads in the processing unit have all stopped execution

The results on both the Intel XScale® core and the ME are shown in Table 3. The ME numbers are larger than the Intel XScale® core numbers because the ME checkpointing consists of stopping the ME, writing a thread halt instruction in the code store at the checkpoint location and restarting the ME execution. The Intel XScale® core checkpointing, on

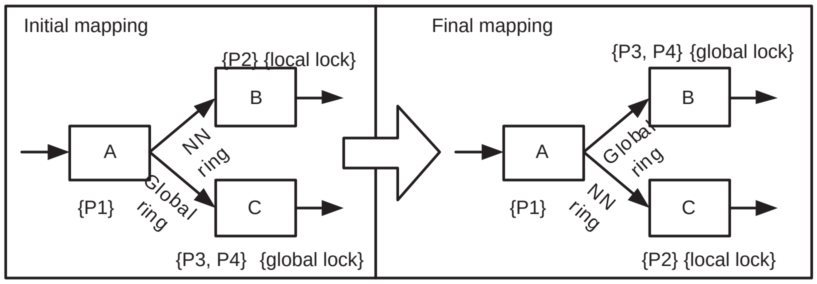

Figure 8: Initial and final mapping

Implementation	inform time (μs)	check time (μs)
ME	60	34
Intel XScale® core	3	3

Table 3: Overhead of checkpointing processing units (Source:Intel)

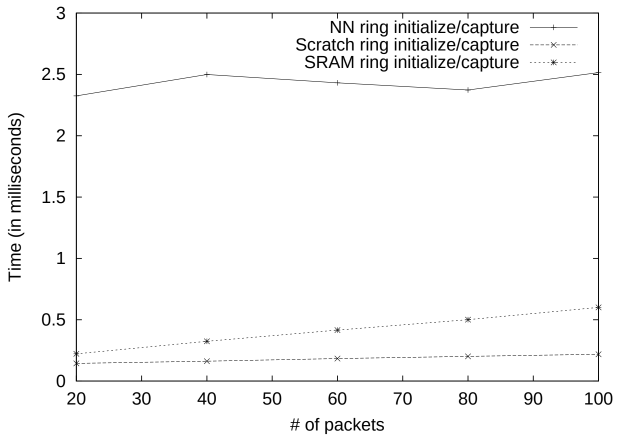

Figure 9: Microbenchmark results for migration of packet channels (Source:Intel)

the other hand, involves just setting a flag in a known memory location. The difference in the ME and Intel XScale® core numbers are due to the overhead of stopping an ME, writing an instruction into the code store and starting the ME again.

4.1.2 Cost of packet channel state migration

We measured the time to save the packet handles from one packet channel implementation and restore them in another implementation. Figure 9 shows the overhead of packet channel migration for the different channel implementations. This figure shows that the overhead of this mechanism is proportional to the number of packet handles saved and restored, as expected.

The numbers for NN ring are high because we need to execute code on an ME to save or initialize the NN ring packet channel. We thus incur the overhead of loading an ME binary image that performs the state migration. The other two packet channel implementations can be migrated from the Intel XScale® core and thus have lower overhead.

4.1.3 Cost of binding

We measured the overhead of the adaptation-time linking approach on both the MEs and the Intel XScale® core. The MEs instructions have no hardware support for relative addressing. Thus ME linking must explicitly relocate the RAL implementation binary and append the instructions in the RAL with the instructions in the pipeline stage binary. On the Intel XScale® core, since the RAL implementations are already loaded as Linux kernel modules, no overhead of relocation of the RAL implementation occurs. The Intel XScale® core adaptation-time linking implementation incurs the overhead of the Linux insmod program, which is used to load the pipeline stage binary into the kernel.

In order to measure the overhead, we generated binaries with varying numbers of:

- call sites that invoke a RAL method. These were equally distributed among a fixed number of RAL instances.
- relocatable instructions (for ME only)

We measured the cost of adaptation-time linking for each binary and the contribution of the various steps involved in the current implementation. The steps are: reading the binary from a memory-mapped filesystem on the Intel XScale® core, relocating RAL implementations (only for ME), patching call sites and writing the linked binary in the filesystem.

Figure 10 shows the results for ME adaptation-time linking as a function of number of call sites. As we can see, the ME adaptation time linking (labeled Total link time) varies linearly with the number of call sites. This is because the overhead in patching call sites varies linearly with the number of call sites. Each call site invoking a RAL method contains a branch instruction with an unresolved target address. The ME binding mechanism patches the branch instruction at each RAL method call site with the instruction address of the correct RAL implementation method that was linked in. The ME linking also incurs a fixed overhead for reading (4.5 ms), writing (3.5 ms) and relocation cost (2.1 ms). The relocation cost consists of the cost of relocating a packet channel implementation and a lock implementation. We also observed that the overhead in ME linking varies linearly with the number of relocatable instructions, but for the sake of brevity we do not show that graph.

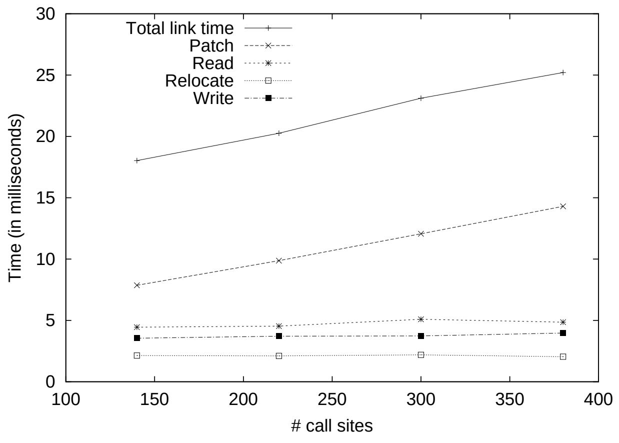

Figure 10: Microbenchmark results for binding code for the ME (Source:Intel)

Figure 11 shows the results of the Intel XScale® core adaptation-time linking. The total link time is around 160 ms for the Intel XScale® core, of which there is a fixed overhead (80 ms) involved in the write operation which uses the insmod utility. The Intel XScale® core patching times are independent of the number of call sites since the Intel XScale® core binding involves renaming the methods in the symbol table which only needs to be done once for each unique method per RAL instance. This is the crucial difference between Intel XScale® core binding and ME binding results.

We also observed that both the ME binding and Intel XScale® core binding times vary linearly with number of RAL instances. This is because both the ME and the Intel XScale® core linking involves patching the handle values for the RAL instance in the binary. We do not show these figures for the sake of brevity.

4.1.4 Cost of loading code

We measured the performance of our loading mechanism on the ME and Intel XScale® core. The Intel XScale® core load involves associating a function entry point with the Linux kernel thread and was measured to be 0.054 ms.

The RAL ME processing unit load method implementation takes as input the pipeline stage binary file name that resides on a memory-mapped filesystem and thus incurs the overhead of reading the binary and accessing the instructions from the binary before writing them into the ME code store. In the experimental setup we generated ME binaries with varying numbers of instructions and measured the

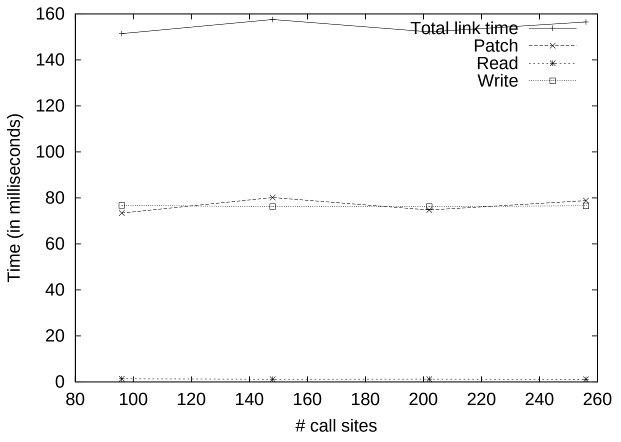

Figure 11: Microbenchmark results for binding code for the Intel XScale® core (Source:Intel)

time to load the binary into the ME code store using the load method of the ME processing unit. The result of this setup is shown in Figure 12. The graph shows the total time taken by ME load as a function of the number of instructions and the contribution of different steps in the implementation: reading the binary, writing the code store and cleanup (freeing resources allocated in reading the binary).

As we can see the result shows that there is a fixed cost overhead incurred in reading the binary (4.0 ms) and cleanup (0.630 ms). The write time into the code store on the ME is proportional to the number of instructions in the binary, as expected.

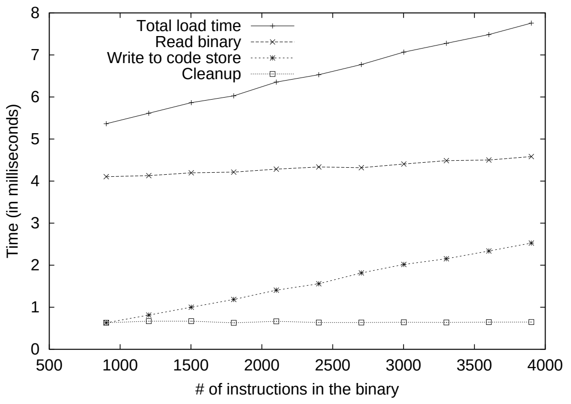

Figure 12: Microbenchmark results for loading code on the ME (Source:Intel)

4.1.5 Cost of starting processing units

We evaluated the costs of starting a processing unit. The IXA SDK provides methods for starting the ME. For the Intel XScale® core, this cost is equal to the cost of creating and starting a new Linux kernel thread. We measured the cost of starting an ME to be 0.036 ms and the cost of creating and starting a Linux kernel thread to be 0.097 ms.

4.2 Application-specific microbenchmark

4.2.1 Function call mechanism overhead

The RAL methods are linked in as function calls and hence incur the overheads of a function call (branch, return and stack setup overheads) for every RAL method that is invoked. We evaluated the run-time performance degradation due to this overhead by comparing the throughput of an application using the RAL methods as function calls with the same application using inlined RAL methods. The application used was the 4Gbps Ethernet IPv4 application (Figure 13) that is shipped as part of the IXA SDK. We evaluated the performance overhead using the IXA SDK Workbench version 3.5.

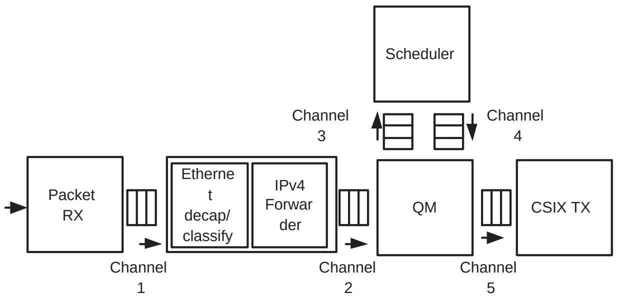

Figure 13: IPv4 IXA SDK application

We ran two configurations for 200,000 cycles in the simulator provided by the IXA SDK Workbench. One configuration had all packet channel RAL methods inlined (this was the unmodified application) and the other was with RAL methods on channel 1 and channel 2 (Figure 13) invoked as function calls. We fed each configuration an input traffic of 64 byte packets at 4Gbps and measured the number of bytes forwarded in 200,000 cycles.

We observed that the configuration with the function call overhead forwarded 100352 bytes compared to the unmodified application that forwarded 116416bytes. This gives us a measure of the performance penalty incurred for processing 64 byte packets at 4Gbps with a function call overhead to be 14%. To validate this result we counted the compute cycles added per RAL method for the function call overhead, to be 20 cycles. Thus the IPv4 pipeline stage incurs a total of 40 extra compute cycles for every packet being processed. The worst case compute cycles of the IPv4 pipeline stage without the overhead is approximately 250 cycles. Thus we could expect a 16% degradation in the performance.

4.3 Cumulative effects of adaptation

We used two metrics to evaluate the cumulative affects of our RTS adaptation mechanisms:

- Total adaptation time: defined as the time taken by the system to reach the final mapping from the initial mapping. This metric is useful to determine how fast the system can adapt.
- Processor downtime: defined as the total time a processor is not processing packets while the system transitions from the initial to the final mapping. This metric gives a measure of the system disruption during adaptation. This metric impacts packet loss and provides important insight into how well the system is able to perform the transition without disrupting processing.

We used the layer 3 switching and forwarding application (Figure 2) implemented using the Baker language. We measured the overhead of adapting between the same processing unit implementations (ME to ME) using the mapping configurations shown in table 5 and the overhead of adapting between processing units of different implementations (ME to Intel XScale® core) as shown in table 4. We only show the pipeline stages whose mapping has changed in the tables.

	L3 fwdr	L2 bridge
Initial mapping	4 MEs	Intel XScale® core
Final mapping	Intel XScale® core	4 MEs

Table 4: Configuration to measure the ME to the Intel XScale® core adaptation overhead

We measured the total time to adapt and the time the processors are down by inserting appropriate timer probes in the code.

Table 6 and Table 7 show the results of this evaluation after running for 5 iterations and averaging them out (the standard deviation for the values was small).

Total time to adapt	254.3 ms
Time proc unit ME 0:3 was down	84.0 ms
Time proc unit ME 0:2 was down	106.9 ms
Time proc unit ME 1:0 was down	61.2 ms
Time proc unit ME 1:1 was down	38.4 ms
Time Intel XScale® core proc unit was down	250.0 ms

Table 6: Results of ME to Intel XScale® core adaptation (Source:Intel)

The processor down times are different for the different processing units because their start times are staggered as shown in the Figure 14. So for the result in Table 7, we can

	L3 fwdr	L2 bridge	Channel 2	Channel 3
Initial mapping	1 ME	3 MEs	sram pkt channel	scratchpad pkt channel
Final mapping	3 MEs	1 ME	scratch pad pkt channel	scratchpad pkt channel

Table 5: Configuration to measure ME to ME adaptation overhead

Total time to adapt	99.5 ms
Time proc unit ME 0:1 was down	96.41 ms
Time proc unit ME 0:2 was down	76.48 ms
Time proc unit ME 1:1 was down	47.30 ms
Time proc unit ME 1:2 was down	25.47 ms

Table 7: Results of ME to ME adaptation (Source:Intel)

see from the Figure, time ME 1:2 was down = (t3 - t1) = 25.47 ms. Similarly time ME 1:1 was down = (t4 - t2) = 47.30 ms.

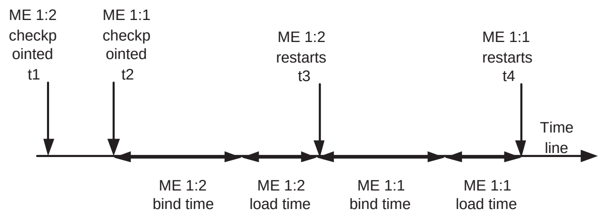

Figure 14: Time line of adaptation steps

5 Discussion

Evaluation of our current prototype shows that the binding mechanism contributes a significant percent of the total adaptation latency. Consider, for instance, a design point where the code deployed on an ME has 200 call sites, and has 2000 instructions, and the associated packet channel has 100 packets in a scratch ring. For this case, adapting the ME takes around 27*ms*. Of this, the time required to stop the ME, load the ME with the new code, and re-start the ME is about 7*ms*; the remaining time is contributed by the binding mechanism.

In our current prototype, the overhead of the binding mechanism affects (1) the duration for which a particular processor is unavailable for processing packets (and hence the amount of packet loss), and (2) the frequency with which processor allocations can be adapted. One can reduce the effect of binding overhead on processor down-time by overlapping the construction of the new binary image with packet processing; a processor can continue to process packets until the new binary image is ready to be loaded. Observe that binding involves patching the handle values for the RAL instances in the binary. Unfortunately, in our current implementation, obtaining handles for RAL instances requires resource allocation to be completed, but this, in turn, requires a processor to be stopped. We are currently exploring ways of obtaining handles to resource instances without performing resource allocation. This will allow us to mask the binding overhead completely; in such a case, the processor down time would be determined by the overhead of taking a checkpoint, load and restart a processing unit.

The above optimization can reduce the duration for which a processor is unavailable for processing packets; however, it does not reduce the adaptation latency. The binding overhead continues to govern the frequency with which processor allocations can be adapted. This overhead exposes a fundamental trade-off between adaptation-time and run-time binding mechanisms. On the one hand, adaptation-time binding yields efficient code that imposes little run-time overhead and produces minimum size binary images; however, it incurs a significant adaptation latency. On the other hand, the use of run-time binding mechanism virtually eliminates the adaptation-time binding overhead; however, it incurs greater run-time overhead and results in larger size binary images.

The relative performance of these two binding mechanisms depends on the system and application characteristics. For instance, as we argued in Section 2.2.1, run-time binding does add additional instruction to process each packet. For a processor with support for hardware multi-threading, the additional computational instructions resulting from run-time binding will have little impact on the packet processing throughput if the total number of computational instructions executed by all threads between successive memory accesses are insufficient to hide memory access latency; in this case, addition of computation instructions only reduces processor stall and has little effect on the packet processing throughput. Similarly, the loss in throughput resulting from run-time binding should be evaluated relative to the drawbacks—of reducing the frequency of adaptation and dropping a set of packets during adaptation—of the adaptation-time binding mechanism. Finally, run-time binding may be better suited for processors that support instruction caches (e.g., the Intel XScale® core on the IXP2400 network processor), have relatively large instruction stores (such that more than one resource instance could be pre-loaded), or process packets with a lower throughput requirement.

Since the choice of the mechanism depends on several factors, we argue that mechanism selection is a policy issue. To provide the flexibility of choosing the right mechanism, we plan to implement the run-time binding design into our system.

6 Related Work

With the advent of specialized multiprocessor hardware for supporting packet processing applications, several research efforts have developed tools and systems to make the hardware easily programmable, and achieve high throughput [1, 2, 3, 4, 11, 19, 28, 29, 30]. Most of these systems allocate resources in the multiprocessor system to pipeline stages of an application statically (at design time). In this paper, we develop a core set of mechanisms required to adapt resources to stages at run time. Our work is motivated by the observation that network traffic fluctuates significantly [18, 26, 32]. In [18], we show that adapting processor allocations to stages at run time can reduce the provisioning level of a packet processing system, and can make the system robust to traffic fluctuations.

In this paper, we build and study the checkpointing, state migration, and dynamic binding mechanisms required to adapt resources in a multiprocessor system to application stages at run-time. Many past systems provide checkpointing and state migration [12, 20, 21, 24, 27], and dynamic binding mechanisms [7, 9, 25] in the general-purpose application domain. However, as discussed in Section 3, our implementation exploits the unique characteristics of the packet processing domain to achieve efficiency. For instance, by exploiting the loop nature of packet processing applications, we reduced the overhead of checkpointing and state-migration significantly. Similarly, given the requirement to support high throughput and the constraints on instruction store sizes, we explore the benefits and tradeoffs of using adaptation-time binding mechanism.

NetBind [19] and VERA [29] allow dynamic allocation of resources in order to support extensibility of network services [5, 8, 16]. NetBind enables dynamic creation of packet processing pipelines through the dynamic binding of small pieces of machine language code. VERA focuses on making a router consisting of a PC with a host processor and a few network processors (1) extensible by allowing dynamic installation of new functionality, and (2) efficient by offloading the most frequently executed packet processing functions to network processors. Such offloading allows VERA to support router extensions on the host processor. Neither NetBind nor VERA address the problem of adapting processor allocations at run-time to maximize the throughput of the system in the presence of traffic fluctuations.

7 Conclusion

Implementors of packet-processing applications on multicore processors must balance two requirements: (1) adapt processor allocations dynamically to reduce the overall resource provisioning requirement for the system, to achieve robustness to traffic fluctuations, and to reduce energy consumption; and (2) utilize for each application stage resources (e.g., memory levels, inter-processor communication mechanisms, etc.) closer or local to the processors on which the stages are mapped to achieve the highest possible throughput. In this paper, we describe the design and implementation of a run-time adaptation system that can meet these two requirements simultaneously. Our design allows each application stage to utilize local resources whenever possible in the steady state. Upon adapting the allocation of processors to stages, the run-time system (1) binds each resource usage within a stage to a new resource instance; and (2) checkpoints and migrates the state from the previous resource instance to the newly-bound resource instance. We describe the design and implementation of our adaptation system in the context of a packet processing system designed using the Intel IXP2400 network processor. We show that our design has little impact (14%) on the steady-state throughput of the system. We further show that our design is able to perform resource adaptation for a real application in less than 100ms, allowing processor allocations to be adapted at a very fine time-scale. Finally, we discuss several optimizations for reducing the adaptation overhead even further.

References

[1] CloudShield Technologies. `http://www.cloudshield.com`.

[2] Payloadplus family of network processors. http://www.agere.com/enterprise_metro_access/network_processors.html.

[3] TejaNPTM: A Software Platform for Network Processors. http://www.teja.com.

[4] M. Adiletta, D. Hooper, and M. Wilde. Packet Over SONET: Achieving 10 Gigabit/sec Packet Processing with IXP2800. *Intel Technology Journal*, 6(3), 2002.

[5] A. T. Campbell, H. D. Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. Villela. The genesis kernel: A virtual network operating system for spawning network architectures. In *2nd IEEE International Conference on Open Architectures and Network Programming*, 1999.

[6] D. Comer. *Network Systems Design Using Network Processors*. Prentice Hall, ISBN 0-13-141792-4, 2002.

[7] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 108. IEEE Computer Society, 1996.

[0] Intel XScale®, Intel® IXP2400 is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

[8] D. Decasper, Z. Dittia, G. M. Parulkar, and B. Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of ACM SIGCOMM*, 1998.

[9] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[10] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL Protocol Version 3.0. Internet Draft, November 1996.

[11] L. George and M. Blume. Taming the ixp network processor. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 26–37. ACM Press, 2003.

[12] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 33(5), 1999.

[13] S. Harizopoulos and A. Ailamak. A Case for Staged Database Systems. In *Proceedings of 1st Conference on Innovative Data Systems Research*, 2003.

[14] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1), 1991.

[15] J.Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *Proceedings of USENIX Annual Technical Conference*, 2002.

[16] R. Keller, L. Ruf, A. Guindehi, and B. Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *Proceedings of Fourth Annual International Working Conference on Active Networks (IWAN)*, 2002.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[18] R. Kokku, T. Riche, A. Kunze, J. Mudigonda, J. Jason, and H. Vin. A case for run-time adaptation in packet processing systems. *ACM SIGCOMM Computer Communication Review*, 34(1):107–112, January 2004.

[19] M. E. Kounavis, A. T. Campbell, S. T. Chou, and J. Vicente. Programming the Data Path in Network Processor-Based Routers. *Software Practice and Experience*, Special Issue on Software for Network Processors, 2004.

[20] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, 1990.

[21] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[22] R. Love. *Linux Kernel Development*. Sams Publishing, 800 East 96th Street, Indianapolis, IN, 2004.

[23] U. Naik and P. Chandra. *IXP2400/2800 Application Design*. Intel Press, To be published.

[24] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.

[25] P. Pardyak and B. N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, 1996.

[26] Y. Qiao, J. Skicewicz, and P. Dinda. Multiscale Predictability of Network Traffic. Northwestern University. Technical report.

[27] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

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

[29] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[30] H. Vin, J. Mudigonda, J. Jason, E. J. Johnson, R. Ju, A. Kunze, and R. Lian. A programming environment for packet-processing systems: Design considerations. In *3rd Workshop on Network Processors and Applications*, February 2004.

[31] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[32] Z. Zhang, V. Ribeiro, S. Moon, and C. Diot. Small-Time Scaling Behaviors of Internet Backbone Traffic: An Empirical Study. In *Proceedings of the IEEE INFOCOM.*, 2003.