Safe Execution of Untrusted Applications on Embedded Network Processors

Herbert Bos, Bart Samwel, Mihai Cristea
Leiden Institute of Advanced Computer Science
Leiden University
{herbertb,bsamwel,cristea}@liacs.nl

Kostas G. Anagnostakis
Dept. of Computer and Information Science
University of Pennsylvania
anagnost@dsl.cis.upenn.edu

Abstract

Controlling the function of embedded network processor systems has so far been confined to simple configuration languages and component models, with the full programming capabilities available only to trusted system-level programmers. In this paper, we consider a software architecture enabling the safe execution of untrusted code on the IXP1200 family of embedded network processors. We extend known techniques used in extensible operating system kernels, adapting them to the particular characteristics of the network processing domain. The result is a restricted execution model trading off some flexibility for robustness, yet enabling a wide range of low-level applications or enhancements not presently possible. Experiments with applications in network monitoring show that the price of safety in such an environment is comparable to overheads observed in providing equivalent functionality on general-purpose processor systems.

1 Introduction

The continuing evolution of networking technology has resulted in a growing need for network appliances with higher performance than provided by general-purpose workstations and more flexibility than provided by application-specific integrated circuits (ASICs). This has resulted in the development of a broad class of system architectures referred to as network processors. Although network processors take different forms depending on the expected needs of packet processing applications, a typical design involves a fair number of fully-programmable parallel processing units, along with a variety of hardware assists to accelerate specific network computations. The various processing elements on the network processor are usually coupled with a general-purpose processor to form a processing hierarchy[11]. The designer of a high-performance network appliance then develops appropriate application firmware for the network processor and software for the general-purpose processor in this hierarchy. Besides offering high performance, the system can easily be updated (for new functionality or bug-fixes) without the need to redesign new circuits.

Although standard, vendor-implemented functionality is usually sufficient, in some cases it is necessary for the network operator (or even the customers) to manipulate the network processor function, beyond what can be provided through a simple configuration language. Providing a robust general-purpose programming environment on a network processor, similar to what is offered by the operating system on a general-purpose system, is not currently possible: existing work either restricts flexibility to the higher levels of the processing hierarchy only[11], or assumes that code provided by users can be expected to be correct and can thus be safely executed on the network processor without further precautions[5].

In this paper we address the problem of designing a general-purpose programming environment across all layers in a network processing hierarchy. In previous work, we have shown how the open kernel environment (*OKE*) provides a safe, resource-controlled programming environment which allows fully optimised object code to be safely executed in a Linux kernel by non-privileged users [4]. This paper discusses a set of extensions to the *OKE* for providing similar functionality at the network processor level. Specifically, we show that a lightweight *OKE*, known as *Diet-OKE*, can be used to allow resource sharing on the micro-engines of the IXP1200 network processor [7]. In the *Diet-OKE*, users can load new applications anywhere in the hierarchy where there is a general purpose processor running an operating system like embedded Linux, or as plug-ins in specific 'slots' in application frameworks for those levels that consist of embedded processing elements. Although this work is centered around the architecture of a specific network processor, we expect that the same set of techniques can be adapted to construct similar architectures for other network processors as well.

The rest of this paper is organised as follows. In Section 2 we present the overall architecture and the issues that need to be addressed to safely execute user code in the lower levels of a processing hierarchy, with particular emphasis on embedded network processors. We also discuss how trust management [3] coupled with a trusted compiler can be used to control the use of shared resources at any level in the processing hierarchy. Section 3 addresses in more detail how these concepts apply to the IXP1200 family of network processors. In Section 4 an example is given of an application framework for special-purpose processing engines on the IXP1200 and it is shown how new applications can be plugged in this framework. A set of simple experiments are presented in Section 5. Finally, in Section 6, conclusions are drawn.

2 Architecture and implementation

Processing in network appliances such as routers and firewalls is typically performed by a processing hierarchy structured similarly to the architecture shown in Figure 1. The specific processing hierarchy shown in the Figure uses the Intel IXP1200 network processor, which we discuss in more detail in Section 2.4. The processing hierarchy here consists of five distinct layers:

L0: software components running on the NP's parallel processing elements (called microengines, or μ-engines);

L1: kernel code running on the NP control processor;

L2: user-space processes on the NP control processor;

L3: kernel modules running on the host CPU;

L4: user-space processes running on the host CPU.

The allocation of functions to different levels of the processing hierarchy is based on the idea that for achieving high performance the "common case" should be dealt with at the lowest possible level, and exceptional, irregular and control tasks should be deferred to higher levels. Beyond the performance-related considerations, the lower levels are much harder to program, while also having limited resources (for example, instruction store on the IXP1200).

While the architecture includes all processing levels, this paper considers only the lowest levels in the hierarchy. Observe that once programs are loaded in any level below $L4$, there is no default (software or hardware) support to enforce 'safety'. For example, privileged instructions are no longer guaranteed to be privileged and memory management units (MMUs) no longer ensure that a party only accesses its own address space. So, if multiple parties are allowed to program these levels *directly*, i.e. using fully optimised native code, other mechanisms are needed to enforce safety by means of resource control. As an example, we discuss code running in a NP or kernel. After we have sketched the principles, we will explain in some detail the mechanisms in the subsequent sections.

2.1 The open kernel environment

The problem of loading code in embedded NPs is in many ways similar to that of loading code in the kernel of an operating system. It is a sensitive operation that imposes risks to the safe and robust operation of the system, and is therefore typically restricted to privileged users, assuming that those users will only provide safe and correct code. From a performance perspective, it would be useful to avoid such restrictions, as this would eliminate overheads of crossing protection boundaries in the case of operating system kernels, or the cost of host-NP communication in the case of network processing hierarchies. Although it is possible to design restricted application-specific languages such as, for instance, the interpreted filter language used by BSD packet filters [10], a system that gives emphasis to generality would rather set the level of abstraction at the level of general-purpose object code.

The *OKE* replaces the hard barrier that prevents users from loading code in the kernel or NP with more fine-grained controls on what such code is allowed to do. In the *OKE*, trust management is used to determine the privileges of user and code. Privileges are expressed in the form of *credentials*. These credentials are used by a trusted compiler to determine the constraints to be enforced for a particular module. We will briefly describe the *OKE*'s two main components: the code loader (CL), and the trusted compiler; we refer to [4] for a full description of the system.[1]

To run code in the kernel or on the NP, a user submits object code and the associated credentials to the CL. If the credentials match the code and the security policy, the CL loads the code (Figure 2). The security policy is provided to the CL at start-up.

Most NP vendors provide 'tailored C compilers' for their devices, enabling us to write code in C or any alternative high-level language for which a translator to C exists. We chose to adopt an alternative C-like language that interfaces easily to the rest of the environment (i.e. NP or kernel) and which we modified in such a way that, depending on the client's privileges, more or less access is given to resources, APIs and data, and/or

[1] A complete version of the *OKE* for the Linux kernel is available for download from `http://www.liacs.nl/~herbertb/projects/oke/`.

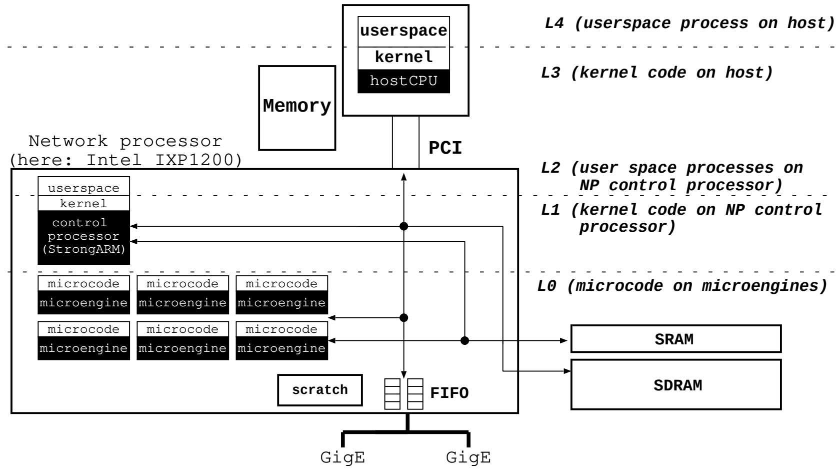

Figure 1. Architecture (the IXP1200 is used as an example)

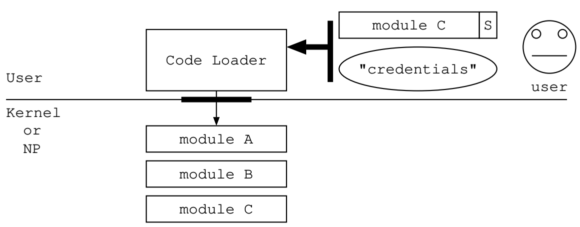

Figure 2. User loads module in the kernel

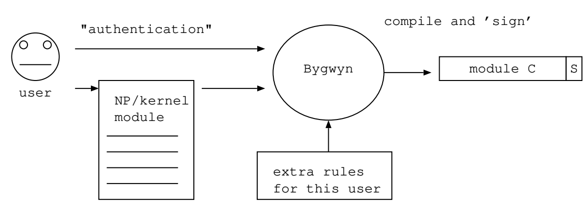

Figure 3. User compiles kernel module

more or less runtime overhead is incurred. The language we use is *Cyclone*, a crash-free language derived from C which ensures safe use of pointers and arrays, offers fast, region-based memory protection, and inserts few runtime checks [8]. However, for true safety and speed, using Cyclone was not sufficient and we extended it in various ways as discussed below. For programs running on the μ-engines, the compiler will generate μ-engine C which is subsequently compiled by the μ-engine C compiler.

The key idea is that restrictions, called *customisations*, are applied to a user module depending on that user's credentials (see Figure 3). Customisations are determined by customisation types which have unique identifiers, called *customisation type identifiers*. As an example, a customisation type may specify that the code is given a specific (safe) API and also some upperbounds on the amount of processing time and memory it is allowed to use. The amount of memory may still be configurable and vary from user to user, but it will always be less than the specified upper bound. In other words, a customisation type defines the broad category in which the restrictions fall, while the instantiation of the type applies the specific configuration parameters for a certain user. The code is bound to the customisation type by means of an unforgeable compilation record which should match the code and the credentials that are later presented by the user to the CL. Once loaded, the code runs natively at full speed. Depending on the users' credentials, they get access to other parts of the NP or the kernel via an API containing the routines which they may call. The routines are linked with the user's code. In other words, the API is used to *encapsulate* the rest of the NP/kernel.

The application of the trust mechanism is not limited to NP or kernel modules: it fits any situation where code must be restricted at compile time to comply with a policy. In the remainder of this section, we will show what the challenges are for implementing policies at compile time, and we will show how we can solve these challenges for the different parts of the processing hierarchy.

2.2 Implementing a policy at compile-time

Given the trust mechanism, we still need to ensure that code complies with a given safety policy at compile time. The following issues must be addressed if we want to enable users to run applications in any environment in a safe manner:

1. memory protection in the spatial domain (bounds checking);

2. memory protection in the temporal domain (references to freed memory);
3. stack overrun protection;
4. processing time restriction;
5. API restrictions;
6. hiding of sensitive data;
7. removing/disabling misbehaving code.

The compiler implements restrictions by prepending (prior to compilation) the appropriate *Environment Setup Code* (ESC) to the code submitted by the user. The ESC instructs the compiler on how to restrict the code for achieving the following goals:

1. remove dangerous constructs from the programming language;
2. remove the ability to import APIs;
3. remove access to certain namespaces.
4. lock certain fields of data structures so that they can neither be read nor written;
5. wrap all entry points of the code with custom wrapping code.

In addition, the compiler statically analyses stack usage and, if necessary, inserts run-time checks to ensure that applications do not overrun their stack bounds.

The mechanisms of the OKE-Cyclone compiler, combined with the basic safety properties of the Cyclone language, are used by the ESC to build a restricted environment for the application.The general structure of Environment Setup Code can be described as a *positive-negative* structure. First, a set of APIs and support structures is declared (or included) with a set of wrappers controlling all direct accesses to critical functions (the positive part). Second, the APIs and critical language constructs that the user should not use directly are hidden (the negative part). After the ESC has been processed, an application is left with a safe API and just enough rope to hang itself, but not enough rope to hang the whole target environment.

2.3 Sharing embedded network processors

Embedded systems often concern special-purpose hardware that is cheap and fast at performing a small number of functions. There is neither the possibility, nor the need to share such systems. Network processors, on the other hand, are representative of an entirely different family of embedded systems. Although these processors are tailored to the specific needs of packet processing applications, they are fully programmable parallel machines capable of implementing a wide range of functions. The basic premise of network processors is a better performance-flexibility trade-off compared to choosing between an ASIC and a general-purpose CPU.

Of particular interest in this context is the monitoring platform envisioned in the SCAMPI project [6]. The goal is to provide a generic "monitoring appliance" enabling multiple applications to gather statistics on network links of at least 10 Gbit/s. The envisioned system should allow potentially untrusted hosts to run monitoring code while ensuring that this code cannot crash the system or access sensitive data. For example, one module could monitor network performance for supporting large-scale distributed computations ("Grid" environments), another one could monitor the network for potential denial of service attacks (e.g. TCP SYN packets), and another module scans a subset of the packets for potentially harmful content (e.g. `#!/bin/perl` in the payload, which may indicate a hacker).

While it is possible to implement such an extensible monitoring system using "off-the-shelf" components, the processing capabilities available to applications on a general-purpose architecture restrict its scope to roughly 1 Gbit/s rates, while also providing a somewhat limited processing budget [1].

Similarly, other applications may need to perform transcoding on (possibly overlapping) subsets of the traffic, set differentiated services [2] bits in the IP header, or add forward error correction code to the payload of some packets. In this model, application providers are 'clients' of the hosting operator. Due to the high data rates, it would be difficult to support these applications on a general-purpose architecture.

Note that in many cases, both in network processing but also in the more general context of embedded systems, there is neither the possibility nor the need to provide a programming interface to untrusted applications, or to offer a multi-programming environment. Our work may be less relevant in these cases. However, in cases such as network monitoring, this is a reasonable (and challenging!) engineering goal.

2.4 IXP1200 network processor: architecture

The basic architectural components of the Intel IXP1200 NP are shown in Figure 1. The NP includes a StrongARM control processor (running embedded Linux) and 6 μ-engines, which are highly optimised processing engines for fast packet processing. Code running on the μ-engines consists of low-level (assembly-like) code, not supported by an OS. Each μ-engine runs at 200 MHz contains 4 hardware contexts (threads) implementing a form of simultaneous multithreading[12], a fairly extensive set of registers, and a small instruction store (1K instructions). Packets are received from the MACs in 64 byte chunks, knows as mpackets and transferred into a set of buffers (capable of storing 16 mpackets) over a proprietary bus (the IX bus). The micro-engines and StrongARM interface to SRAM, SDRAM and a small amount of on-board scratch memory. Although many different configurations are possible, the IXP used in this work was provided with two 1 Gigabit Ethernet MACs for packet reception. More recent version of the IXP contain more micro-engines each of which supports more threads. For example, the latest model, known as IXP2800, contains 16 micro-engines with support for 8 threads each, instruction stores of 4K, and additional supporting hardware units. By interfacing to 10Gbps Ethernet, it surpasses the link rate of the IXP1200 by approximately an order of magnitude.

3 Sharing resources on the IXP1200

The necessary mechanisms for safe programming of an operating system kernel are provided by the *OKE* as described in [4]. As the StrongARM processor on the IXP is capable of running a commodity operating system such as Linux, the techniques used in the *OKE* apply without further modifications. The challenging part is enabling safe programming on the μ-engines: the μ-engines do not run an operating system and do not provide support for common programming concepts found in general-purpose architectures. For instance, the μ-engines do not support stacks or recursion, so the stack overrun problem does not exist. Therefore, some of the *OKE* functions become redundant. On the other hand, μ-engine safety requires issues such as processing bounds to be addressed in a different way, and some issues are unique at this level.

The resulting modified *OKE* used on the μ-engines is known as *Diet-OKE* , and we sketch solutions for each of the remaining problems listed in Section 2.2.2. Although there is currently no full compiler support for *Diet-OKE*, all mechanisms have been implemented and tested in isolation. The basis of our solutions lies in an OKE-Cyclone compiler generating *μ-engine C*, a dialect of the C programming language with μ-engine programming extensions. As the current OKE-Cyclone compiler produces ANSI C, the necessary translation to μ-engine C is relatively straightforward. As the storage class for all variables can be explicitly specified in μ-engine C, the OKE-Cyclone compiler therefore needs to be extended with storage class specifiers. Another issue with μ-engine C is the presence of many intrinsic functions in μ-engine C, but these intrinsics have the same appearance as function calls and are therefore easy to support. Applying API restrictions for μ-engines is in no way different from API restriction in the regular OKE. Hiding of sensitive data in OKE-Cyclone is handled using the `locked` type modifier, and this mechanism can also be used unmodified for hiding parts of data-structures in μ-engine programs. The `locked` and `const` modifiers provide the tools for detailed access control performed fully at compile-time.

3.1 Handling misbehaving code

Diet-OKE attempts to catch potential problems in the code as early as possible, e.g., at compile time. However, there will always be situations that cannot be caught statically and therefore require the insertion of run time checks. Code misbehaviour comes in various forms, e.g., memory access violations, CPU time violations, API violations, and deadlock. Violations can be detected either within the μ-engine code itself (e.g., memory access violations and API violations), or from the outside (processing time violation, deadlock). There are multiple possible responses to violations. For instance, one can attempt to fix the situation that occurred as it occurs, or, alternatively, one may simply terminate the application. Fixing errors usually costs extra instructions, and we prefer not to spend cycles and valuable instruction store space on fixing errors caused by incorrect code. Therefore, *Diet-OKE* simply terminates any application that caused violations.

Terminating an application running on a μ-engine is handled by the IXP's StrongARM core. The software on the StrongARM is able to stop an individual μ-engine and remove the application that was running on that μ-engine, making the μ-engine available for a new application. If a violation is detected from software running on the StrongARM, the violating application can be terminated immediately. When a μ-engine detects a violation, it notifies the StrongARM of the violation and then leaves it to the StrongARM to take action. When a violation is detected by instrumentation code inside an application itself, e.g. by a bounds check inserted by OKE-Cyclone, the application notifies the StrongARM of the violation and then goes into an infinite loop, awaiting death.

One last issue with terminating an application is that it may be holding locks on shared resources. To release these locks, one needs to maintain lock ownership information. Each global lock structure therefore needs to include a field identifying the μ-engine that owns it. To make this operation atomic, we use the hardware-supported *CAM locks* associated with the IXP SRAM. These locks are fairly efficient: they are locks on memory addresses in SRAM that do not block future reads on the SRAM but that only block other CAM lock requests on the same address. The CAM unlock operation also allows an atomic *write* to the memory address to be unlocked. These two operations are sufficient for managing locks on shared resources as needed by our framework.

3.2 Enforcing processing constraints

The processing constraints that need to be enforced on μ-engines are different from the StrongARM and the general-purpose CPU. In the latter, the processor is *shared* between applications, requiring the use of a timer facility, or embedding guard instructions within program loops for determining if an application has exceeded its allocated processing share. In the former, applications are isolated on different μ-engines, and such checks are not necessary.

The only consequence of an application spending too much time processing a packet is that the packet needs to be buffered until the application signals completion and releases the buffer. When an application performs extensive processing, or runs into an infinite loop, buffer space will not be released. This would not be a concern if we could restrict applications to process only one packet at a time, as the number of buffers can be configured to exceed the number of applications. Some applications may, however, require buffering multiple packets before computing a result.

The system therefore needs to consider whether an application has exceeded a (policy-controlled) time budget *for a particular packet*. To address this problem, we opted for embedding the timeout checking in the receive loop of the application support framework. This approach is straightforward and at the same time efficient. Applications running on μ-engines are forced to set a bit signalling that they are done with a packet. If an application does not provide this signal, the timing violation will result in the application being terminated.

3.3 Packet access

Packets are read into a μ-engine by reading them from the RFIFO (the μ-engine hardware receive buffer) into SDRAM in 64 byte chunks (mpackets). A prefiltering function determines whether or not an application should be given access to a packet header and if so, what sort of access (read or write). Illegal access is prevented by only supplying packets to an application after prefiltering has been done. Access to specific parts of a packet can be guarded at compile time using a combination of access APIs and *locked* fields.

3.4 Memory access

Ensuring access to memory locations is safe is the most complex feature of *Diet-OKE*. Any implementation of memory access protection in the spatial domain requires some form of bounds checking which is complicated by the fact that IXP μ-engines have no hardware support for memory management. Therefore, we are forced to implement the checks in the software, at the cost of expensive processing cycles. The Cyclone language has built-in support for bounds checking with varying levels of flexibility and speed. Analysis of the code generated by the compiler has shown that some of the bounds checks can be eliminated through aggressive optimisation. In addition, the Cyclone language has facilities that allow for factoring out bounds checks from code blocks, effectively performing them only once in advance instead of every time an array is referenced. The combination of these two factors alleviates the pressure that the bounds checks will put on the processing cycle budget of the applications.

Memory protection in the temporal domain required the use of a garbage collector in the OKE implementation for the Linux kernel. It is clear that this cannot be done for μ-engines. Instead, the protection is hidden behind APIs. We expect that in most cases it will be sufficient for applications to preallocate fixed chunks of memory on start-up. This memory remains allocated throughout the lifetime of the applications. Naturally, as memory is a shared resource, the amount an application can preallocate would be controlled by policy constraints.

4 Application framework

We have implemented an application framework for packet processing on the IXP that allows up to five untrusted applications to share the μ-engines and perform independent network processing functions. It should be mentioned that this application framework is only an example, tailored to the task of network monitoring, assuming multiple monitoring applications loaded by different users on the NP.

The framework dedicates one μ-engine on the IXP to the reception of packets, while the other μ-engines run the untrusted processing applications[2]. A μ-engine that transmits packets is not included in the framework, but transmissions can be implemented by a consuming μ-engine. As such, the framework without a transmitter is mostly useful for monitoring of packets coming out of a splitter for that purpose only (i.e., that need not be transmitted).

The application μ-engines consume the packets asyn-

[2] Although more engines could be dedicated to packet reception, this leaves fewer μ-engines for applications, while incurring more synchronisation overhead. Also, when receiving data from only a single port, it is the port latency that is the limiting factor, not the number of μ-engines used.

chronously, allowing applications with highly varying packet consumption speeds and per-packet speed variations to work together without any problems. For example, one application might want to process almost every packet but for only a very short time, while another application might want to process only every one in a thousand packets but for 500 times as long. Situations such as these are handled gracefully by this framework.

In the remainder of this Section we will describe the data structures used to communicate between the μ-engines in the framework, the activities performed by the μ-engine that does the receiving, and the structure of the applications on the other μ-engines.

4.1 Framework data structures

The data structure used to communicate between the receiver μ-engine and the applications is essentially a circular buffer consisting of slots that can contain one packet each. Because of the properties of the different storage interfaces of the IXP, the buffer is split over SRAM memory (which has relatively low latency and supports synchronisation primitives) and SDRAM memory (which can be used to receive packets). The SRAM memory contains the bookkeeping information, while the SDRAM contains the actual packet data. Every slot in the bookkeeping buffer has a corresponding fixed-size slot in the SDRAM buffer. Every bookkeeping slot contains fields for the following information:

- The number of bytes already received in the packet.
- A flag indicating whether the packet is complete.
- Five flags indicating whether application n (n = 1,2,3,4,5) has finished processing the packet.
- A read lock for each μ-engine and a single write lock.

4.2 Microengine activities

The receiver μ-engine adds packets to the buffer as it receives them. It instructs the IXP's FBI (FIFO Bus Interface) to place the received data in the SDRAM slot associated with the bookkeeping slot. When the data has been received, it updates the bookkeeping information in SRAM to reflect the reception of new data. The receiver μ-engine can use multiple threads to receive up to four mpackets simultaneously. Our measurements have shown that a configuration with only two threads gives the highest throughput. This can be explained from the fact that the transfer of mpacket data from the MAC to the RFIFO (the IXP's internal receive buffer) has to be serialised, and at a certain point the addition of more threads just leads to a higher synchronisation overhead.

The application μ-engines run a loop that consumes packets from the buffer. Their reading position may be very close to the packet the receiver μ-engine is currently receiving (actually, they may already read the packet as it is still being received), but they may also lag behind a bit. Depending on the application, the μ-engines may process multiple packets at the same time in multiple threads, and the threading functionality is used to overlap the processing of multiple packets as much as is possible without sacrificing safety.

The receiver μ-engine has one additional task, which consists of "mopping up" the packet slots behind the receiving applications. The cleaning code cleans up one packet slot for every packet received, so it keeps an exact distance in packets from the head of the buffer. The purpose of this task is to limit the amount of packets that a task may lag behind, because buffer space is limited and slots eventually have to be reused. When a packet slot is mopped up, it is possible to detect that an application has not finished processing the packet (from the flag in the bookkeeping information), and when this is the case, this is regarded as an application misbehaviour, and the application is immediately terminated.

4.3 Application structure

In the specific application framework, an entire μ-engine is allocated for each application. Extending this framework for an application to allocate multiple μ-engines is straightforward. The user can decide how the resources (e.g. threads) of the μ-engine need to be utilized for the particular application. A user typically dedicates some threads to packet processing, and others to auxiliary tasks (such as performing maintenance work on module state). The structure of auxiliary threads is generally left to the application, within the safety limits determined by the ESC. The control of the packet processing thread is however fully in the hands of the ESC: the ESC provides *slots* where users can plug application-specific code.

A typical packet processing thread consists of a loop that processes one packet per iteration. Processing in the loop can take different forms depending on what part of the packet needs to be inspected, and the type of access (read or write) that is needed by the application. For instance, if an application is interested only in IP headers these lie entirely in the first mpacket and the application does not have to wait for additional mpackets to be received. The current application framework is optimized for read access, assuming that reading is more common than writing on packet data (which is the case in monitoring applications). Obtaining a read lock is done by setting a bit in the control structure. Writers have to set the global write lock and obtain *all* of the read locks.

The IXP provides atomic test-and-set operations. It is obvious that readers are given preference and whenever any application is active reading, the writer keeps trying until the read lock becomes available.

The packet processing thread structure is specified by the application programmer. This is passed to the trusted compiler as a policy statement which causes the ESC to be parameterised for different applications (as discussed in Section 2.1). The ESC then generates a packet consumption skeleton for the specific policy as the main loop for the packet processing threads and declares a number of prototypes for functions that the application must implement. These are the 'slots' referred to in the Introduction[3]. The application programmer implements these functions to specify what actions to take in response to packet reception, and the compiler inlines the code provided in the packet consumption loop.

5 Experiments

We measure the performance of the application framework and the impact of run-time checks on the performance of four toy applications. All measurements reported are obtained on the Intel IXP simulator, providing a cycle-accurate simulation of the IXP1200 hardware. Our experiments assume 200 MHz μ-engines, as provided by the IXP1200 board in our test-bed.

For determining the performance of the packet reception framework, we first measure the maximum loss-free rate achieved as a function of packet size. The results are shown in Figure 4. As expected, the throughput is lower for small packets because of per-packet overheads, but the throughput is generally between 600 and 700 Mbit/sec (for a theoretical maximum rate of 1 Gbit/s). This is approximately the same as reported by Intel in [9].

Next, we implemented four example packet processing applications that serve as the basis of our tests. The applications perform the following tasks:

- TCP SYN flood detection: detect TCP packets with the SYN flag set and increase a counter if one is encountered. This can be used for implementing the approach of [13]

- UDP packet counting: counting of the number of packets destined for either Id Software (Doom) or Sun RPC ports.

- Intrusion detection: primitive content-matching by searching for the string "`/bin/`" in bytes 2-18 of the payload of TCP packets (which may indicate a intrusion attempt).

- Marking for Differentiated Services (DiffServ[2]): setting the DiffServ field of an IP packet to a specific value if the packet originated from a specific IP address.

For emulating the safety checks of *Diet-OKE*, we added hand-crafted run-time bounds checks to the original code in the same way they would have been added by an OKE-Cyclone compiler. The application most heavily affected by this is the "/bin" detection application, because it contains a loop over an array of characters. However, the other applications are also affected because they contain a check on whether the header fields they reference lie within the bounds of the packet.

As shown in Figure 5, we have measured the packet latency for each of the applications with checks as well as without checks. The applications turned out to run most effectively when they ran with only two threads per μ-engine, due to overloading of the SRAM bus with synchronisation activity in the presence of more threads. The reason this happens is because of the (admittedly inefficient) synchronisation mechanism of the current application framework where each processing thread is actually *polling* the next packet slot it has to process for the presence of a new packet. Every measurement on the applications was done twice, once with the other applications running as well on other μ-engines, and once with only one application running. The results are shown in Figure 5. As expected, the checked versions take slightly longer than the unchecked versions, and the greatest difference is found in the "/bin" detection application. Also, the measurements taken while other applications were running take slightly longer because the applications must compete with each other for access to the memory bus. The "`/bin`" detection application is relatively slow, especially when taking into account that the number of available cycles per packet per μ-engine is 204 cycles when the IXP is receiving 64-byte packets at 500 Mbit/sec. However, the value we measured was the per-packet *latency*, not the amount of processing cycles, and by running this application with multiple packet processing threads (or even multiple μ-engines) it is in fact possible to keep up with high packet rates.

To illustrate the way the framework runs on the IXP, and how the specific features of the IXP can be used for reducing the effects of hardware latency, a snapshot of the thread activity while all four applications were running is shown in Figure 6. The time dimension runs from left to right (the cycle numbers are displayed at the top). Thick black lines indicate processor activity, while the other, thinner lines indicate hardware activity. Thread 0 and thread 1 in the figure are the receiving threads running on the first μ-engine. Thread 0 is waiting on operations performed by the IX bus, the connection between

[3] These 'slots' should not be confused with the slots described in Section 4.4.1.

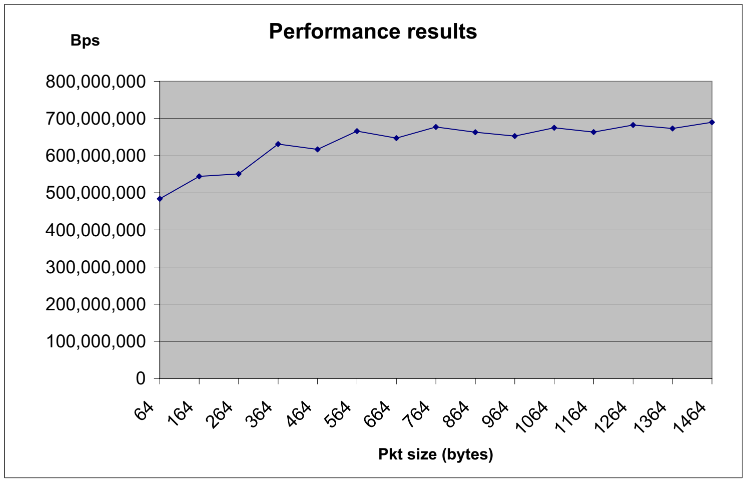

Figure 4. Maximum reception throughput with 2 receive threads on one μ-engine

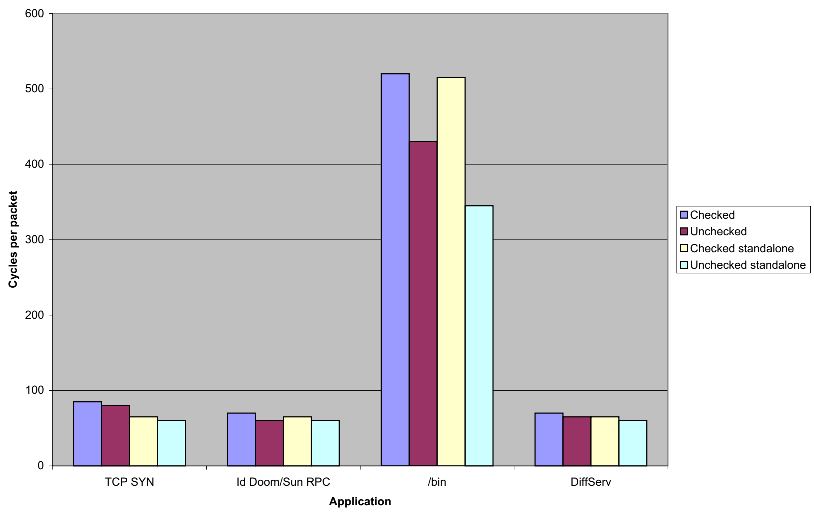

Figure 5. The overhead of runtime checks for various applications

the MAC and the IXP, and while this thread is waiting, thread 1 updates internal logic and fires off an update of the bookkeeping data in SRAM without waiting for it to complete (this can be seen from the fact that the processing line continues after the SRAM request has been fired). Thread 8, which runs on μ-engine 2, is reading a packet header from SDRAM. The fact that it fires off multiple simultaneous SDRAM requests illustrates the general IXP feature that threads can fire off multiple hardware requests without waiting for the results. In this way, the memory accesses are highly parallelised and much of the latency is hidden. In the meantime, thread 9, 20 and 21 are seen to be polling the SRAM bookkeeping information for the availability of new packets.

6 Summary and concluding remarks

We have considered the problem of safely supporting a multi-programmed workload of untrusted applications in a system based on an embedded network processor. We have shown that it is possible to expose the processing resources across all levels in the resulting packet processing hierarchy through a single coherent framework. For the higher levels of the hierarchy, we have directly applied the results of our work in the Open Kernel Environment. For the set of low-level parallel processing units of the network processor, called μ-engines, our work has resulted in a stripped-down and somewhat modified version of the *OKE* known as *Diet-OKE*. We believe that the functionality provided by *Diet-OKE* allows for new applications in which embedded network processors can be useful.

Naturally, such flexibility comes at a price. First,

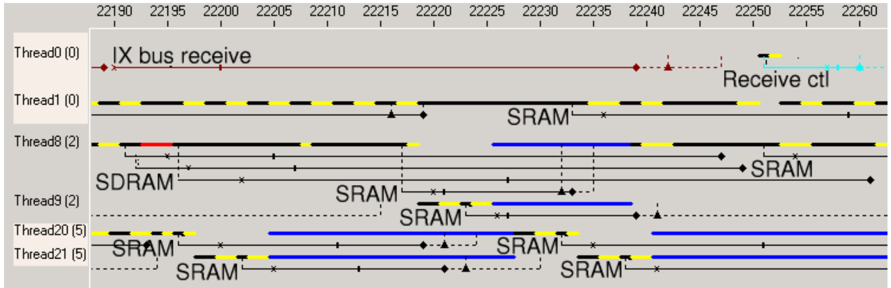

Figure 6. Snapshot of some of the micorengine threads

the runtime checks needed to ensure safety incur overheads. Our experiments suggested that "typical" overheads would be at the order of 20% of the equivalent unsafe code – this is roughly the same as observed in similar general-purpose systems. Second, users that wish to run code on the μ-engines through the use of *Diet-OKE* will have to adhere to the specific programming model stipulated by the application framework of that domain. The model could be restrictive for some applications, although this is not evident in our experiments, as the application framework was designed with the specific applications in mind. As a result, and depending on the framework's environment setup code, the programmer may be unable to exploit fully the available resources. Nevertheless, it is clear that safely exposing these processing resources *requires* mechanisms such as those employed in *Diet-OKE* .

Considering the cost of safety, the design presented is suitable only when the safety overheads incurred are offset by the expected performance benefits. We have discussed the particular example of network monitoring, where moving part of the application function from higher-level processing elements to the embedded network processor can greatly improve performance. We expect that network monitoring, and applications with similar needs, will find the techniques described in this paper useful.

Finally, we anticipate that the basic design principles demonstrated in this paper, if not the architecture itself, can be valuable in guiding the design of other embedded systems with a need for an open software architecture.

Acknowledgements

This work was supported by the EU SCAMPI project (IST-2001-32404), and by USENIX and NLnet through the ReX Programme. The work of the last author is also supported in part by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795.

References

[1] K. G. Anagnostakis, M. Greenwald, S. Ioannidis, and S. Miltchev. Open packet monitoring on FLAME: Safety, performance, and applications. In *Proceedings of the 4th IFIP Int'l Working Conference on Active Networks (IWAN)*, October 2002.

[2] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services. RFC 2475, http://www.rfc-editor.org/, December 1998.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603, pages 185–210. Springer-Verlag, New York, 1999.

[4] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proceedings of IEEE OPENARCH'02*, New York, USA, June 2002.

[5] A. T. Campbell, S. T. Chou, M. E. Kounavis, V. D. Stachtos, and J. Vicente. NetBind: a binding tool for constructing data paths in network processor-based routers. In *Proceedings of IEEE OPENARCH 2002*, June 2002.

[6] S. Consortium. SCAMPI - A SCAlable Monitoring Platform for the Internet, IST-2001-32404, Apr. 2001. `http://www.ist-scampi.org/`.

[7] Intel Corporation. Intel IXP1200 Network Processor. http://developer.intel.com/ixa/, 2000.

[8] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of the USENIX 2002 Annual Technical Conference*, June 2002.

[9] E. J. Johnson and A. R. Kunze. *IXP1200 Programming*. Intel Press, 2002.

[10] S. McCanne and V. Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *USENIX Winter 1994 Technical Conference*, pages 259–270, 1993.

[11] 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)*, pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[12] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[13] H. Wang, D. Zhang, and K. G. Shin. Detecting SYN flooding attacks. In *Proceedings of INFOCOM 2002*, June 2002.