HOKES/POKES: Light-weight resource sharing

Herbert Bos and Bart Samwel
{herbertb,bsamwel}@liacs.nl

LIACS, Leiden University, Niels Bohrweg 1, 2333 CA, The Netherlands

Abstract. In this paper, we explain mechanisms for providing embedded network processors and other low-level programming environments with light-weight support for safe resource sharing. The solution consists of a host part, known as HOKES, and a network processor part, known as POKES. As common operating system concepts are considered to be too heavy-weight for this environment, we developed a system that pushes resource control all the way to the compiler. The HOKES/POKES architecture is described in detail and its implementation evaluated.

1 Introduction

In this paper, we explain how to provide light-weight support for resource sharing in network processors (NPs) and operating system (OS) kernels. In many embedded systems, the programming environment offers little support for resource sharing and safety. Often, the software consists of a single, monolithic application that is special-purpose and well-tested, and not supported by hardware-assisted memory isolation, privileged instructions, or the like. This is true not only for special-purpose signal-processors, but also for reconfigurable computing, such as FPGAs. Increasingly, however, embedded hardware allows for more flexible application domains. The network processor (NP) is an example of this trend [Coo02,Mic99]. It may contain on-chip a general-purpose control processor and a set of independent processing engines all running in parallel (an example is shown in Figure 2). The engines, commonly known as microengines (MEs), run their own micro-applications and share the resources on the board (for instance, buses, memories, data streams, etc.).

In practice, however, the code running on NPs still consists of a single monolithic application (albeit with various parallel components). The reason that NPs are not really shared by multiple programs is not that it would not be beneficial to applications. On the contrary, there are systems that could exploit such functionality to their profit, if only it existed. For example, plugged into a router, an NP board could forward packets on most of its MEs, while using the remaining ones to run user code (e.g. a monitoring application, or a filter). As a more concrete example, the EU SCAMPI project develops a monitoring platform for high-speed links that aims to allow multiple monitoring applications to be active simultaneously on the same network card [SCA01]. The real reason why such architectures are not shared is that there are simply no appropriate mechanisms

for doing so. For example, the time-sharing techniques developed for general-purpose operating systems such as UNIX are not appropriate, as they are too heavy-weight.

In this paper we describe the details of a solution known as OKE that pushes the burden of resource sharing in environments without explicit hardware support for this purpose all the way to the compiler. The focus is on NPs and OS kernels and a C-like language. In our terminology, the OKE part that is running on host-like systems is known as HOKES (host OKE system), while POKES (peripheral OKE system) is the part running on the MEs of the NP. Together, they are able to provide a measure of safety to the system by restricting code in its usage of processing time, memory and APIs. A complete HOKES implementation for the Linux kernel is available under the GNU Public License from `www.liacs.nl/~herbertb/projects/oke`. We have also started the POKES implementation and will evaluate the overhead in the results section. Furthermore, we maintain that the principles described in this paper may apply to other environments also. A high-level overview of an early version of HOKES was given in [BS02]. This paper describes the implementation of the low-level mechanisms, compiler support and NP-specific issues.

In Section 2, we describe the HOKES/POKES architecture. Section 3 explains how isolation and resource safety are subject to higher-level policies. In Section 4, we discuss the various mechanisms that we employ to provide resource safety. In Section 5, we evaluate our work. Related work is discussed in Section 6, and conclusions are drawn in Section 7.

2 Architecture

2.1 Processing hierarchy

As shown in Figure 2, the architecture that we wish to program consists of a five-level (L_0,L_1,...,L_4) processing hierarchy. In our case, it comprises one or more host processors running Linux (with the usual kernel/userspace domains), and one or more Intel IXP network processors, each containing a StrongARM control processor running embedded Linux (also with kernel and userspace code) and a set of microengines that run no OS whatsoever. Observe that, besides the PCI distance and amount of available resources, there is very little to distinguish between the combinations of levels L_1/L_2 and L_3/L_4. Indeed, although they probably will be used for different purposes, in our implementation they are treated the same. The application granularity of L_0 is assumed to be the ME, so application *foo* runs on (at least) one ME, dedicated to *foo*.

In the implementation we have used a P4 1.8 GHz host processor and an Intel IXP1200 NP running at 200 Mhz, with a StrongARM control processor, and 6 MEs with 4 hardware contexts each. The NP is mounted on an ENP2506 board fitted with 2 1Gbps Ethernet ports and 256MB SDRAM and 8MB SRAM.

CL	=	code loader
NP	=	network processor
ME	=	microengine
CREC	=	compilation record
OKE	=	open kernel environment
HOKES	=	host-like OKE system
POKES	=	peripheral OKE system
ESC	=	environment setup code
FEC	=	forward error correction
GC	=	garbage collector
SC	=	stack checking
DM	=	dynamic memory management
PC	=	pointer checks
MT	=	multithreading protection
TP	=	processing time protection

Fig. 1. Glossary

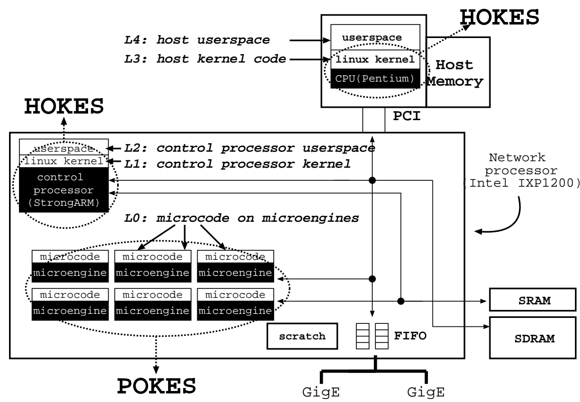

Fig. 2. Processing hierarchy and HOKES/POKES domains

2.2 Software

The problem of efficient resource sharing in this hierarchy is difficult, especially for levels L_0, L_1 and L_3 which concern environments without hardware support for isolation. Worse, we cannot apply traditional OS approaches, as these are too heavy-weight (in particular for the MEs). Instead, we have developed new abstractions for code isolation that push, to a large extent, resource control all the way to the compiler. In our software model, given the appropriate (and explicit) privileges, application code is allowed to run anywhere in the processing hierarchy. However, what such code is allowed to do is limited by static and dynamic checks that may control any resource, including memory and bus access, processing time, stack space, etc. The generic solution for any of the levels in the hierarchy is known as the *Open Kernel Environment* (OKE). The implementation of OKE on a host-like system, i.e. a system running a general-purpose operating system, such as the Pentium and StrongARM in Figure 2, is known as HOKES. The microengines on the NP do not run any OS whatsoever and hence require a somewhat different approach to achieve the same goal. This is known as a peripheral OKE system, or POKES. For the evaluation of HOKES we will use the implementation on the actual host processor (Pentium), as it is more convenient for performing measurements and data collection than the embedded StrongARM. For POKES, however, we have no choice other than using the microengines directly.

In order to make programming safe, efficient, and familiar to programmers, we extended and modified the *Cyclone* programming language, which itself is a dialect of C (retaining the C 'look and feel') [JMG^{+}02]. The result is known as *OKE-Cyclone*. The trust management used in the compilation and loading process is based on *KeyNote* [BFIK99] and described in detail in [BS02].

The idea is that a code loader explictly grants users the right to load code of a specific *type*, where 'type' means: 'subject to a set of restrictions on resource usage'. For this purpose, they request a compiler to compile their code according

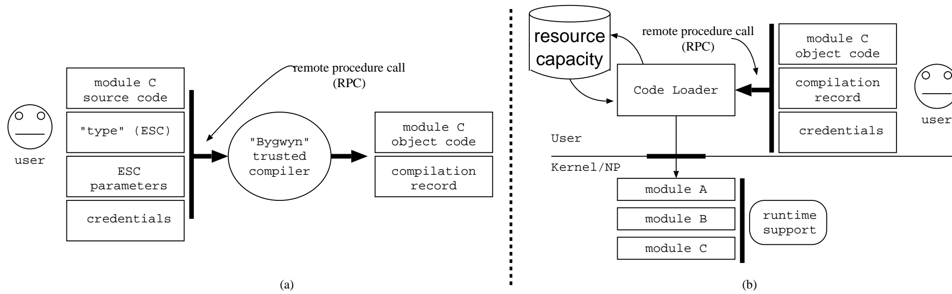

Fig. 3. User (a) compiles and (b) loads code using RPC from a remote site

to this type. A type is defined by so-called 'environment-set-up code', or ESC, that is associated with a user's program (and which instruments the code and checks whether specific conditions on resource usage hold) and is instantiated by the user by parameterisation. For example, the ESC may specify that the maximum amount of heap that can be used is 1 MB, while the parameterisation may indicate that the real maximum to be used for this code is only 0.5 MB. ESC will not normally be written by the users themselves, but defined once and for all by the authority that wants to restrict a specific group of users in a specific way. For example, there could be an ESC for 'students in a course of embedded software'. The compiler compiles the code with the ESC specified and then generates and signs a proof stating that this code was compiled with these restrictions by this compiler. The exact steps are the following(see Figure 3):

- To a (possibly remote) compile-server we submit (1) our source, and (2) the 'type', or ESC, to which the object code should comply.
- This trusted compile-server, known as *bygwyn*, first creates a *combined translation unit* that incorporates the ESC and our source code.
- *Bygwyn* then calls the OKE-Cyclone compiler to generate object code using custom language constructs to enforce safety (and isolation).
- *Bygwyn* returns the object code (an OKE *module*) and a signed *compilation record* (CREC) that vouches for the code's compliance to the ESC.
- Users may then submit the object code, the CREC and ESC to an *OKE code loader* (CL) at a site where the code should be executed.
- The CL uses the CREC to check whether the (1) object code matches the given ESC, (2) the compiler that signed the CREC is trusted, and (3) there is sufficient resource capacity to accommodate the request. It also checks the users' credentials to see if they are *allowed* to load code corresponding to this ESC. If none of the checks fail, the code is loaded.

3 Safety policy (ESC)

In essence, an ESC is policy code that is *prepended* to, and sets up a compile-time environment for, the user-supplied code. For example, it defines run-time

support code and explicitly declares the permissible API, restricting the code to just this API, by taking away the ability to (1) declare/import new APIs, (2) access the "private" parts of the ESC, and (3) perform privileged (unsafe) operations. ESCs are parameterised by the user to include, limit or remove certain capabilities. For example, users may specify that they have no need for dynamic memory, so that the support code that would otherwise be needed can be removed (yielding more efficient code). An ESC also has a component specifying *parameter limitations*. For default parameter values that may be overwritten by users with the appropriate credentials, this includes the range of valid values. It may also include a `FINALLY` keyword to indicate that this parameter may not be overwritten by users. This permits one to specify for instance that some users might be allowed to specify that they need up to 1 MB of dynamic memory, while others might be allowed to specify only 0.5 MB (or may not even be allowed to use heap memory at all). Apart from the ESC, these limitations are expressed in the users' credentials.

When user code is compiled with a given policy (ESC), the trusted compile-server creates a translation unit that consists of (1) parameterisation of the ESC (in the form of macro definitions), (2) ESC, and (3) user code. This is processed by the OKE-Cyclone compiler to generate compliant object code. The OKE assumes a closed world, allowing for whole program analysis to optimise away dynamic checks that are only needed in an open world. This does not mean that modules consisting of multiple components cannot be composed. As long as the APIs are declared in the ESC, users can build larger programs by explicitly clicking together a score of modules. Even so, the OKE is especially useful for applications in restricted areas (e.g. kernels and embedded systems) and, therefore, the modules are often fairly small (i.e. contained in a single unit).

4 Language Features for Safety

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.

4.1 HOKES/POKES Spatial Pointer Safety

Cyclone is strongly typed and provides pointer safety in the following ways. Firstly, it provides *bounded pointers* that are statically tagged with a 'valid

length' or number of elements that must exist in the memory the pointer points to. These pointers are either NULL or point to at least the number of elements indicated by their 'valid length'. Conversions between these pointers are checked statically. Secondly, Cyclone provides *non-nullable* pointers, that can be statically proven to be non-NULL. Unlike 'normal' C pointers, non-nullable pointers do not need dynamic checks when they are dereferenced. Converting pointers to non-nullable pointers before using them in a loop, allows programmers to remove dynamic NULL-checks from inner loops. Thirdly, Cyclone also supports 'normal' C pointers, e.g. pointers with bounds that are only known at run-time and which incur run-time checks on all accesses.

4.2 HOKES/POKES Temporal Pointer Safety

Bounds checking solves the problem of *spatial* pointer safety, but not that of *temporal* pointer safety, such as returning the address of local variable. Cyclone solves this by including *region* information in pointer types [GMJ+02]. The mechanism statically tags every pointer type with an identifier of the region of the memory it points to. A pointer that is tagged to point to a region *foo* is only capable of pointing to memory in region *foo* or in region *bar* that *outlives foo*, i.e., a region whose memory is guaranteed to exist when *foo* goes out of scope. In addition, *foo*'s memory can store only pointers to memory that outlives *foo* (because otherwise a "shorter-lived" region might disappear during the life of *foo*, leaving a dangling pointer in *foo*). When the address of a local variable of function *foo* is taken, the result will be a pointer into the *region* of *foo's* local variables. Any attempt to return it, or store it globally, will lead to region errors at compile time. Region tags can be explicitly specified by the programmer to express certain exceptional situations, but in C-like code the default region tags usually suffice (i.e., the code looks and feels like C, but has invisible safety enforcement). The region system is not used within the dynamic memory heap, which is normally garbage collected.

The Cyclone regions work well for Cyclone-only programs, but there are safety issues when interoperating with explicitly memory-managed languages such as C, as is the case, for instance, in HOKES. When a module holds a pointer to a kernel memory block that is explicitly memory managed, the memory block may be released, leaving the pointer dangling. For HOKES, we have therefore implemented Linux support for *delayed freeing*, ensuring that any memory released by the kernel is not immediately reused, but instead placed on a *kill list*, a list of blocks that still need to be "physically released".

In addition, we have written a HOKES garbage collector (GC) from scratch. We have placed the GC under user control, so that users may explicitly request a GC round (e.g. to clean up their own heap memory), or postpone it indefinitely. However, just prior to becoming active, every HOKES module which may have pointers to kernel memory *always* initiates a GC round (this is known at compile time). GC takes O(n), where n is the number of memory blocks allocated by the module. During garbage collection, all memory on the module's heap that can no longer be reached will be released.

During a module's garbage collection phase the HOKES detects whether or not a module holds pointers to 'freed' memory from the explicitly memory-managed region and acts appropriately by nullifying the pointers or by terminating the module (if the pointers are non-nullable). Once we have verified that no module holds pointers to a 'freed' memory block, the memory block is also physically freed. Currently, the released memory blocks are physically released even earlier, immediately after all modules have *deactivated*. This puts some restrictions on the programming model, because modules cannot safely stay active for very long times without exhausting the kernel's memory resources. We don't think this limitation is inherent in the approach, it is only caused by the simplicity of the current implementation. A more important limitation is that kernel memory cannot currently contain pointers back to memory controlled by HOKES's GC, because the GC cannot detect the kernel's references. All memory shared by HOKES and the kernel must be explicitly managed by the kernel.

We stress that delayed freeing is only needed for those modules that can actually hold pointers to kernel memory. This is a property that is checked at compile time: programs that do not contain any pointers tagged "kernel region" cannot point to explicitly memory-managed kernel memory. In other words, when considering a `free` for a chunk of kernel memory, there is no need to wait for modules to become inactive if they are guaranteed not to have pointers to this memory anyway.

Our current GC implementation uses a mark-and-sweep GC algorithm and is precise (it assumes strong typing and does not need to assume that all memory potentially contains pointers). The GC is supported by automatically generated code based on whole program analysis, which is done in the front end of the OKE-Cyclone compiler. The compiler detects which types of memory can be allocated by a module (by enumerating the types of all `new` and `malloc` expressions in the program) and generates *marking functions* for every type it encounters. A marking function for a type defines how a memory block of the type can be scanned for pointers, it contains a call to a function of the GC for every pointer in the type. The memory allocation calls in the module are then modified to pass to the memory subsystem pointers to the marking functions corresponding to the allocated type. Whenever the memory subsystem wants to find out which other memory blocks are referenced by a given memory block, it calls the marking function associated with the block and the marking function will call the GC with the addresses of all the other blocks referenced in this block.

POKES has no GC and currently does not even support truly dynamic memory. It is assumed that all memory accessible to MEs is statically declared. Region-based protection is still used, but in a more restricted version (e.g. to guard against common programming errors). As we shall see in 4.8, the memory may well be shared between MEs.

4.3 HOKES/POKES Language Restrictions

Although regions and pointer tags provide pointer safety, they provide no access restrictions for APIs and unsafe language features. To be able to restrict explic-

itly access to unsafe language features and APIs, we added the construct `forbid` to the language. Using this construct, it is possible to restrict access to features like `extern "C"` (which would allow users to import C APIs they don't have access to), `extern "C include"` (a Cyclone construct that allows for the inclusion of unsafe C code). In addition to that, `forbid namespace` enables us to forbid access to a complete namespace to all the code following this declaration. This allows an ESC to declare private helper functions and to import APIs in a private namespace, and to remove this namespace from the user code's view afterwards, leaving only the permissible APIs. In HOKES, unauthorised interaction with other modules is prevented by opening a *random* namespace for the user code. Because HOKES uses Cyclone's exception mechanism to interrupt misbehaving code in a safe manner, it is not safe for HOKES modules to catch certain types of exceptions. To prevent this we created the construct `forbid catch`, that allows exception access control at exception type granularity.

4.4 HOKES/POKES Wrapping: APIs and Entry Points

Upon entry from the environment into code that is part of a HOKES or POKES module, some entry/exit code needs to be executed, e.g. to perform HOKES garbage collection or catch any exceptions thrown from the module. We realised that it would be very useful indeed to be able to pass *function pointers* to external code so that such code may call these functions directly. However, to do so safely, we need to be able to wrap each of these functions. For this purpose, we created a new construct, `wrap extern`, that *automatically* wraps every exported (`extern`) function with wrapping code specified by the `wrap extern` declaration. At the same time, we prevent taking the address of functions that were not declared `extern`, so function pointers to non-extern, unwrapped functions cannot exist. In restrospect, our decision to bind the concept of "potential entry point" to the keyword "extern" has turned out to be inconvenient, because it forces the addition of entry/exit code even to functions that are never used outside of the module, but are called through function pointers.

The structure of the OKE also does not allow an ESC to make an external API directly accessible to an application. The reason is related to timeouts, a subject we will discuss in Section 4.7. Moreover, when an API is potentially unsafe, e.g. if it returns explicitly memory-managed memory directly or if it has weak parameter checking, extra work is needed to make Cyclone and the external API interoperate safely. In these situations, the APIs are *wrapped* by the ESC.

4.5 HOKES/POKES Protection of Sensitive Data

When data is shared between programs, we often want only certain fields of a structure to be shared, while others should not be exposed. In network monitoring, for instance, it may not be permissible for applications to find out which websites a user visits. To prevent this, the IP address of a packet is often scrambled (anonymised). However, scrambling takes time, even if the application is

perfectly safe and does not even attempt to access the sensitive field. To address this, we created a new type modifier `locked` in OKE-Cyclone. The value of a `locked` variable cannot be used in calculations and cannot be converted to any other type, and is therefore rendered unusable at zero run-time overhead. An ESC can declare certain fields of structures `locked` to prevent user code from accessing them. The fact that a field is `locked` does *not* imply that the field cannot be written! The Cyclone language already has a construct to express this, `const`. In fact, `locked` non-`const` fields provide the interesting possibility to enforce a certain 'dataflow', that is, if a module is supposed to create a certain data structure to achieve it's task and one of the fields of this data structure is declared `locked`, then it can only fill this field using a source of `locked` data that the ESC provides: it is not possible to cast an existing non-`locked` value to a `locked` value.

4.6 HOKES Stack Overrun Protection

Unfortunately, HOKES stack overruns are hard to prevent without dynamic checking (in POKES this is not a problem as the MEs do not support a stack). However, we do have the advantage of having whole program analysis at our disposal, and use this to reduce the amount of checks. Stack overrun protection is configured by two variables, the *bound* and the *granularity*. The bound defines the amount of stack space that a program is allowed to use, while the granularity defines the maximum distance (measured in stack space) between checks in the program.

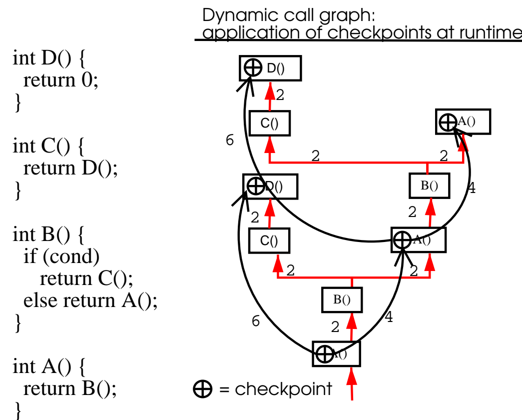

Fig. 4. Dynamic call graph: all functions take 2 units of stack (granularity = 6)

The process is illustrated in Figure 4 (it is assumed that for each function call the required stack space is 2 units). The OKE-Cyclone compiler analyses the entry points and the code's call graph to determine good locations for the

checkpoints that satisfy the granularity constraint. For example, consider the set of (partly recursive) function calls of Figure 4. Suppose checkpoints were placed at the entry points of `A()` and the entry point of `D()` only (and not in the other two functions). Now, whenever `A()` is called, it is possible to check whether enough stack space exists to reach each of the following checkpoints. In the figure, bent arrows denote the distance between checkpoints, while straight arrows represent the call graph. A check is needed whether there is enough stack to store all of `A`'s local variables (in the general case: in the example there are no local variables), make a function call to `B`, store all local variables in `B()` on the two (different) paths to the next function calls (either `A()` or `C()`) and finally to make these calls. In case of a call to `C()`, we repeat this process until we have reached a checkpoint, i.e. we check whether there is enough stack to call `D()`. In other words, at the entrypoint of `A()` we need a total of at least 6 units of stack space. Even though no checkpoints were added in `B()` or `C()` at all, we are ensured that whenever either of these functions is called, there will be enough stack space to proceed to the next checkpoint.

Discussion. In the current implementation it is not possible for the programmer to directly control the locations of the checkpoints, but as this can be added fairly easily, it is expected to be in the next HOKES release. Right now, checkpointing is done by the compiler which tries to balance *bound* and *granularity* (minimising the number of checkpoints, while not placing them too far away from each other). For safety, we assume that all calls to imported functions may result in calls to all entry points into the code, while this is usually not the case. Performance would benefit if the ESC could specify whether an imported function can result in a callback or not.

4.7 HOKES Processing time protection

On many host systems that HOKES targets, processor time is a resource that must be protected. To prevent user code from getting stuck in an endless loop or to use up all processor time for too long, we had a choice between two solutions. The first involves the addition of dynamic checks in backward jumps, which incurs considerable run-time overhead and was therefore not used in HOKES. The second involves a limited amount of OS and hardware dependency, exploiting the timer interrupt in Linux. The implication is that for every new OKE environment, either a new method should be invented or dynamic checks should be used. For example, in the next section we will show that for POKES we adopted a very different solution to limit processing time.

HOKES uses the platform's *timer interrupt* to check for timeouts. The HOKES timers are checked on returning from a timer interrupt, i.e. after all truly time-critical tasks have been handled, but *before* returning control to the interrupted code. When a timeout flag is detected, HOKES jumps to a callback function that was registered by the ESC. When appropriate, the callback throws an exception to terminate the computation. The timeout mechanism takes into account

whether a modules is executing HOKES or OS code when the timeout occurs. When executing system code, written in a different language, we cannot throw a Cyclone exception and rewind the stack at that point in time. Rather, a flag is set to indicate to the ESC's wrapper functions that they should not return control to the user code but throw a timeout exception instead.

4.8 POKES Processing time protection

As each application runs on a separate processor, POKES processor time is not considered to be a shared resource. Hence, there is no use for the timeout mechanisms discussed in the previous section. However, as throughput is essential, packets have to be processed in time and memory is shared between the applications. So we rephrased the processing time constraints in terms of cycle budgets, memory sharing and locking issues. In POKES, MEs are assumed to share a single data stream and the memory in which the stream is stored is a shared resource. If an application is not able to keep up with the arrival rate of the data, the backlog of data-to-process grows and the throughput drops. This is particularly bad in NPs where the goal is precisely to keep up with line speed and which have only a small cycle budget per packet (or set of packets).

Processing time protection in the current POKES is closely tied to the application framework, which reserves a single ME for receiving packets from the network and storing them in memory. All other MEs are available to the applications. The ESC for this framework provides for each ME a 'main' loop which prefilters packets and makes them available to the ME hardware contexts. Given the appropriate credentials, programmers may 'plug in' application code in this loop (with POKES restrictions) and load the complete program on the ME.

The packet fetch ME dedicates a single thread (known as 'mop-up') to dealing with tardy applications. Whenever an ME is falling behind too much (it has not processed enough packets in a given amount of time), the mop-up terminates it. We consider this to be the NP equivalent of timeouts. The ME only posts a kill request, the actual termination is done by the StrongARM. The packet fetch that we implemented places packets in a circular buffer spread over SRAM and SDRAM. The actual packets are stored in SDRAM, while buffer control structures are kept in SRAM. For example, a bit field per microengine per buffered packet is used to indicate whether a ME is 'done' with this packet. Applications may choose to process every packet or only a percentage of the packets. However, for all packets in the buffer they have to set their 'done' flag in time. In the same structure we also keep bit fields to implement readers/writers locks (with readers' preference) and fields indicating whether the packet has been fully received.

Packets are written in 64 byte chunks and applications do not have to wait for the packets to be received in their entirety: as soon as the first chunk of bytes has arrived, they may start processing. The point is that they need to process the packets within the cycle budget. This is enforced by the mop-up. At some distance from the writing position the mop-up thread explicitly *removes* packets from the circular buffer. If the mop-up finds that, for any ME, the 'done' bit is

not set, it means the corresponding application has not completed the processing of this packet. In other words, the application is too slow and will be terminated. By varying the distance between mop-up and receive, we may limit or extend the cycle budget for sets of packes available to applications.

Finally, we extended the default Intel device driver to provide memory mapping of all the IXP's SDRAM all the way to the host applications running in userspace. MEs may pass a reference to a packet to a queue destined for host applications, which prompts these applications to process the packet further on the host (accessing the required data from across the PCI bus).

Discussion: a POKES compiler. Although there is currently no full compiler support for POKES, all mechanisms have been implemented and evaluated in isolation by handcrafting the code exactly like it will be generated by the compiler. Furthermore, we are in the process of implementing a compiler for POKES that will be rather simple in that it generates microengine C, an Intel-proprietary dialect of C with ME programming extensions. The current compiler is capable of producing ANSI C, so the translation to microengine C is fairly straightforward, requiring mainly 2 things: (1) as the storage class for all variables can be explicitly specified in microengine C, the compiler needs to be extended with storage class specifiers, and (2) we need to deal with the presence of many intrinsic functions in microengine C (but as these intrinsics have the same appearance as function calls, they are easy to support).

5 Evaluation

5.1 HOKES: Full kernel-based monitoring and transcoding

HOKES has been evaluated in various kernel experiments in network transcoding. The most interesting example applications are (1) transcoding with increasing packet size, adding forward error correction (FEC) to packet payloads, and (2) transcoding with decreasing packet size (resampling audio packets to contain only 50% of the original data).They were implemented directly in a single module that attaches itself to a Linux netfilter hook. The ESC is responsible for removing the appropriate IP packets from the netfilter framework, casting the packet structure to the corresponding types in OKE-Cyclone and passing it to the module (ensuring that the appropriate fields are made `const` and/or `locked`). It is also responsible for transmitting packets again.

For both cases, Table 2 shows the results for both the HOKES kernel code and the exact same implementation in C (also running in the kernel). In brackets the relative overhead is mentioned. Because the overhead varies with the packet size (more data needs to be processed), we listed both the overhead for minimum sized and maximum sized packets. A lot of the overhead, especially in the audio resampling transcoder, consists of initial costs. After that, in the loop of the transcoder, there is hardly any overhead. For this reason the difference between optimal C code and the HOKES solution is smaller, percentage-wise, for larger packets.

5.2 HOKES Micro-measurements

Given the appropriate privileges, an ESC may be parameterised to 'turn off' specific OKE mechanisms (e.g. a module may be compiled without the GC, and/or without stack checks, etc). If such parameters are not explicitly declared as `FINAL`, they may be overwritten by users with the appropriate credentials, allowing them to determine what runtime support is included in the compilation. This conveniently allows us to test the overhead of many of our mechanisms in isolation. In this section we measure the runtime overhead of various aspects of: stack checking (**SC**), garbage collection (**GC**), dynamic memory management (**DM**), pointer checks (**PC**), overhead incurred by spinlocks to prevent multiple threads from accessing the same module at the same time (**MT**), and processing time protection (**TP**). For each of these items we measure various aspects, as summarised in Table 1.

All speed measurements are in cycles and were conducted on an Intel P4 running a Linux 2.4.20 kernel and are fast-path values (i.e. without initial cache misses and inaccurate branch predictions and without any misbehaviour in the code that would cause it to be terminated). After startup, in the applications of Section 5.1, the fast-paths are always taken as the system was used solely used for packet transcoding. All code was compiled to object code using `gcc-2.9.5`. We suspect that using `gcc-3.2` performance will be better as the abstraction penalty (putting all components in function calls, etc.) in `gcc-3.2` is less severe (e.g. because its support for inlining has improved). We should stress than in POKES the overheads (and indeed the mechanisms) for GC, TP and SC do not exist.

Test	Aspect measured	Result
$\mathbf{SC}_1$	init time at entry point	≈ 1 cycle
$\mathbf{SC}_2$	additional time to process checkpoint code	5-10 cycles
$\mathbf{SC}_3$	code size increase	16 instr. at entrypoint 15 instr. at other checkpoints
$\mathbf{PC}_1$	overhead per normal (C-style) pointer dereference	≈ 1 cycle
$\mathbf{PC}_2$	code size increase	3 instr. (commonly only 2 executed)
$\mathbf{MT}_1$	overhead at entry point	35 cycles
$\mathbf{MT}_2$	code size increase at entry point	15 instr.
$\mathbf{TP}_1$	overhead at entry point	68 cycles
$\mathbf{TP}_2$	extra overhead for calling kernel from HOKES module	10 cycles
$\mathbf{TP}_3$	code size increase per entry point	24 instr.
$\mathbf{GC}_4$	overhead for turning GC on, but not using it	10 cycles (for check)
$\mathbf{GC}_5$	overhead for user requesting GC round with nothing to do	98 cycles
$\mathbf{GC}_6$	overhead for GC round for checking 1 non-collectible pointer	73 cycles
$\mathbf{GC}_7$	overhead for GC round for checking 1 collectible pointer	440 cycles
$\mathbf{GC}_8$	time to sweep a block	10-41 cycles (plus cost of `kfree`)

Table 1. Overhead of various aspects of OKE mechanism

5.3 POKES: Monitoring and transcoding in a network processor

POKES was evaluated using the application framework of Section 4.8. As memory is allocated statically and processors are dedicated, the only true overhead incurred by POKES is caused by memory bounds checks. We tested four applications, all running on their own MEs:.

1. *TCP SYN detection*: denial of service attacks are often caused by sending many TCP SYN packets, so we count the number SYNs per time unit.
2. *UDP port detection*: this application detects any packets destined for UDP ports belonging either to Id Software (Doom), or to SunRPC.
3. *Scanning for a string*: for intrusion detection it is often needed to scan a packet for potentially harmful content (e.g. `/bin/perl`. In our example, we scan the first 16 bytes of the payload for the string."`/bin`".
4. *Set DiffServ field*: in this application we *write* a byte in the DiffServ field.

Although simple, all tasks derive from real-world applications. The throughput that can be achieved without any application running was close to 700 Mbps, on a 1 Gbps card, and we were able to sustain this load also with all applications running.

The average overhead caused by the additional memory bounds checking for the various applications is summarised in Table 3. The large overhead for application 3 is caused by its frequent memory accesses during the string search. Each of these is checked, leading to the highest absolute and relative overhead: 20%. At maximum rate (700 Mbps), the cycle budget per packet is significantly less than 520 cycles: roughly 200. The only reason why we were able to keep up was that we hide latency by using multiple hardware threads (a new thread starts scanning a new packet, whenever the current thread is waiting for a memory access to complete).

Program	**min.pkt** (μs)	**max.pkt** (μs)
FEC_C	1.3	18
FEC_{HOKES}	1.5 (15%)	19 (6%)
resample_C	1.3	8
resample_{HOKES}	1.8 (38%)	8 (≈0%)

Table 2. Total HOKES overhead

	Experiment	**Unchecked code** (cycles)	**POKES** (cycles)
1	TCP_SYN	80	85 (6%)
2	UDP_port	60	70 (17%)
3	String_scan	430	520 (20%)
4	DiffServ	65	70 (8%)

Table 3. POKES bounds check overhead

6 Related Work

There are several OSs that specifically target embedded systems and we do not intend to cover all of them in detail. Most relevant to our work are embedded Linux and VxWorks AE [Win01]. Embedded Linux is just like ordinary Linux but tailored to embedded processors and it is what we currently run on the NP's

general purpose control processor. VxWorks (which can also be used on this processor) is a real-time OS that is widely adopted in the embedded industry to control hardware. Moreover, it provides more than just CPU protection. For example, it is able to use the MMUs of modern microprocessors to provide partitioning and protection of memory in a flexible manner ("dial-in protection"). It differs from our work in that we attempt to provide isolation to systems without having to rely on such hardware assists.

In research labs, we have also seen a number of general-purpose OSs that can be safely extended (e.g.Nemesis [LMB+96], ExoKernels, [EKO94], and SPIN [BCE+95]). Of these SPIN, which makes use of language-specific safety properties to ensure that applications do not interfere with each other, is most similar to our work. Unfortunately, all of these systems are research-oriented and not widely used, certainly not in embedded systems. Our aim is to provide isolation mechanisms that can be applied in a popular OS (notably Linux), and even in systems without OS support whatsoever (notably the MEs of an NP).

Previous attempts at providing software-based isolation include interpreted solutions(e.g. BPF [MJ93]), and native code solutions like Software Fault Isolation (SFI) [WLAG93] and Proof Carrying Code (PCC) [NL96]. We do not consider interpreted solutions suitable for high-throughput systems. Instead, it is our explicit goal to open up lower levels of the processing hierarchy to fully optimised code. SFI uses run-time checks to enforce safety. Not only are such checks costly , they also only take into account memory isolation (e.g. bounds checking), and not control isolation (e.g. branch checking). PCC provides code safety by supplying a *proof of correctness*, so that the loading site may check the correctness of code prior to loading it. The problem here is that generating proofs is a complex task which, to date, cannot be fully automated. Other approaches that allow programmers to load code in the Linux kernel are SILK [SPB+02] and FLAME [AIM+02]. SILK differs from our work in that it does not provide safety, and FLAME in that it is limited to monitoring functionality. RBClick described in [PL03] is similar to OKE, but differs in that all checks are completely static.

7 Conclusions

In this paper we have described the low-level mechanisms and compiler support for the Open Kernel Environment (OKE) which provides light-weight support for resource sharing and isolation in a processing hierarchy consisting of host processors and network processors. We have discussed both host side (HOKES) and network processor side (POKES). Policies are used to restrict a user's code in terms of access to resources. Whether or not users are granted access to resources is determined by their credentials. At the same time, the OKE targets performance by focusing on fully optimised code. Experimental results in the fields of monitoring and transcoding show that in Linux the overhead ranges from roughly 5 to 40 percent, and experimental results on Intel IXP1200 network processors show overheads of up to 20 percent. The OKE is currently a component in the operating system support for the EU SCAMPI project.

Acknowledgements

This work was supported by the EU SCAMPI project (IST-2001-32404). We would like to thank Lennert Buytenhek and Mihai Cristea for their help.

References

[AIM+02] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, Michael B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proc. of NOMS'2002*, April 2002.

[BCE+95] B. Bershad, C. Chambers, S. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. Gun Sirer. Protection is a software issue. In *HotOS-V*, May 1995.

[BFIK99] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromytis. The KeyNote trust-management system version 2. *NWG, RFC 2704*, September 1999.

[BS02] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proceedings of OPENARCH'02*, New York, USA, June 2002.

[Coo02] Intel Coorp. Internet exchange architecture: Programmable network processors for today's modular networks. White Paper, 2002.

[EKO94] D. Engler, F. Kaashoek, and J. O'Toole Jr. The exokernel approach to extensibility. In *Proc. of USENIX OSDI*, Monterey, Cal., November 1994.

[GMJ+02] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proc. of PLDI'02*, Berlin, Germany, June 2002.

[JMG+02] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proc. of USENIX'2002*, June 2002.

[LMB+96] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *JSAC*, 14(7), September 1996.

[Mic99] IBM Microelectronics. The network processor enabling technology for high-performance networking. White Paper, 1999.

[MJ93] S. McCanne and V. Jacobson. The BSD Packet Filter: A new architecture for user-level packet capture. In *Proc. of USENIX*, San Diego, Jan. 1993.

[NL96] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of OSDI'96*, Seattle, Washington, October 1996.

[PL03] P. Patel and J. Lepreau. Hybrid resource control of active extensions. In *Proc. of OPENARCH'03*, San Francisco, CA, April 2003.

[SCA01] SCAMPI Consortium. SCAMPI - A SCAlable Monitoring Platform for the Internet. Technical report, EU IST Research Project, Proposal Number: IST-2001-32404, Action Line IST-2001-IV.2.2, April 2001.

[SPB+02] N. Shalaby, L. Peterson, A. Bavier, Y. Gottlieb abd S. Karlin, A. Nakao, X. Qie, T. Spalink, and M. Wawrzoniak. Extensible routers for active networks. In *DARPA AN Conference and Exposition*, June 2002.

[Win01] Wind River Systems MCL-DS-VAE-0111. VxWorks AE Datasheet. `http://www.windriver.com/products/vxworks5/`, 2001.

[WLAG93] R. Wahbe, S. Lucco, T.E. Anderson, and S.L. Graham. Efficient software-based fault-isolation. In *SOSP'03*, pages 203–216, December 1993.