Lightweight Thread Tunnelling in Network Applications

Austin Donnelly

University of Cambridge, Computer Laboratory, 15 J.J. Thomson Avenue, Cambridge, CB3 0FD, U.K. Austin.Donnelly@cl.cam.ac.uk

Abstract. Active Network nodes are increasingly being used for non-trivial processing of data streams. These complex network applications typically benefit from protection between their components for fault-tolerance or security. However, fine-grained memory protection introduces bottlenecks in communication among components. This paper describes memory protection in Expert, an OS for programmable network elements which re-examines thread tunnelling as a way of allowing these complex applications to be split over multiple protection domains. We argue that previous problems with tunnelling are symptoms of overly general designs, and we demonstrate a minimal domain-crossing primitive which nevertheless achieves the majority of benefits possible from tunnelling.

1. Introduction

Modern network elements have many software components – for instance to support user-programmability. Software systems in such contexts must address a fundamental concern, that of multiplexing (sharing) the network element amongst many users. Generally, this involves some form of sandboxing to allow code to be executed on behalf of untrusted users [19, 16]. Consequentially, the multiplexing scheme must trade off between performance and security as well as other factors.

Sandboxing can be performed either at the language level (by using safe languages such at Java or ML), or at the machine level (by appropriate memory protection and CPU features). There has been much prior work on language-level sandboxing [1, 15, 5], and most current EEs (Execution Environments) rely on these techniques [21, 13]. However, language-level sandboxing lacks flexibility: invocations between components written in different languages must negotiate some common data marshalling format. Language-level sandboxing also reduces the utility of large bodies of pre-existing code by making it harder to re-use them.

Using hardware facilities to control access to memory and schedule the node's resources is desirable because it allows any language to be used, permitting legacy code re-use. Marshalling can be efficient since native machine formats for data can be used. The main drawback of using hardware to protect the EEs is that communication between protection domains is expensive due to context switches and cache invalidations. Furthermore, these penalties are increasing: as CPU speeds rise, more and more of their performance comes from effective caching of memory contents, branch predictions, and speculative execution. Frequent switching makes these caches ineffective.

Overall, these costs dissuade application designers from placing their modules in separate protection domains, especially if there is a continual stream of data passing through the application.

Thompson wrote: "The UNIX kernel is an I/O multiplexer more than a complete operating system. This is as it should be." [20]. This vision of a simple I/O multiplexer is one to which we find ourselves drawn once again, this time in the context of Active Network nodes. We introduce Expert, an OS designed specifically for network elements, filling this I/O multiplexer niche [4]. In this paper, we describe how Expert's memory protection architecture and its lightweight thread tunnelling directly support modular hardware-protected applications without suffering an undue performance penalty.

A good multiplexer will schedule the resource it manages. To this end, Expert uses the concept of a *path* (first introduced in the Scout OS [14]) to represent a flow of packets and their associated processing resources. The alternative of using multiple processes chained into a pipeline causes several problems: (1) extra context switching adds overhead; (2) as each process has its own scheduling parameters, it only takes one under-provisioned process to rate-limit the entire pipeline; (3) per-flow resource reclamation is complex, needing all processes to participate, and atomic revocation may be impossible; and (4) if multiple flows with different service characteristics are to be processed by the pipeline then each process must ensure it sub-schedules internally.

Unlike Scout, Expert paths can seamlessly cross protection domains. As an example, Fig. 1 shows the execution trace of a path which has tunnelled from module A to execute privileged code in module B, which in turned tunnelled into module C before returning back to module B and thence to A. Modules A, B, and C are all in separate protection domains, allowing B and C to implement trusted functionality securely.

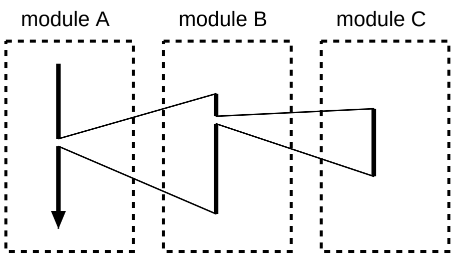

Fig. 1. Tunnelling between protection domains in Expert.

Section 2 discusses existing thread tunnelling schemes, and their shortcomings. Section 3 describes Expert's lightweight thread tunnelling primitive, and how it allows protected modules to be entered. A transcoder used as an example application is described in Sect. 4, and results quantifying the performance of the tunnelling system and the transcoder built over it are presented in Sect. 5. Section 6 concludes this paper and suggests areas for further work.

2. Background

Thread tunnelling was originally proposed as a solution to the performance problems observed in micro-kernel systems, where much inter-component communication takes place. In this guise, tunnelling is usually integrated into the IPC mechanism [2, 11], rather than using a message-passing approach.

Thread tunnelling designs all need to perform a number of core functions. A thread tunnelling primitive takes a thread and changes its runtime environment without passing through the scheduler: how much is changed depends on the individual design. At the very least, the thread's memory access rights are changed; multiple address-space operating systems may also need to switch address space. The vast majority of systems integrate a procedure call (i.e. a program counter change) with the tunnelling primitive. Most systems switch to an alternate stack, but there is some variety in how and when this stack is allocated. Asynchronous events may be masked while a tunnelled call is in progress. Table 1 compares five tunnelling systems against a selection of these features.

Table 1. Thread environment changes in different tunnelling systems.

OS	rights change?	PC forced?	addr. space switch?	stack switch?
The CAP	✔	✔	✘	✔
Spring	✔	✔	✔	✔
Mach	✔	✔	✔	✔
Escort	✔	✘	✘	✔
Expert	✔	✔	✘	✘

The CAP. Cambridge CAP programs are structured using a number of *protected procedures* [22]. A protected procedure can be invoked by any process having an *Enter* capability for it. This contains capabilities for the protected procedure's code, data, stack. There is no separate bind phase; mere possession of an Enter capability is sufficient to call a protected procedure.

Spring Shuttles. Spring introduces *doors* and *shuttles* as the building blocks of its IPC system [7]. Spring *doors* are capabilities, possession of which allows a call to be made to another protection domain. Each door is tailored to a specific client at bind time to allow servers to track their callers. When a call is made through a door, a free server thread is selected by the kernel, and resumed at the door's entry point; this creates a chain of threads, one per server called. Resource accounting and scheduling information is kept in a *shuttle* object shared by all threads in such a call chain.

Mach Migrating Threads. Ford and Lepreau [6] modified the Mach 3.0 IPC system to add tunnelling behaviour. They call their new scheme *migrating threads*. While they quote impressive speedups (a factor of 3.4 improvement when using migrating threads in place of static threads), they encounter practical problems: the need to support thread debugging/tracing, and propagating aborts due to server failures.

Paths Crossing Protection Domains. The designers of Scout [14] argued convincingly that paths are a good way of encapsulating the scheduling and resources used by flows of packets traversing a system. In Scout, data travels between modules along pathways determined at connection setup time. Escort [18] refines Scout by allowing protection domain boundaries to be specified, thus allowing modules with similar trust requirements to share memory protection rights while being protected from other modules in the system. However, protection crossings in Escort are expensive and Escort does not allow batching of packets before pushing them across a protection boundary, so it is impossible to amortise the cost of a protection switch over multiple work units.

3. Thread Tunnelling in Expert

Expert starts from the premise that the thread tunnelling primitive should be as simple as possible while still being flexible enough to support more complex schemes. The tunnelling primitive only changes memory access rights and forces the program counter to the module's entry point – state switching and other environmental modifications are delegated to the called code. We now describe Expert's thread tunnelling architecture in detail.

3.1. Pods

Expert binds protection domains (pdoms) to protected code modules (pods). Each pod's code is only executable by the pod's associated (or *boost*) pdom; paths executing within the pod run in this protection domain. Paths can trap into the pod by invoking a special kernel call which notes the boost pdom and forces the program counter to the pod's advertised entry point.

The kernel keeps a stack recording the boost pdom of each pod called, allowing nested invocation of one pod from another. The access rights in force at any time are the union of all pdoms on the boost stack plus the current base pdom.

This leads to a "concentric rings" model of protection. It makes calling into a pod a lightweight operation because the access rights are guaranteed to be a superset of those previously in force, so no caches need to be flushed on pod entry. Returning from a nested call is more expensive: there is by definition a reduction in privileges, so over-privileged data must be flushed from the TLB (Translation Lookaside Buffer) and other caches. However, this penalty is unavoidable; it is mitigated in systems which permit multiple nested calls to return directly to a prior caller (e.g. EROS [17]) but we believe that common programming idioms mean that such tail-calls are rare. For example, error recovery code needs to check the return status from invoking a nested pod; often a return code needs mapping.

All pods are passive: they have no thread of execution associated with them. This is in contrast with most other thread tunnelling schemes where the servers tunnelled into can also have private threads. Passive pods make recovery from memory faults easier since they are localised to the tunnelled thread.

3.2. Binding

Before allowing client threads to make pod calls, Expert requires them to bind to a pod offer. This enables bind-time access control checks and per-client initialisation. A pointer to this per-client state is stored with the binding record and supplied to the pod on each call. As usual, explicit binds allow a pod to be instantiated multiple times, allowing clients to select between instances by binding to a specific pod offer.

Expert uses a two-phase binding scheme illustrated in Fig. 2. First, clients contact a privileged Pod Binder process (stage 1.1) and ask it to setup an initial binding to a named pod offer (stage 1.2). The initial call on this temporary binding is special: it is interpreted by the pod as a bind request (stage 2.1), and may fail if the pod chooses not to accept this client. Otherwise, the temporary binding is upgraded to a full binding (stage 2.3). This two-phase scheme allows the pod to initialise any per-client state with full access to the calling client's particulars.

3.3. Invocation

Expert introduces a `call_pod()` system call, which takes as arguments a binding ID and an opaque pointer to the arguments.

Pre-switch Checks. `call_pod()` checks that the binding ID is within the client's binding table, and that there is room on the boost pdom stack for another pdom; these are the only tests needed, and both are simple comparisons of a value against a limit. The binding ID is used to index into the client's binding table to recover the pod to be called, and the state pointer to be passed in. Invalid (i.e. unallocated) binding IDs are dealt with by having the Pod Binder pre-allocate all possible binding IDs to point to a stub which returns an error code, thus creating a fast path by eliminating such error checking from the in-kernel code.

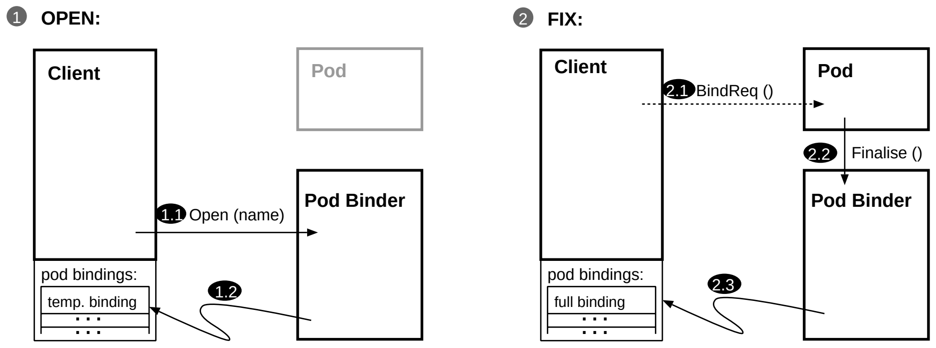

Fig. 2. Two-phase binding in Expert.

Argument Marshalling. There is no marshalling needed when a client tunnels into a pod – the native machine conventions are used. Large arguments are passed by reference in pre-negotiated external buffers. Of course, code in the pod must check that pointers received from the client identify state which is valid for that caller.

Pdom Stack Manipulation. The final step before calling the pod is to push the boost pdom onto the pdom stack, thus making available the rights granted by that pdom the next time a page fault occurs. When the pod call returns the boost pdom is popped off the pdom stack, page table entries are modified, and the TLB is flushed. This ensures that pages which were previously accessible while running in the pod are no longer available. To achieve this, the kernel arranges for control to return to itself after the pod call, rather than directly to the user application.

3.4. Concurrency Control

Taking out locks in pods can cause an effect similar to priority inversion: paths tunnelling into a pod bring their resource guarantees with them, so paths with a large CPU guarantee can be stalled by a path with a lower CPU guarantee. Several solutions exist:

- Paths running in critical regions can inherit the highest CPU guarantee of all blocked paths. This is analogous to priority inheritance in a priority-based system [10].
- Servers could provide "extra" cycles to ensure a minimum rate of progress through critical sections, or servers could simply reject calls from threads with insufficient CPU guarantees [12].
- Blocked paths could donate their cycles to the path currently running in the critical region, thus "pushing" it through. The pushed path later repays the loaned cycles once it has exited the critical section [8].

– Non-blocking data structures such as those proposed by [9] can be used.
– Critical regions can be kept to a minimum number and length. This pragmatic solution is no defence against malicious paths which might proceed arbitrarily slowly, but it works well in most situations. This is the approach taken by Expert.

Note that this means the programmer is responsible for managing concurrency themselves; while this promotes flexibility, it adds to the programmer burden unless standard libraries or code-generation tools are used to automate lock acquire and release.

As with any case of abnormal termination, if any locks are held then the data structures protected by the locks may be in an inconsistent state. Standard solutions to this include rolling back changes, working on shadow copies before committing, forced failure of the whole component, or using lock-free data structures.

3.5. Stack Switching

Unlike Expert, most thread tunnelling systems use a new stack for each protection domain. This prevents threads which have not tunnelled from manipulating the tunnelled thread's stack and/or snooping sensitive intermediate data.

However there are tantalising advantages to not switching stacks on protection switch. The performance is better, since arguments can be passed on the stack directly without copying. Stack pages are likely to be in the cache and to have a valid TLB entry. Also, memory usage is reduced by requiring only T stacks rather than $T \times P$ for T threads traversing P pods.

Expert's `call_pod()` system call does not switch stacks, but instead uses the caller's stack while running inside a pod. The stack access rights are unmodified. This is safe because Expert does not allow other threads from the same base pdom to run while one is engaged in a tunnelled call. A pod may manually re-enable multi-threading if it deems it safe to do so (e.g. after having switched to another stack). The scheme has the twin merits of being simple and fast.

In any case, a paranoid pod can manually perform a stack switch as its first action, since the binding state passed in by the kernel can contain a pointer to a pre-prepared stack. If the kernel were to always switch stacks, a pod would no longer have the choice of whether to run on the same stack as its caller or not.

4. An Audio Transcoder

In this section we describe an example application which benefits from being decomposed into separate modules. We assume an Internet radio station which produces its output as a 44.1kHz stereo 192kbit/s stream of MPEG-1 Layer III audio (MP3). The illustrated application transcodes this source stream into three tiers: "gold" (the premium stream), "silver" (44.1kHz stereo, 128kbit/s, at a reduced price) and "bronze" (11kHz stereo, 32kbit/s, available for free). The transcoders are positioned on Active Network nodes close to the clients they serve, minimising the traffic crossing the core.

We describe the transcoder's design, and show how Expert allows precise control over the scheduling and protection of the various components. Control over resource scheduling allows the transcoder to degrade the level of service experienced by non-paying customers to ensure that paying customers are served promptly. The fine-grained protection offered by Expert should also increase the robustness of the system, although this effect is evidently hard to quantify.

4.1. Requirements

Isolation. Music fidelity should reflect payment. The listeners who have paid nothing must make do with whatever spare capacity is available in the system. Thus the gold, silver and bronze tiers are not only media quality metrics, but should also reflect the OS resources needed while processing streams of these tiers to ensure that streams from higher tiers are processed in a timely manner without loss.

Per-client customisation. Per-client customisation is needed, for example to target adverts, offer different disc-jockey "personæ", provide personalised news and weather, encryption or watermarking; we use AES (Advanced Encryption Standard) in our example.

Protection. Individual application components should be able to access only the areas of memory they need to perform their functions. For example, the encryption keys should only be readable by the encryption modules, so that key material cannot be leaked from the system.

4.2. Transcoder Architecture

Figure 3 shows how the transcoder application is segmented into components. Arrows depict packet movement and rate; thicker arrows correspond to higher data rates. The modules implementing the basic functionality are shown in rounded rectangles: DEC is an MP3 decoder instance, each ENC is an MP3 encoder instance, and each AES is an instance of an encryption module together with its key material. The rectangles represent the scheduled paths in this system: the common-rx path handles network receive, decoding, and encoding; the gold paths perform encryption and transmission, one per gold stream; the silver paths do the same for each silver stream; and the bronze path encodes and transmits one or more bronze streams. This diagram shows three streams at each tier. Note that unlike the others, there is a single bronze path to handle all free bronze clients; using a path per stream allows independent scheduler control over each of the paid-for streams to meet the isolation requirement, and provides memory protection.

The three caches shown are implemented as pods to allow sharing of their data, and the AES encryption modules are pods to minimise key material visibility within the application.

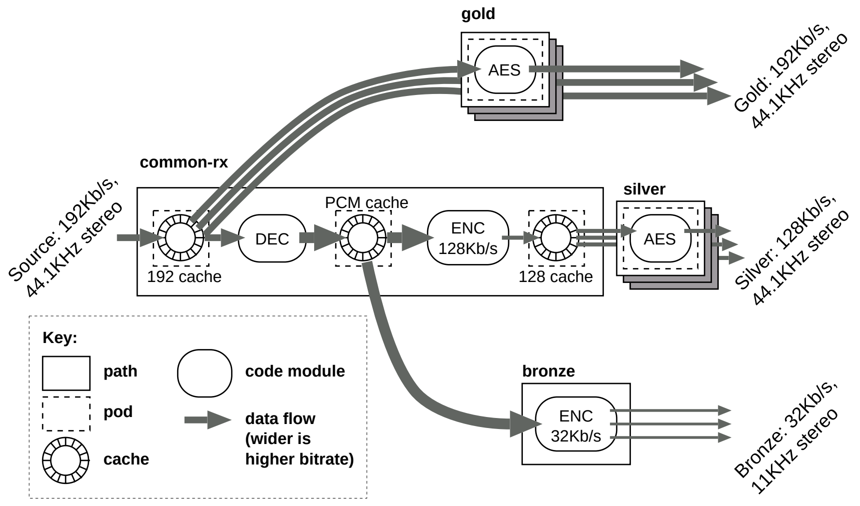
gold
AES
Gold: 192Kb/s, 44.1KHz stereo
common-rx
PCM cache
DEC
ENC 128Kb/s
192 cache
128 cache
silver
AES
Silver: 128Kb/s, 44.1KHz stereo
Source: 192Kb/s, 44.1KHz stereo
Key:
path
pod
cache
code module
data flow (wider is higher bitrate)
bronze
ENC 32Kb/s
Bronze: 32Kb/s, 11KHz stereo

Fig. 3. Data flow through the transcoder. See text for description.

We implemented this architecture in two ways: a *path-based* variant, and an *all-in-one* variant using a single task containing multiple threads. The path-based version schedules the data flow through the system, and provides memory protection between components as described above. The all-in-one does neither: it is a baseline configuration to assess the overhead required to provide protection and scheduling.

5. Results

The test platform in all these experiments is an Intel Pentium Pro system running at 200MHz, with 32MB RAM, 256kB L2 and a split L1 cache: 8kB I / 8kB D. The test machine runs the Expert OS, and the transcoder application in either the path-based or all-in-one variant. It is easy to overload this modest machine: looking at systems when they are overloaded is instructive because this is where differences in architecture matter – if a system cannot shed load in a controlled fashion then it cannot offer different service levels, and is vulnerable to denial of service attacks.

5.1. Micro-benchmarks

Despite the obvious pitfalls of micro-benchmarks [3], they can be used to give a rough idea of the cost of various primitives.

Table 2 compares the cost of various protection switching schemes under Linux 2.2.16 and Expert. It shows how many cycles it takes to execute various types of call. Results for both hot and cold caches are given: "hot" is the average of 100,000 back-to-back calls; "cold" is the exponentially weighted moving average of 40 calls, with activity between each timed call to ensure the caches are filled with unrelated code and data. The cold cache number is more meaningful since (in an optimised system) calls which span protection domains are likely to be made infrequently and thus without already being in the cache.

Table 2. Cycles taken for different calls with hot and cold caches.

OS	proc call	system call	pod call	pipe bounce
Linux (hot)	7	340	n/a	3500
Linux (cold)	44	760	n/a	9100
Expert (hot)	7	280	2900	n/a
Expert (cold)	44	460	5000	n/a

The tests are as follows: "proc call" is a C-level procedure call to a function which takes no arguments and returns no value. The "system call" is `getpid()` on Linux, and a comparable minimal system call on Expert. The "pod call" is a switch from the untrusted client protection domain to a trusted pod environment and back again (the kernel implementation of `call_pod()` comes to just 44 instructions on Intel x86). The "pipe bounce" test sends 4 bytes to another process and waits for a response; this emulates the kind of lightweight IPC that pod calls can replace.

The table shows that an Expert pod call is between 17% and 45% faster than IPC on Linux. It also has better cache behaviour, as can be seen from the cold cache numbers. If the pod being entered is configured to switch to a private stack, the cost rises to 4100 / 6300 cycles for hot and cold caches respectively. Instrumenting the protection fault handler shows that this extra cost arises because twice as many faults are taken when the stack needs to be switched.

5.2. Transcoder Application

A Pentium II 300MHz running Linux is used as the "radio station" source, sending a stream of 192 kbit/s MP3 frames. Each frame (typically around 627 bytes) is encapsulated in a UDP packet and sent to the transcoder. The stream lasts around 146 seconds, and is paced to be delivered in real-time. Transcoded output is sent to a third machine which runs `tcpdump` to calculate the rates achieved by each stream.

Cost of Protection. In this experiment, the transcoder runs on an otherwise unloaded system. The amount of CPU time it consumes is recorded by the scheduler as a fraction of the total cycles available.

The transcoder application is initially configured for one gold, one silver and one bronze stream. We measure the CPU time required by both the path-based and the all-in-one variants as additional silver streams are added until the machine is saturated.

Table 3 shows the measured CPU time required for loss-free operation of the transcoder. The "path-based" row shows the cost for the path-based variant; the other row shows the cost for the all-in-one variant. The path-based version is, as expected, more expensive. Adding protection and proper scheduling costs between 2% and 6% only.

Table 3. Percentage CPU time required to service one gold, one bronze stream against a varying number of silver streams.

	# silver streams					
Variant	1	3	5	7	9	11
path-based	89	92	95	98	–	–
all-in-one	87	88	90	92	93	96

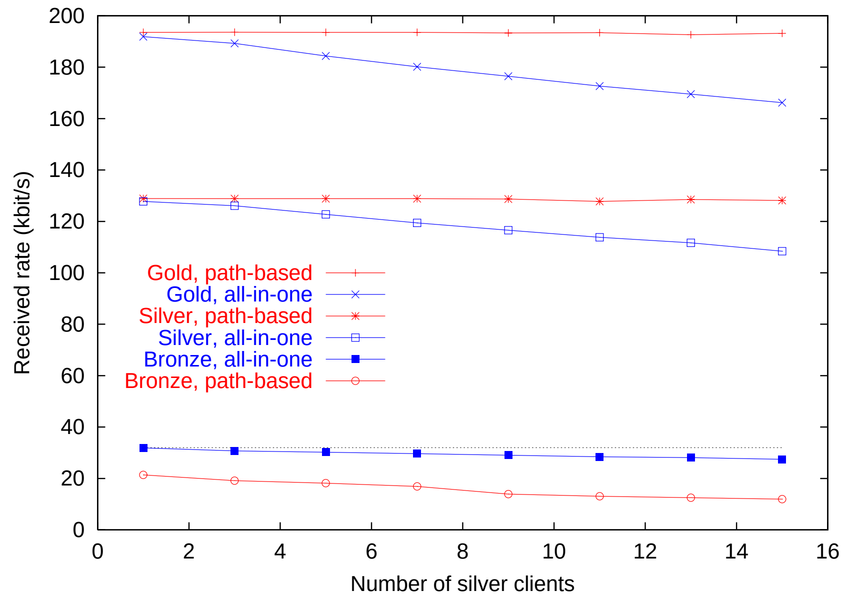

Fig. 4. Achieved rates for gold, silver, and bronze streams with and without isolation.

Benefits of Isolation. In this experiment, the transcoder services five gold streams, one bronze stream, and an increasing number of silver streams. The path-based version is configured to give the common-rx path a 45% share of CPU, the gold and silver paths each get 2%, and the bronze path 15% plus access to any "slack" time in the system. These guarantees are sufficient to meet the CPU needs for the common-rx, gold and silver paths, but the bronze path ideally needs approximately 46%. This means that the bronze path will mostly be running on slack time, i.e. as the number of silver paths increase, the CPU available to the bronze path will diminish. In this manner, the transcoder's administrator has expressed the policy that the bronze path's performance is unimportant compared to the common-rx, gold and silver paths.

For the all-in-one version of the transcoder, the task is allocated 85ms/100ms allowing it to monopolise almost all the machine's resources. The receiver records the average bandwidth achieved by an average gold (i.e. 192kbit/s) and silver (128kbit/s) stream, and the average bandwidth of the single bronze stream (ideally 32kbit/s).

Figure 4 shows these bandwidths both for the all-in-one and the path-based settings. Ideally, all the lines should be horizontal and co-incident, which would indicate that regardless of offered load, the streams continue uninterrupted. However, it is clear that the all-in-one design suffers large amounts of loss as the load increases. In comparison, the path-based gold and silver streams continue almost unhindered, all the losses being concentrated on the bronze stream.

6. Conclusion

We described how Expert binds protection domains to code modules and lets threads tunnel into these "pods", allowing fine-grained memory protection. Expert's efficient and flexible tunnelling support allows this protection to be used, even in the kinds of I/O-intensive applications typical of Active Network nodes. Expert's use of paths is well-suited to scheduling the resource consumption of such I/O-driven applications.

In an example application memory protection added a cost of between 2%-6%. The example also showed the benefits arising from correct scheduling of the CPU expended in processing media streams: under high load the CPU allocations of valuable streams were protected by sacrificing the performance of others. These features of Expert should make it an attractive base for running multiple Active Network Execution Environments.

6.1. Future Work

Nested pod invocations set up concentric rings of protection; relaxing this requirement would allow more flexible security policies, but makes each call slightly more expensive. Investigating the overhead involved might be worthwhile.

As yet, no Execution Environments have been ported to Expert. The RCANE system [12] showed how standard EEs perform when run over an OS with direct support for quality of service and tight memory protection; we expect similar results for Expert, with the additional benefit of being language-neutral.

Current work on Expert focuses on the uni-processor case; however workstation-based routers of the future are likely to have either multiple symmetric CPUs or a hierarchy of CPUs at a variety of distances from the data-path. Extending Expert to these multi-CPU machine architectures is an obvious step.

6.2. Acknowledgements

I would like to thank Jonathan Smith at the University of Pennsylvania, whose encouragement helped make this paper happen. I would also like to thank Tim Harris, Keir Fraser, and the anonymous reviewers for their helpful feedback.

References

[1] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pages 267–284, Colorado, December 1995.

[2] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. Lightweight remote procedure call. *ACM Transactions on Computer Systems*, 8(1):37–55, February 1990.

[3] Brian N. Bershad, Richard P. Draves, and Alessandro Forin. Using microbenchmarks to evaluate system performance. *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 148–153, April 1992.

[4] Austin Donnelly. *Resource Control in Network Elements*. PhD thesis, Cambridge University Computer Laboratory, January 2002. Also available as CUCL Tech. Rep. 534.

[5] Marc E. Fiuczynski, Richard P. Martin, Tsutomu Owa, and Brian N. Bershad. Spine: A safe programmable and integrated network environment. In *Proceedings of Eighth ACM SIGOPS European Workshop*, September 1998. See also extended version published as University of Washington TR-98-08-01.

[6] Bryan Ford and Jay Lepreau. Evolving Mach 3.0 to a migrating thread model. In *Proceedings of the 1994 Winter USENIX Conference*, pages 97–114, January 1994.

[7] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *Proceedings of the USENIX Summer Conference*, pages 147–159, Cincinnati, OH, June 1993.

[8] Timothy L. Harris. *Extensible virtual machines*. PhD thesis, Computer Science Department, University of Cambridge, April 2001. Also available as CUCL Tech. Rep. 525.

[9] Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A practical multi-word compare-and-swap operation. *Proceedings of the 16th International Symposium on Distributed Computing (DISC 2002)*, 2002.

[10] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[11] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP-14)*, pages 175–188, Asheville, NC, December 1993.

[12] Paul Menage. RCANE: A Resource Controlled Framework for Active Network Services. In *Proceedings of the First International Working Conference on Active Networks (IWAN '99)*, volume 1653, pages 25–36. Springer-Verlag, 1999.

[13] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, April 2001.

[14] David Mosberger and Larry L. Peterson. Making paths explicit in the Scout operating system. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI'96)*, pages 153–167, Seattle, Washington, October 1996.

[15] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, Paris, January 1997.

[16] Larry L. Peterson, Scott C. Karlin, and Kai Li. OS support for general-purpose routers. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 38–43, March 1999.

[17] Jonathan S. Shapiro, David J. Farber, and Jonathan M. Smith. The measured performance of a fast local IPC. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems*, pages 89–94, Seattle, WA, November 1996.

[18] Oliver Spatscheck and Larry L. Petersen. Defending against denial of service attacks in Scout. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, New Orleans, Louisiana, February 1999.

[19] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *ACM Computer Communications Review (CCR)*, 26(2):5–18, April 1996.

[20] Ken Thompson. Unix implementation. *Bell System Technical Journal*, 57(6, Part 2):1931–1946, July/August 1978.

[21] David J. Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proceedings of the 1st IEEE Conference on Open Architectures and Network Programming (OPENARCH '98)*, April 1998.

[22] Maurice V. Wilkes and Roger M. Needham. *The Cambridge CAP computer and its operating system.* Elsevier North Holland, 52 Vanderbilt Avenue, New York, 1979.