AsyMOS - An Asymmetric Multiprocessor Operating System

Steve Muir, Jonathan Smith
{sjmuir,jms}@cis.upenn.edu
Distributed Systems Laboratory
University of Pennsylvania, Philadelphia, PA 19104-6389
Phone: (215) 898-0618, Fax: (215) 573-2232

Abstract

As the role of the computer as a communications device increases, we must reexamine the role an operating system plays in managing resources to support users. In support of general purpose computation, symmetric multiprocessing has generally proven better than attached processors, master/slave, or other configurations.

In this paper, we examine a different approach, Asymmetric Multiprocessor Operating System (AsyMOS), which applies a subset of available processors toward supporting an abstraction of a virtual 'smart device'. As a software solution, AsyMOS is able to exploit the cost/performance advantages of sharing memory and packaging that accrue to small scale SMPs, while tracking processor performance much more tightly than front-end processors can.

The ability to move OS functionality into the 'smart' device is demonstrated in the context of a network subsystem. Application-specific resource management is facilitated by the exporting of interfaces directly to applications.

A prototype implementation of the architecture running on commodity hardware demonstrates quantitative advantages over a traditionally structured SMP operating system and provides a framework for further research into functional devolution.

Keywords: AsyMOS, asymmetric, multiprocessor, operating system, architecture, network, device

1 Introduction

A recurrent theme in computer systems research is the bottleneck presented by I/O devices in modern systems. One approach which has met with a degree of success has been to make the device itself 'smarter', transferring some of the OS functionality onto the device itself in order to increase parallelism in the system.

While such approaches often provide short-term benefits to system performance, their use of custom hardware to provide the device with 'smarts' often proves to be their undoing as they fall behind the rapidly increasing power of general purpose CPUs and new system architectures.

We propose a new architecture for an asymmetric multiprocessor operating system, AsyMOS, which logically attaches general purpose CPUs to devices as a means of making those devices 'smarter'. AsyMOS runs on commodity\(^2\) SMP systems without being tied to specific hardware devices, thus advances in the architecture of these systems and the component CPUs will be passed on directly. Additionally, these systems have a much more favourable price/performance ratio than more specialist e.g. workstation, architectures, leading to the gradual replacement of the latter in many environments.

A different approach to increasing the performance\(^3\) of devices has been driven by the realisation that the interfaces presented by traditional operating systems often hide too much of the device functionality behind high-level abstract interfaces e.g. BSD Unix's sockets model, the standard filesystem read() and write() interfaces. Although fine for general purpose use, these interfaces are unsuitable for applica-

---

\(^1\)This research was supported by DARPA under Contracts N00014-96-C-852, MDA972-95-140013, and DABT63-95-C-0073. Additional support was provided by the AT&T Foundation, and the Hewlett-Packard and Intel Corporations.

\(^2\)Initially Intel multi-processor PCs, but the architecture is portable to any SMP system.

\(^3\)Performance here covers a variety of metrics, including throughput, latency, and QoS.
tions with strict performance requirements e.g. audio/video playback.

Thus it has become popular to give applications a degree of control over their own resource usage. Operating systems instead provide only some minimal level of functionality necessary to share devices among multiple applications. The remaining functions which would normally be provided by the operating system e.g. filesystem implementation, network protocol stack, must be provided by the applications or shared libraries.

An alternative mechanism for providing application-specific resource management is extensibility. This allows applications to extend the functionality of the operating system with their own fragments of code. The operating system uses various mechanisms to guarantee that different applications’ extensions cannot affect each other except in authorised ways.

The AsyMOS architecture provides both of these mechanisms so that devices may be used most efficiently. Instead of devices being accessed through device-specific drivers, AsyMOS presents a variety of functional interfaces to the operating system which allow the OS to access the device in the most appropriate manner. This is somewhat similar to hardware devices which provide both PIO and DMA interfaces to the operating system.

These functional interfaces are also exposed to applications so that they may directly access the ‘smart’ devices. Both applications and the OS are able to download extensions onto the device processors, thus allowing for dynamic partitioning of functionality between the device and the OS.

2 Proposed Architecture

The structure of a traditional SMP operating system is shown in Figure 1.

A group of CPUs, in this case 4, share access to a number of I/O devices (disk, Ethernet) over the system bus4. The CPUs access a single region of shared physical memory via a memory bus which is usually both faster (clock speed) and wider (in bits) than the system bus. In the Intel Multiprocessor Architecture [Intel96] the processors can also send inter-processor interrupts via a dedicated bus.

The operating system, usually a standard uniprocessor OS modified to support multiple CPUs, views each CPU as functionally equivalent. Any application can be executed on any CPU, although for reasons of efficiency the OS may pin an application to a given processor. Any processor can initiate I/O, but usually only a single processor handles interrupts.

Whilst this approach potentially provides the most efficient use of the processors for computational tasks, it has two main drawbacks:

- The OS kernel must implement some form of concurrency control to protect shared data structures. A fine-grained approach is complex, particularly to retrofit to a uniprocessor OS, so many systems implement a very coarse-grained protection mechanism, reducing parallelism between CPUs.

- When an application accesses a device via the OS the application’s code may be forced out of the CPU’s primary cache by the necessity to reference a large amount of device driver code.

The AsyMOS architecture addresses these problems and provides other enhancements by partitioning the set of CPUs into functional groups (Figure 1).
As shown conceptually in Figure 2, AsyMOS partitions the system’s processors into functional groups. Here two processors are used as application processors, APs, and two as device processors, DPs. The two DPs are further subdivided into a network processor and a disk processor.

Each DP is associated with one or more devices, usually all of the same class e.g. network, disk. The combination of device and processor is seen by the native OS as logically a single ‘smart’ device. Although the device is still physically connected to the system bus, and hence accessible by the application processors, the OS only accesses the device through the associated processor.\(^5\)

A functional group may contain multiple processors and/or be associated with multiple devices e.g. a single network processor may control multiple Ethernet cards, or two disk processors may control a large array of disks. In order to simplify the device processor architecture, a processor assigned to a functional group becomes dedicated to that task. Hence only application processors run user-level applications. Whilst this reduces the number of CPUs available for computational applications, if the system workload has a high enough proportion of I/O then the overall performance will be the same, possibly even higher, depending on the impact of the AsyMOS enhancements. The legitimacy of this point is further discussed in Section 5.

Application processors and device processors communicate via two mechanisms—inter-processor interrupts and shared memory.

Inter-processor interrupts provide a relatively low latency means of communications, particularly from DP to AP where the AP may be executing arbitrary code.

Since the processors all access memory over the high-speed memory bus, and the hardware guarantees cache consistency, shared memory provides both lower-latency and higher throughput communication. The device processor polls shared memory when idle so as to provide the lowest latency means of communication with the APs. This mechanism also allows applications to communicate directly with the AP without having to enter the native OS.

### 2.2 Benefits of the AsyMOS Architecture

1. Since device processors only interact with devices and the native OS and never run user-level applications they do not need access to the majority of the OS functions. Instead, they run the AsyMOS lightweight device kernel, LDK (see Section 2.3).

2. All device-specific code is moved out of the native OS and into the LDK. This reduces both the working set of the native OS, thus lowering cache contention, and the coordination required between devices and the OS.

3. The device processor handles all interrupts raised by its associated devices. By coalescing interrupts (see Section 4.1) the application processor need be interrupted much less frequently.

4. Parts of the native OS functionality can be offloaded onto the device processor (see Section 2.4).

5. Applications can dynamically download functions onto the device processor. In contrast to the transfer of OS functionality, which is a static division performed when the native OS and LDK are compiled, applications can also download fragments of code onto the DP at run-time.

---

\(^5\)On a system with multiple busses it may be possible to physically isolate devices from the application processors.
2.3 The Lightweight Device Kernel

AsyMOS gains many of its performance benefits over standard operating systems from the nature of the LDK. The LDK serves two purposes—handling interrupts from its associated devices, and communicating with the native OS.

- Since it only has to handle devices of a certain class and communicate with the native OS, the LDK has no need for file system, terminal, scheduling, or process management functions.

- By virtue of being a single task which always runs on a fixed processor it never has to be context switched, thus eliminating the (typically large) overhead which other kernels incur when switching between tasks.

- The LDK always runs at the most privileged level, significantly reducing the overhead of invoking an interrupt handler on some architectures e.g. Intel Pentium.

- It is not part of the native OS, thus removing the need for much of the concurrency control which impedes parallelism in the native OS. Data structures common to both LDK and native OS are modified to support safe and efficient concurrent accesses by both.

These last two items combine to reduce the overhead of handling an interrupt, thus reducing device interrupt latency.

All four of these points taken together result in a much smaller code footprint for the LDK, reducing cache contention on the device processor and hence increasing performance.

Whilst the structure of the LDK leads to many performance advantages over the native OS, the biggest benefits are perhaps to be gained by the transferral of functions from the native OS onto the device processor.

2.4 Functional Devolution

As shown in the Jetstream and Osiris projects, one way to increase the application-to-application throughput of a network device is to perform common data-path manipulations in the device itself. This idea lies at the heart of the AsyMOS architecture.

Although AsyMOS is a general operating system architecture and thus not specific to any particular type of device, consideration of a concrete example will help to show the advantages of this functional devolution. Since network performance is one of the most studied examples let us consider which functions could be moved from the native OS onto the device processor:

1. **Checksumming.** It is well-known that one of the bottlenecks in network protocol stacks is checksumming of data, but that performing the checksum while copying the data or on the device itself can alleviate the problem. It would therefore seem like a good candidate for moving onto the DP.

2. **Demultiplexing.** One of the conclusions that both the Jetstream and Osiris projects came to was that low-level demultiplexing of packets is highly advantageous. As well as increasing network throughput it also allows the operating system to more accurately account for network usage by applications.

Both these projects used hardware assistance to provide efficient low-level demultiplexing, an option not available to AsyMOS. However, the power of the DP coupled with the efficiency of packet-filtering technology [Engler96] means that packets can be demultiplexed to end-points sufficiently rapidly that demultiplexing should also be considered for implementation on the DP.

3. **Reassembly of fragmented packets.** As one way of coalescing interrupts this is immediately attractive for provision on the DP. It is also attractive from the native OS viewpoint as the native OS then only receives complete packets and so can be unaware of network fragmentation.

4. **ARP processing.** The whole of ARP’s functionality could be offloaded onto the DP, including the sending and receiving of requests, and maintenance of the ARP tables.

5. **Device level bridging and routing.** An AsyMOS system configured as a bridge or router could perform either of these functions on packets without service from the application processors.

One of the major advantages of the AsyMOS architecture is that it is completely flexible, allowing all of these possibilities and more to be implemented and tested quantitatively in a purely software environment. This offers a much shorter development cycle than hardware alternatives such as FPGAs, and is not restricted to expensive custom hardware with complete computing engines onboard.
### 3 Implementation

An implementation of AsyMOS is currently being developed and has sufficient functionality to allow some of the key assertions about the architecture to be quantitatively tested (see Section 4).

Although the prototype offers only a very limited subset of the functionality described previously, it provides a starting point and much of the basic technology necessary for a fuller implementation.

This initial implementation is based upon version 2.0.30 of the Linux kernel and runs on a dual-processor 166MHz Pentium PC. The device processor controls only a single Ethernet card, a 3Com 3C905 100BaseTX Fast Ethernet adapter. The pseudo-device consisting of the device processor and this card is known as NetP. The structure of this pseudo-device as seen by Linux is shown in Figure 3.

Linux's somewhat rudimentary SMP support has been extended in the following ways to support AsyMOS:

- Multiprocessor interrupt distribution has been added, utilising the Intel I/O Advanced Programmable Interrupt Controller (APIC) to route interrupts to processors other than the boot processor (the normal Linux mode of operation).
- A message-passing mechanism is provided to support communication between the DP and AP. It uses inter-processor interrupts and allows for simple cross-processor procedure calls.
- The scheduler is extended to allow certain processors to be designated as non-schedulable. This facility is used to prevent the Linux scheduler from attempting to schedule user-level processes on the device processor.

These modifications provide a framework upon which to implement the lightweight device kernel and the NetP pseudo-device(s). The LDK is currently implemented as a Linux kernel thread which handles device interrupts and communicates with the application processor(s).

A modified version of the 3c905 device driver is used by the LDK to control the Ethernet card. Calls to functions which are logically part of the Linux kernel i.e. those functions which read and/or write kernel state, are translated into DP-to-AP messages.

A Linux interface to the NetP pseudo-device has been implemented which translates device-specific calls to the pseudo-device into the appropriate invocations of device processor functions. These functions are executed by the LDK in response to the appropriate AP-to-DP message.

### 4 Preliminary Results

Whilst the current implementation of AsyMOS is still very much a prototype, it has nevertheless been possible to perform some preliminary experiments to determine the effectiveness of the architecture.

Both tests were conducted on a dual-processor 166MHz Pentium PC from Xi Corporation, running RedHat Linux 4.1 with either version 2.0.30 of the Linux kernel or the prototype AsyMOS implementation. The Pentium processor has split L- and D-caches, each 8k in size, 2-way set-associative with 32-byte line size [Intel97a].

#### 4.1 Interrupts Delivered to Application Processor

One of the ways in which AsyMOS aims to make devices seemingly smarter is by delivering fewer interrupts to the application processor(s). This can be done in two ways:

1. Coalescing of interrupts so that multiple interrupts to the device(s) only cause a single event notification to be sent to the application processor.
System | Interrupts/5 mins. 
--- | --- 
Linux | 774.2 
AsyMOS | 731.8 
AsyMOS w/ Garbage Filter | 131.0 

Table 1: Interrupts taken by Application Processor in a 5-minute period (average)

2. Processing of packets purely at the device level. This is possible in two common cases: if the driver can determine that the OS will discard the packet, or if the packet is of a type which can be handled entirely by the device e.g. ARP, ICMP ping.

Of these possibilities, the current implementation only performs early discard of ‘uninteresting’ packets. To this end the driver has been equipped with a garbage filter which discards all IPX packets and ARP requests for other hosts (both of which are broadcast on the Ethernet).

The experiment measured the number of interrupts taken by the application processor due to background traffic only on the Distributed Systems Lab 100BaseTX LAN. The measuring host was idle, running only the normal Unix daemons, and the number of interrupts taken was measured over 5 periods, each of 5 minute duration, then averaged. The results are shown in table 4.1.

These results clearly show that a very simple filter (4 header field comparisons) at the device level can reduce the number of interrupts delivered to the application processor by about 80%. Obviously this figure is dependent on the characteristics of background traffic on the LAN, but the DSL network does not seem to be particularly unusual in its configuration (mostly Unix machines, a few PCs for word-processing, not particularly heavily loaded).

As expected the rates for Linux and AsyMOS are essentially the same due to the structure of the Linux networking code (every ‘packet received’ interrupt from the card results in a message being sent to the OS to process that packet; every ‘ transmit completed’ interrupt causes another message to be sent to the OS to indicate that transmission has finished).

This technique can be extended to more complex garbage filtering, including (e.g.) sending ICMP error messages in response to TCP and UDP packets sent to ports without listeners. In this way, AsyMOS presents an effective mechanism for tackling the problems of receive livelock [Mogul95] and many denial of service attacks.

4.2 Cache Contention due to Device Driver Code

As stated earlier, another goal of the AsyMOS architecture is to reduce cache contention on the application processor. This can be measured in a number of ways, but one valid metric is the number of I-cache misses taken in order to send a packet and receive a reply.

The disparity between L1 cache performance and system memory is well known [Hennessy96a], and, additionally, inefficient I-cache utilisation has been shown to be a major cause of poor performance in protocol stack implementations [Blackwel96].

To reduce the amount of common (device independent) code referenced by the OS, this measurement was taken between two code locations inside the kernel. The I-cache miss counter [Intel97b] was reset just before the OS calls the device driver’s hard_start_xmit function (which forces the driver to send a packet) and then read just after the device driver calls the netif_rx function (the upcall used to process a received packet). The code in between these locations represents the following sequence of actions:

1. Device driver sends packet. On the 3Com 3c905 card used in our implementation this merely initiates a DMA transfer—completion of packet transmission is signalled by an interrupt.
2. OS returns to running application code.
3. ‘Transmit complete’ interrupt is handled by OS.
4. Reply packet arrives, causing another interrupt. The device driver reads the packet, then calls the netif_rx upcall to process it.

The packets to be sent were generated using the standard ‘ping’ program in two configurations. First, packets were generated at a rate of one per second, with a total of 50 packets being sent. The system is idle in between a reply arriving and the next packet being sent. Second, 100 packets were sent in the ‘flood’ mode, which causes a packet to be sent as soon as a reply is received for the previous packet sent. This prevents the system returning to the idle state in between packet transmissions, though not in between sending and receiving a reply.

In both cases the measurements were averaged over all but the first two packets (to discount discrepancies
<table>
<thead>
<tr>
<th>System</th>
<th>Slow ping</th>
<th>Flood ping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>380.6</td>
<td>355.6</td>
</tr>
<tr>
<td>AsyMOS</td>
<td>297.9</td>
<td>250.5</td>
</tr>
</tbody>
</table>

Table 2: Application Processor L-cache misses per send-receive pair (average)

caused by the initial ARP packet) i.e. 48 packets in the first test, 98 in the second. Each test was repeated 5 times and the results averaged, as shown in Table 4.2.

These figures show that AsyMOS does indeed reduce cache contention on the Application Processor by about 20-30% in this test. Since micro-benchmarks are often (rightly) treated with some scepticism [Hennessy96b], [McVoy96], we need to look into the reason for the reduction. In fact, it is wholly due to the replacement of complex device-specific code with the more compact AsyMOS stub functions. It is thus reasonable to conclude that AsyMOS is likely to show some degree of performance increase over Linux.

5 Conclusions

Measurements of a traditional multiprocessor UNIX OS [Bach84] show that the system typically spends about 30-60% of the time executing inside the operating system, a large proportion of which is performing I/O. Furthermore, computers are more frequently being used in environments where I/O is more important than computation e.g. network computers, Web surfing.

These two facts combined lead to the conclusion that the I/O proportion of a modern system’s workload is sufficiently high to make AsyMOS a viable architecture.

The architecture is particularly well-suited to the management of network devices, which are usually relatively dumb and require a large amount of care and attention. By dedicating a fraction of the computing resources to looking after these devices we remove the responsibility from the operating system, increasing both overall system and network performance.

The preliminary measurements detailed show that the AsyMOS architecture does provide quantitative benefits over a traditional operating system, even though the implementation used to gather these results was very much a prototype. Despite these tests being relatively low-level, we believe that the benefits they demonstrate will be conferred onto communications applications running on an AsyMOS system.

It might be argued that the extra processor(s) used for communications support in AsyMOS would be more ‘effectively’ used as general purpose processors. The definition of ‘effective’ must be accompanied by a workload definition and metrics, and for us, effective means that communications-oriented applications must be supported efficiently.

With the AsyMOS architecture, we are able to cost-effectively offload many tasks of such applications. This architecture negates many of the key weaknesses (complexity, resource contention, etc.) of a uniprocessor OS e.g. Linux, running on SMP hardware, and also provides for greater resource accountability and control. This combination of benefits, we believe, will allow communications-oriented applications to be more effectively supported by AsyMOS than by a symmetric OS.

As the number of processors in a multiprocessor system increases, the problems of resource contention and concurrency control become ever more significant. AsyMOS is therefore even more attractive on those systems, where reduction of these effects provides increased performance relative to the uniprocessor case. Additionally, a greater degree of flexibility is possible when adjusting the device versus applications processor balance to the workload.

The architecture provides a great deal of flexibility to the operating system designer in order that devices can be utilised most efficiently. Whilst previous approaches to this have often required expensive custom hardware, AsyMOS runs on commodity systems. By not tying the device intelligence to a specific custom device the architecture will not be left behind by advances in CPU technology.

As such, we believe that AsyMOS has two important roles to play: as a testbed for the investigation of new system architectures, particularly networking subsystems, and as an operating system suitable for deployment on network-intensive systems.

5.1 Further Work

Since the current implementation of AsyMOS is still in a very basic form, the most pressing task is to refine the implementation to provide the functionality detailed in Section 2.

The first step of this process will be the development of the lightweight device kernel, which is central to the architecture and will provide many advan-
tages over the current implementation’s execution of a modified Linux kernel on the device processor.

Once the lightweight device kernel is in place it is expected that the next area of attention will be the interfaces between the device processor and the general-purpose operating system. It is hoped that by providing a suitable user-level interface much of the networking subsystem can be moved into user-space, thus allowing easier experimentation with the partitioning of functionality between user-space libraries, operating system and device processor.

At the same time the capability to dynamically download application code onto the device processor(s) will be investigated to see how this flexibility can be safely provided to applications.

Looking into the longer term, AsyMOS may prove to be an ideal platform for Active Network nodes. In an environment where network throughput is more important than general computational power the AsyMOS architecture should prove to be ideal and provide most benefit over other MP operating systems.

Finally, it should be emphasised once again that although the focus of this paper has been on the relevance of AsyMOS to networking systems we hope to be able to apply many of the ideas presented in a more general manner. The key concept of utilising CPUs to provide intelligence for devices may prove to be applicable to other classes of device. For example, a device processor associated with an array of disks could be used to process data being read from and written to the array, providing on-the-fly encryption and compression.

\section{Related Work}

Both smart devices and application-level resource management are topics which have recently been investigated by a number of researchers. Vertical and extensible operating systems in particular are currently hot topics in OS research.

\subsection{Shifting the Burden of I/O}

Mainframe computers have for a long time used channel controllers as a means of removing the burden of controlling a relatively slow device from a much faster CPU [IBM70]. These channel controllers usually take the form of smaller computers attached directly to the devices they are responsible for; they are somewhat analogous to AsyMOS’s device processors.

More recently, the advent of high-bandwidth physical networks e.g. ATM, has forced the networking community to look into ways of overcoming the bottlenecks of current workstation architectures in order to provide that bandwidth to applications. Two interesting approaches were Hewlett-Packard’s Jetstream/Afterburner project [Watson93] and Bellcore’s Osiris ATM adapter [Davie91], [Druschel94].

The former opted to provide common data-path operations (checksumming and low-level demultiplexing) in hardware without any programmability; the latter project offered a completely programmable processing engine (CPU and memory) on the adapter card which could be programmed as desired by the OS.

Whilst offering some of the functionality of the HP offering, AsyMOS is more closely related to the Osiris project. However, an important difference is AsyMOS’s use of a general-purpose CPU as the device processor, allowing us to ride the tidal wave of processor advances and not be left to drift on the gentle ripples of I/O board improvements.

\subsection{User-level Resource Management}

The benefits of application-specific resource management have been demonstrated in a number of contexts e.g. filesystem hints [Patterson94], the differing requirements of continuous media (audio and video) and batch traffic (FTP, NFS, etc.) in the face of lost and misordered data [Clark90].

Three projects which are based around this notion are the University of Cambridge’s Nemesis project [Leslie96], MIT’s Exokernel [Engler95], and the University of Washington’s SPIN project [Bershad95].

Nemesis and Exokernel both adhere to the vertical operating system model whereby the OS provides only the minimal functionality necessary to share devices. However, they differ in their perceptions of what this minimal level of functionality is—whilst Nemesis implements what are put forward as the three key functions of protection, translation and multiplexing, the Exokernel only performs multiplexing of hardware between tasks (which logically includes some degree of protection).

SPIN is perhaps the best example of an extensible OS. It allows applications to extend the operating system kernel with functions written in a type-safe language (Modula-3).
6.3 Asymmetric Software on Symmetric Hardware

The Softnet project [Forschheimer81], an early packet radio network developed at the University of Linkoping, Sweden, proposed a distributed network of computational nodes, each of which consisted of a dual-processor computer. One processor (the node processor) executed user tasks and was responsible for system management, the other (link processor) was used to control the radio transceiver and the interface to the node processor.

Although conceptually similar to the Softnet architecture, AsyMOS differs fundamentally in that the functionality of the device processor can be much more readily changed. The operating system is able to tailor the functions executed by the DP as appropriate to the current workload. In contrast, the link processor in the Softnet architecture executed a ROM-based program which performed specific tasks related to the control of the radio transceiver.

6.4 Active Networking

Looking to the future, AsyMOS provides an attractive architecture for the Store—Compute—Forward elements which form the nodes of Active Networks [Tennenhause97]. Firstly, switchlets\(^\text{\textsuperscript{7}}\) can be executed directly on the device processors, leaving general purpose processors to handle only the overall system control functions. Secondly, an AsyMOS device processor can process packets much more efficiently than a general purpose OS due to the benefits of the lightweight device kernel (see Section 2.3).

7 Acknowledgements

Steve Muir would like to thank Jonathan Smith for a great deal of useful input, and not objecting to have his interrupts disabled at times. Ian Pratt and Paul Barham at the Cambridge University Computer Laboratory provided feedback on the architecture and implementation. Jonathan Shapiro gave technical insights into the Pentium architecture and made encouraging noises whilst the project was in its infancy.

Jonathan Smith would like to thank the Cambridge University Computer Laboratory for hosting him while some of this work was performed, as well as the U.K. EPSRC for their support.

\(\text{\textsuperscript{7}}\)A switchlet is an active packet—a packet whose payload is a program to be executed on the Active Network nodes

8 References and Bibliography

References


Leslie96 Ian Leslie et al.; “The Design and Implementation of an Operating System to Support Distributed Multimedia Applications”, IEEE Journal on Selected Areas in Communications, Volume 14, Number 7 (September 1996), pp.1280–1297


