Angel: Resource Unification in a 64-bit Micro-Kernel

Kevin Murray†, Tom Stiemerling‡,
Tim Wilkinson†and Paul Kelly‡

†Systems Architecture Research Centre,
Department of Computer Science, City University,
Northampton Square, London EC1V 0HB,
ENGLAND.

‡Department of Computing,
Imperial College, London,
ENGLAND.

4 June 1993

Abstract

The appearance of 64-bit processors allows a new approach to microkernel design — a single unified address space. This paper describes this kind of approach as adopted in Angel.

From our experience with a message passing microkernel MESHIX, we discovered that a multi-address space, POSIX architecture is unsuitable for general parallel applications development. Angel was therefore designed to provide a more flexible environment. Central to these aims is a simplification of resources. This is achieved through the use of a single address space holding volatile and persistent data and all processes, and which absorbs communication to remove the distinction between local and remote machines. In addition, a simple user-level, first citizen, threaded process structure with software interrupts has been developed. The resulting kernel, written in C++, is compact and simple yet offers fully protected services and is more flexible than many of its contemporaries.

A single address space provides some unique problems — most importantly the need to provide POSIX compatibility. We have investigated this, and developed a modified compiler. This is capable of generating code from unmodified POSIX applications which may be forked to a new address but still executed correctly. This avoids compromising the single address space and only entails a small performance penalty (approximately 5%).

A first generation microkernel is now complete and runs as an emulation on the Sparc. A native system will soon be available.[1]

1. This research has been funded by a grant from the UK Science and Engineering Research Council under grant GR/G28277.

1 Introduction

In the mid eighties, the System Architecture Research Centre at City University developed a message-passing, UNIX compliant kernel (*Meshix* [1]) for our scalable distributed memory architecture (*Topsy* [2]). This kernel is a microkernel based, message passing operating system, relatively typical in structure to many current systems such as MACH [3], Chorus [4] and Amoeba [5]. Over the last two years we have been examining its structure and performance in a critical manner. This has demonstrated a number of issues that have not yet been addressed by most current microkernel architectures. Additionally, we believe that a POSIX centred architecture [6] does not provided the best model for exploiting the resources available in a parallel machine.

During the design of *Meshix* there have been several major advances in the field of computer architecture. This has enabled a new approach to be taken in microkernel design. First, the advent of 64-bit address space processors (eg. DEC Alpha [7], MIPS R4000 [8] and the custom chip in the KSR-1 [9]). Second, the communication devices available to connect computers have dramatically increased in speed (approaching one gigabit/sec) [10].

The goal of our research is to find the best way to overcome the limitations identified with *Meshix* (and other microkernels) whilst exploiting improved computer hardware and architecture. Resource unification provides an approach to achieving this goal. This paper examines the reasoning behind resource unification, and why the new generation of processors and networks make it feasible. An overview of the approach is then given, followed by our implementation *Angel* [11], a Single Address Space Architecture (SASA). The benefits this gives are outlined as well as the problems introduced.

2 Shortcomings of Meshix and other Microkernels

The original goal of *Meshix* and the *Topsy* architecture was to produce a scalable, parallel UNIX multicomputer with a simple, modular, message passing microkernel. The objective of a scalable parallel message passing computer is shared with a number of other research systems and projects [12, 13, 14]. To an extent this was achieved, although some aspects of performance were disappointing. To discover why *Meshix* did not live up to expectations, a detailed performance evaluation was performed. The following sections outline some of the conclusions drawn about both *Meshix* and POSIX architectures.

2.1 Performance of IPC and RPC

Monolithic UNIX implementations generally have superior IPC performance in comparison to modular message-passing designs such as *Meshix*. The classical approach to improving RPC performance, the lightweight RPC optimisation developed for the DEC Firefly system [15], requires non-trivial modification to the operating system's structure. The approaches used by other groups to improve IPC performance, notably Chorus [16], although adopting LRPC techniques, also uses other methods. These include replacing context-dependent addresses with unique addresses, so speeding up message delivery whilst reducing security, combining mutually trusted servers into a single address space, so reducing context

switches, and by placing all of the IPC management into the microkernel. It takes all of these extensions and modifications to produce a system whose performance is comparable with, though not equal to, monolithic systems.

2.2 Processor Cache Utilisation

As processor cache architectures become more complex, the costs in time and complexity of changing the virtual-to-physical address map become more serious [17]. This causes the designer to avoid the benefits of complex hardware architectures (with their performance advantages), and to use simpler techniques, such as copying instead of re-mapping, to dodge the associated problems (e.g. cache and TLB flushing). This seriously impacts potential performance.

2.3 Replicated Software Caches

Meshix provides several *ad hoc* caches, such as a TLB cache, a page cache and a filesystem block cache. Ideally there should be a single instance of these similar mechanisms. This would not only simplify the kernel, but should improve overall cache utilisation and flexibility.

2.4 Parallel Processing Support

UNIX is not an ideal system to use when trying to exploit a parallel machine, primarily because its process model is one of distinct heavyweight processes. For efficient parallel programming a lightweight, flexible and extensible process model is required. Although a heavyweight process model can be augmented with lightweight "threads", these threads are not real first class entities in the operating system as certain system operations performed by one thread affect others.

2.5 Dynamic Load Balancing

In a scalable parallel system it is necessary to support a load balancing mechanism. This is important since work must be distributed over the machine efficiently [12]. With a general purpose computer, the load on various parts of the system, typically represented by the number of active threads, can change. To maintain the efficiency of the application or system, it is necessary to re-allocate work between processors. In many UNIX based systems it is either impossible to share threads between processes, or it is a costly and complex operation involving non-trivial kernel support [18].

2.6 Complex Data Sharing

When co-operating processes wish to share complex data structures in *Meshix*, they must be exchanged through message passing, files or shared memory segments. Since it is not always possible to guarantee that these structure will reside at the same addresses in all processes, costly packaging must take place to allow them to be interchanged.

---

3 Resource Unification

We believe that many of the problems above can be handled by use of resource unification. This section examines what this involves, how it affects the design of the operating system and how it solves the problems identified. Resource unification can be thought of as an application of the philosophy of minimalism—it tries to provide a minimal but complete interface.

3.1 Resources in an Operating System

The basic operation of an operating system can be considered to be the control, management and allocation of resources. In a typically system these will include:

- Processor Time,

- Volatile Memory (eg. DRAM),
- Persistent Storage Media (eg. disks), and
- Communication Access and Bandwidth.

Processor time is allocated by switching a processor between various processes, with limited control provided by a set of system calls (e.g. wait, pause, alarm) but ultimate control exercised by the kernel. Processor memory is the part of the unique address space that each process is allocated (controlled through various system calls such as fork, exec and sbrk). Persistent storage is seen through a totally independent interface; the filesystem (providing system calls such as read, write, open, close and mmap). Yet another interface is provided for access to the communications capability of the computer (in a multi-processor machine, this is typically via message passing or shared memory, whilst between computers it is TCP/IP).

As demonstrated, the result is numerous different interfaces, one for each resource. A kernel which supports these typically implements each independently, resulting in a larger (and potentially more bug ridden) system than one with fewer interfaces. In addition, the programmer has more interfaces to remember and handle correctly.

3.2 The Single Address Space Architecture(SASA)

Conventional filesystems exist for several reasons. First, persistent data (ie. that stored on disk) is viewed as different from transient data. Second, sharing data between programs requires that it be provided with a "name" or "address" so that it can be found again or two processes can ensure they are referring to the same data. Since the names are typically used by "humans", it is sensible to make them mnemonic. Consequently, the filesystem has persisted since it provides a clean handle on data and, with a conventional 32-bit processor, the 4 gigabyte address space is too small to hold all data (simple commercial databases are large than this).

The advent of 64-bit processors gives a truly dramatic increase in the amount of data that can be contained within the address space of a processor. It is now possible for it to contain all the persisted data of the filesystem as well as the processes' transient data and code. This forms the heart of a single address space architecture—there is a single interface to the entire memory hierarchy from processor cache to magnetic or optical disks. Providing all accessible data via a single memory interface, where each data item always resides at the same address, has various benefits:

- *Improved Data Sharing*: Since the data always resides at the same address, regardless of the process that is using them, complicated data structures (such as graphs and trees) can be easily shared without the need for "marshalling"[2].
- *Improved Cache Utilisation*: With a conventional operating system, when a context switch occurs, any cached process dependent data must be flushed. In an SASA this is no longer necessary since the data is equally valid in all processes[3]. Consequently, less data is flushed, the cache utilisation improves, and performance is increased.

2. The act of encoding the data to remove pointers.
3. Virtual memory translation will prevent accesses by unpriviledged processes.

- *Single Unified Interface*: Providing a single interface simplifies the design and maintenance of the kernel as well as applications. This single interface also aids the programmer by reducing the number of interfaces to remember.

3.3 Distributed Shared Memory

Parallel architectures come in two flavours, shared memory and distributed memory. Current work has lead to the conclusion that shared memory is a far simpler paradigm to use [19]. Fortunately for distributed memory systems, there is a technology based around distributed shared memory (DSM) which provides the appearance of shared memory [20].

This is a very attractive mechanism to support especially with the increased bandwidth and network support hardware available. However, initially it would appear to be a very inefficient mechanism when compared with a message passing system. In order to examine this, a set of detailed measurements of the *Meshix* communication system were made and the behaviour of a simple hardware assisted distributed shared memory (DSM) scheme modelled. The communications costs of DSM and message passing, and ways to reduce or eliminate them, were then compared.

The results are detailed in other papers [21, 22], but concluded that many of the costs in message passing are an inherent fact of the model, (e.g. data copying, mapping and security). By comparison, the DSM has far fewer inherent costs. Consequently, a DSM-based system should outperform a message-passing based system, all other factors being equal. This leads to the belief that DSM should be the basis of future parallel systems, in common with many other researchers and manufacturers [9, 23, 24, 25, 26].

3.4 SASAs and Parallel Computers

In a SASA, DSM techniques are used to extend the single address space from one processor to all processors in a system. This results in IPC being incorporated into the SASA. Extending a single address space across a parallel machine and making it the only means of co-operation has many benefits:

- *Unification and simplification of network*: As the single address space is the only means of communication between processors, the interface to the network is hidden behind the single address space and accessed as shared memory, rather than via an additional set of system calls.
- *Data Migration and load balancing*: As the data is shared through DSM, it is trivial to move it between processors, even when the structures are complex. Additionally, moving a thread between processors, is trivial: the thread's context need only be loaded in another processor and the thread will be running on it. As it accesses data, it will be demand paged via the DSM. Hence introducing load balancing into a SASA is far simpler than into a conventional architecture.
- *LRPC*: The optimisations of the LRPC scheme originally for single processor RPCs can be applied to multiprocessors. In essence this mechanism relies on agreeing and creating a region of memory that is accessible to both RPC parties in which parameters and results are to be stored. In a SASA this is trivial. In fact it can be taken further. If there is sufficient trust, parameters can be read direct from the client exactly as is already done in a monolithic UNIX system. With a flexible protection scheme it is possible to support a complete range of RPCs, from high performance, high trust through to lower performance, low trust.

3.5 A Protection Scheme for a SASA

Conventional operating systems provide protection in two ways:

- *Private address spaces*: Every process is given its own private address space. Therefore no process can access the memory of another.
- *Filesystem name resolution*: Whenever a process tries to open a file (resolve its name into a file handle) the operating system first checks whether or not the process is allowed to access the file.

By nature, a single address space architecture breaks both these protection mechanisms. The first is broken as all processes reside in the same address space. The second is broken since persistent data (files of old) also resides in the same address space. Another protection scheme must be provided to protect the data held in this space which prevents both malicious and accidental access to, or damage to, the data.

Additionally, current protection schemes are not sufficiently flexible. For example, consider the file "/etc/passwd" in a conventional UNIX system. Typically, only the user "root" can write to this file, though anybody can read it. In fact, a system is required whereby only the program "passwd" can read and write to the file, and a limited set of programs, such as "finger" and "login", can read it. This would allow finer control of protection and avoid "mistakes" compromising the entire system.

The above points have prompted the development of a new scheme based around domains which list the data that can be accessed. When a program seeks to access new data, those items it can already access are used to determine if it can access the new datum. Processes are not tied to domains; they may freely choose or change domains if they hold the relevant permissions. This results in a scheme in which address translation and protection are disassociated, which is as it should be since one is concerned with memory management, which is of no concern to the application, whilst the other is concerned with security, which is of concern to the application.

3.6 Fault Tolerance under a SASA

The use of a single address space enables a low cost fault tolerance scheme to be provided transparently. In essence it relies on two facts; first that all interactions are visible to the DSM system so allowing it to identify co-operating groups of processes that will be affected in the event of failure. Second, in a SASA all data is visible to the system—there is no hidden state in the kernel.

To protect against failures, periodic checkpoints of dependent process groups, determined by analysis of DSM traffic, are made to remote memory. When a failure occurs, the processes affected are rolled back to the last checkpoint and the data held on the failed node is restored from the copies made during the last checkpoint. By sensible use of intelligent checkpoint, this system can be extremely efficient [27].

3.7 Problems with an SASA

Although the adoption of a SASA has many benefits, it also produces a number of characteristic problems. These are described below.

Limited Address Space

The most commonly cited problem with a SASA in that the address space could be completely consumed. This problem does not arise with conventional filesystems since the files are accessed through an unlimited address space, the hierarchical name of the file. However, even if addresses are allocated and never released (i.e. the data they contain persisted forever) it would take over one hundred processors

working in parallel, consuming addresses at the rate of four gigabytes a second, more than a year to use a full 64-bit address space[4]. This could be considered to represent a pathological case, such as a program which deliberately fills up disk space in a conventional system, since in normal operation many addresses are recycled — e.g. process data objects when the process dies.

Fixed Sized Objects

When entities that contain data (objects) are created in a SASA they must be given a fixed length. It is not possible to allow an object to grow since it might collide with another. Thus when an object must be extended, a new larger object must be created and the old object copied into it[5].

Persisting Objects

The single address space encompasses both temporary and persistent objects. This results in problems when allocating objects and associating them with backing store. In order to guarantee that data cannot be lost due to lack of disk space, it would be necessary to preallocate disk space to all created objects. This needs to be done even if the objects are temporary otherwise paging is impossible. However, if is quite common to allocate very large objects but only make sparse use of them. This would result in the preallocation of disk space which is never used.

Alternatively, backing store can be associated with an object page when the page is first modified. This means disk space is only allocated to objects pages which are actually used. Unfortunately, this makes it possible for a write to memory to "fail" due to lack of disk space (rather like a file write to a full disk).

Needless to say, neither of these options is attractive. Currently, the latter scheme is being used whilst a better scheme is sought.

POSIX Support

Perhaps the most important obstacle in the adoption of a SASA is its inability to support a conventional POSIX-style operating system. On closer examination of the problem, it should be possible to support most POSIX programs through the use of compiler technology [28] and an associated set of libraries and servers.

3.8 Related Research

There are several research groups that have done work related to our goal of resource unification, either tackling it in a different manner or developing a similar scheme for different reasons. One area involves unifying the interface to system resources by developing a single name space in which all system entities reside. The best known example of this is Plan 9 [29, 30] which provides access to all data through a configurable filesystem and the single filesystem interface. This allows distributed data to be handled transparently but provides no coherency (making fine grain data sharing impossible) and still requires complex data structures to be marshalled before they can be exchanged.

Single address spaces have been developed by other researchers, although their major motivation is not that of unification but rather providing a simple model for data sharing — *Angel* does achieve the same goals. These systems include Psyche [31, 32], Opal [33, 34] and work at the University of New South Wales [35], all of which attempt to provide a true single address space shared by all processes. Unlike *Angel*, they all base their protection scheme around the possession of capabilities (in the work at New South Wales they are "password protected" in a manner similar to capabilities in Amoeba) rather than around the permissions which have already been acquired. Only the work at New South Wales provides a process (or

4. Alternatively, it would take well over one million full length feature films to fill this address space.
5. To speed this procedure, a copy-on-write mechanism should be provided to perform the operation.

Virtual Processor in *Angel* terminology) the ability to change protection domains with ease, or for them to be shared between processes.

4 Angel—A SASA Implementation

In order to demonstrate the benefits of the single address space architecture we have implemented an operating system using the principles described. Currently the system exists as a emulator running as a group of processes under SunOS. The code for the emulator is in the process of being moved to a native system.

4.1 The Process Model

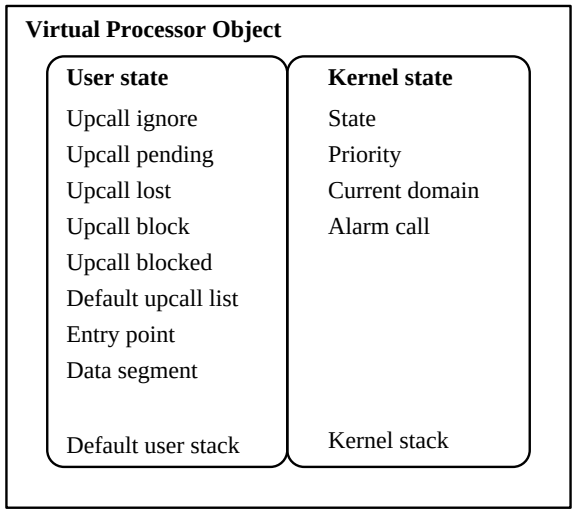

Figure 1: The VP control structure.

The process model adopted is derived from a number of systems, but has been simplified as far as possible. The distinction between single, multi-threaded and multiple co-operating processes have been eliminated; the operating system provides the same services to all.

The kernel does not support threading internally. Instead, it supports *virtual processors (VPs)* which are handled in a similar way to UNIX processes. However whenever a VP performs an action which would block (eg. a page fault) the VP is "upcalled". This activation allows the VP to decide whether it can continue with another thread rather than give up the processor to another VP. Hence, whilst threads are implemented in applications, they are still first citizens. The kernel sees the VP as an object in which it can store information about the VP and which is used by the VP to convey information to the kernel—figure 1 gives an overview of the information it contains.

Naturally many programs are not interested in such events; those which are singly threaded are not interested in page faults. The VP object allows fine control of what events are of interest it and which ones are not.

Upcalls

The upcall mechanism is used to augment the single address space provided to VPs and is the primary component of the VP object. Pure shared memory is insufficient since it provides no way to cause an action on another VP unless it is

actively waiting for it. Upcalls provide a means of one VP sending an "interrupt" to another, so long as it holds sufficient permissions.

Upcalls convey limited information: the sending VP, the upcall type, and an associated address. This address refers to any additional information required by the receiver. When an upcall is delivered the VP has the options of ignoring it, queueing it, or queueing it and taking an interrupt. It also has the option to block further upcalls. The delivery is made into an advertised upcall list. Although a default list is provided in the VP object, a larger one can be installed if required (as it often is in servers). In the event that an upcall cannot be delivered because the list is full, an "overflow" is indicated. This allows a recovery procedure to be invoked (so regenerating any lost page faults, lock request, etc.) and so prevent deadlock.

The upcall scheme may be considered as a limited message passing mechanism but with a more flexible delivery system—the VP can specify whether or not it wishes to be interrupted on a given upcall. Additionally, the fact that two VPs are on different processors, or the same processor, is irrelevant to the microkernel; the DSM mechanism handling the difference.

4.2 The Protection Scheme

A domain lists a set of objects and the access and administration attributes held on these objects. Thus a domain specifies which parts of the single address space may be utilised, and in what ways. The domain is described in an ordinary object (which is only writable by the Object Manager) and VP is free to specify which domain it wishes to execute in, and may change domains at any time, provided that it has permissions to do so.

The attributes that may be associated with an object in a domain include:

- *Readable,* the object may be read,
- *Writable,* the object may be modified,
- *Executing,* this is the code object that a VP started executing from (this may not be requested),
- *Change Permissions,* the permissions associated with the "biscuit" (see below) may be changed,
- *Inspect Permissions,* the permissions associated with the biscuit may be seen,
- *Delete Biscuit,* the biscuit may be deleted (revoking all other access to the object using this biscuit),
- *Create Biscuit,* a new biscuit with a new set of permissions may be created for the object,
- *Runnable,* this object may be converted to an executing object,
- *Upcallable,* an upcall may be performed to this object.

The scheme differs from conventional protection schemes when one considers how it allows a new object to be placed into the domain. The biscuit, a bit-string, is presented to the object manager, together with a list of permissions requested on that object. The object manager inspects the structure associated with the biscuit which lists the objects the domain must already contain, and the permissions thereon, before any given permission for this object can be granted. If the domain meets all the requirements, then the new object is added to the domain with the

permissions requested. Thus the scheme is based around the abilities that the domain already has rather than around user or process identifiers. It is worth noting, however, that it is trivial to implement a user identifier based scheme in this system simply by creating an object to represent a user. A biscuit is initially returned when an object is created; thereafter they may be made on request by the object manager if the permission *create biscuit* is held. Biscuits may be freely exchanged between processes, or stored in other objects.

In this scheme, all the protection mechanisms are "unified" at a single point. This one place could easily be subject to checks or formal verifications as deemed necessary to ensure security. Also, as there is only one point, there is no chance of unexpected interactions between protection points which might breach security. The result is a highly flexible system with increased security.

4.3 The SASA Implementation

The SASA implementation operates as an aliasing cache hierarchy. Figure 2 depicts a simple four module system; at the top of the hierarchy is the single address space as seen by VPs, at the root are the disks. In order to explain its operation, consider the following example. A VP makes an access in the single address space. This corresponds to a virtual address on a specific processor. This address is translated to a physical memory address (if possible), and the access if fulfilled.

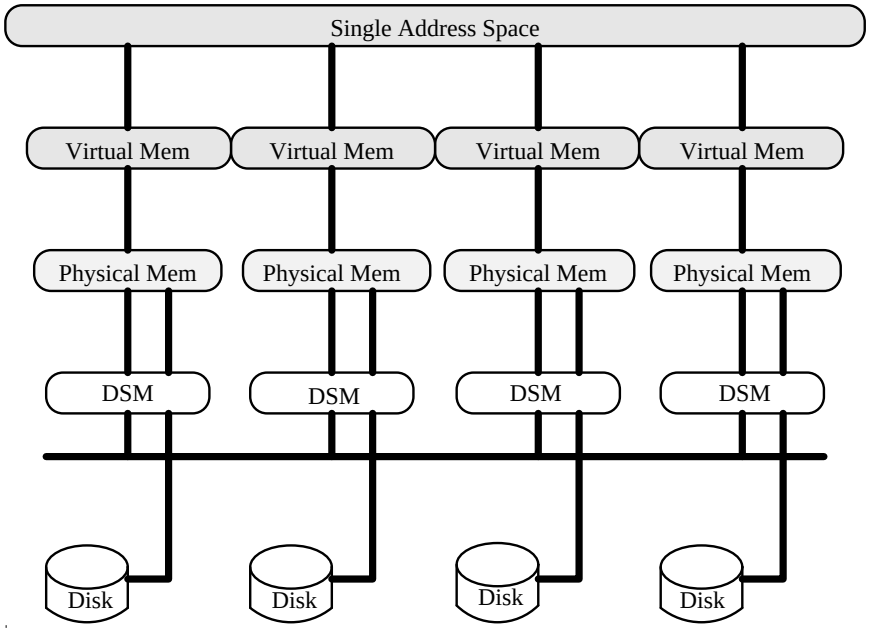

Figure 2: The virtual memory hierarchy.

If translation is not possible, the microkernel is invoked to establish one. Unfortunately this is not trivial since the copy-on-write (COW) mechanism *Angel* needs results in the mapping between a virtual address and a datum being many-to-one rather than one-to-one. This is because the COW mechanism can causes a datum to appear in two or more places. Therefore, first the virtual address is unaliased into a *pageID*, which does have a one-to-one mapping with data. Next, the pageID is used to search for the associated data. This is done first in local memory (since the pageID may already exist), then on local disk, and finally on the DSM network (causing each remote node to be searched in a similar manner). If found, the data

is transferred to local memory. If not found, the pageID is a new one, and a zero filled page is allocated. Finally, a translation is established and the faulting VP restarted. Note that if data is not initially found in local memory, a VP has the option to receive two upcalls. The first is to allow the scheduling of another thread whilst the mapping is established, the second to inform the VP that the mapping has been established.

By treating the entire memory as a disk cache hierarchy, the virtual memory and DSM system is kept small and simple. The only complication is the introduction of pageIDs for handling copy-on-write aliasing. It is intended that the use of copy-on-write in *Angel* will analysed and, if found to be of little worth, the pageIDs will be removed and a copy-on-reference scheme implemented instead, which provides a one-to-one mapping between addresses and data.

4.4 The Code

The current microkernel handles virtual memory, distributed shared memory, networking, disk management, virtual processor management and upcalls. In total it consists of only 3,500 lines of C++ code (2,500 lines of source and 1,000 lines of headers)[6]. This has taken approximately nine man-months of work and is well structured to allow easy porting to other systems (88K and Alpha systems are already under consideration). The SASA nature of the system has greatly simplified the internal structure of the kernel since address are no longer process dependent and caching schemes are simplified.

However, the microkernel is of limited use without services. Work is therefore underway to produce a set of libraries, including POSIX.1 and POSIX.4a threads, for use in developing services. Initial services will include an *Object Manager* and a *Namespace manager*[7].

5 Conclusions

Angel, as outlined in this paper, is the basis of SARC's next generation operating system research, building on the lessons of *Meshix*. Since it is a "research vehicle" rather than a commercial product we had far more flexibility in the design. This allowed us to drop fundamental POSIX compliance. The result is an operating system architecture which is microkernel based, parallel, replicated, decentralised, and based around the concept of resource unification which we believe will improve efficiency and simplify the kernel design and implementation.

A Unix-hosted implementation is now complete, and work is underway for a native implementation. The speed of development bares out the belief that the single address space architecture results in a simpler kernel. The reduction of the number of complex areas in the kernel should also improve its reliability and maintainability.

The ease with which numerous systems can be built above the single address space, and the efficiency available in these schemes, shows the performance benefits of this operating system architecture. Examples are the ease with which LRPC can be implemented, that load balancing of threads is completely transparent, and the simple efficient fault tolerance scheme designed for this architecture. In addition,

6. This can be expected to increase when the simulated devices are replaced in a native implementation.
7. Probably in the form of a POSIX filesystem in order ease the development of a standalone system.

is should still be possible to support a POSIX compliant interface above *Angel* without compromising its design.

We are now looking at novel approaches to provide the service layers above the Angel kernel. The scheme currently being worked on is based around a question space implemented using the Linda tuplespace [36].

Bibliography

[1] P. Winterbottom and P. Osmon, "Topsy: An Extensible Unix Multicomputer," in *UK IT90 Conference, Southampton University*, 1990.

[2] P. Osmon, T. Stiemerling, A. Valsamidis, A. Whitcroft, Wilkinson.T., and N. Williams, "The Topsy project: a position paper," in *Parle '92*, June 1992.

[3] N. Accetta, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "MACH: A new kernel foundation for UNIX development," in *USENIX Summer Conference*, July 1986.

[4] M. Rozier, "Overview of the CHORUS Distributed Operating Systems," Tech. Rep. CS-TR-90-25, Chorus Systemes, 1990.

[5] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren, "Amoeba: A distributed operating system for the 1990's," *IEEE Computer*, pp. 44–53, June 1990.

[6] D. Kuhn, "IEEE's Posix: making progress," *IEEE Spectrum. (USA)*, vol. 28, pp. 36–39, December 1991.

[7] Dobberpuhl and others, "A 200Mhz 64-bit Dual Issue CMOS Microprocessor," in *International Solid-State Circuits Conference*, February 1992.

[8] MIPS Computer Systems Ltd., *RISC Microprocessors, V_r4000 - User's Manual.* NEC, 1991.

[9] U. Ramachandran, G. Shah, S. Ravikumar, and J. Muthukumarasamy, "Scalability study of the KSR-1," Tech. Rep. GIT-CC93/03, College of Computing, Georgia Institute of Computing, Atlanta, Georgia, 1993.

[10] "Fibre Channel Rev 0.93." Working Draft ANSI, December 1991.

[11] T. Wilkinson, T. Stiemerling, P. Osmon, A. Saulsbury, and P. Kelly, "Angel: A Proposed Multiprocessor Operating System Kernel (Extended Abstract)," in *European Workshop on Parallel Computing*, March 1992.

[12] K. Murray, *Wisdom: The Foundation of a Scalable Parallel Operating System.* PhD thesis, University of York, Department of Computer Science, 1990.

[13] Perihelion Software Ltd., *The Helios Operating System.* Prentice Hall, 1989.

[14] A. Barak and R. Wheeler, "MOSIX: An integrated Multiprocessor UNIX," in *Proc. of the Winter 1989 USENIX Conference*, pp. 101–112, February 1989.

[15] B. Bershad, T. Anderson, E. Lazowska, and H. Levy, "Lightweight remote procedure call," *ACM Operating Systems Review*, vol. 23, pp. 102–113, December 1989.

[16] A. Bricker, "A new look at micro-kernel-based UNIX operating systems: Lessons in performance and compatability," in *EurOpen Spring'91 Conference, Tromsoe, Norway*, May 1991.

[17] J. C. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance," in *ASPLOS, International Conf. on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, CA (USA)), pp. 75–85, April 1991.

[18] S. Zatti, "A Multivariable Information Scheme to Balance The Load in a Distributed System," Tech. Rep. UCB/CSD 85/234, Computer Science Division, EECS UIniversity of California, Berkeley, 1985.

[19] G. Bell, "Ultracomputers: A Teraflop Before Its Time," *Communications of the ACM*, vol. 35, pp. 27–47, August 1992.

[20] K. Li, *Shared Virtual Memory on Loosely Coupled Multiprocessors*. PhD thesis, Yale University, Department of Computer Science, 1986.

[21] P. Osmon, T. Stiemerling, A. Whitcroft, Wilkinson.T., and N. Williams, "Evaluating Meshix – a Unix compatible micro-kernel Operating System," in *OpenForum'92*, November 1992.

[22] A. Whitcroft and P. Osmon, "The CBIC: Architectural Support for Message Passing or Shared Memory?," in *U.K. Performance Engineering Workshop*, September 1992.

[23] E. Hagersten, A. Landin, and S. Haridi, "DDM – A Cache-only Memory Architecture," Tech. Rep. Research Report R91:19, SICS, Sweden, November 1991.

[24] M. Hill, J. Larus, S. Reinhardt, and D. Wood, "Cooperative shared memory: software and hardware for scalable multiprocessors," in *ASPLOS V*, pp. 262–273, September 1992.

[25] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam, "The Stanford DASH multiprocessor," *IEEE Computer*, vol. 25, March 1992.

[26] "SCI: Scalable Coherent Interface." IEEE Standards document P1596/D1.7, August 1991.

[27] T. Wilkinson, "Implementing Fault Tolerance in a 64-bit Distributed Operating System," Tech. Rep., City University, 1993.

[28] T. Wilkinson *et al.*, "Compiling for a 64-Bit Single Address Space Architecture," Tech. Rep. TCU/SARC/1993/1, SARC, City University Computer Science Department, March 1993.

[29] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9 from Bell Labs," in *Summer UKUUG Conference, London*, pp. 1–9, July 1989.

[30] R. Pike, D. Presotto, K. Thompson, and H. Trickey, "Plan 9, A Distributed System," in *Spring EurOpen Conference, Troms.*, pp. 43–40, May 1991.

[31] M. Scott, T. LeBlanc, B. Marsh, T. Becker, C. Dubnicki, E. Markatos, and N. Smithline, "Implementation Issues for the Psyche Operating System," Tech. Rep., University or Rochester, Department of Computer Science, 1988.

[32] M. Scott, T. LeBlanc, and B. Marsh, "A Multi-User, Multi-Language, Open Operating System," Tech. Rep., University or Rochester, Department of Computer Science, April 1989.

[33] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska, "How to Use a 64-Bit Virtual Address Space," Tech. Rep. 92-03-02, Department of Computer Science and Engineering, University of Washington, March 1992.

[34] J. Chase, H. Levy, M. Baker-Harvey, and E. Lazowska, "Lightweight Shared Objects in a 64-Bit Operating System," Tech. Rep. 92-03-09, Department of Computer Science and Engineering, University of Washington, March 1992.

[35] G. Heiser, K. Elphinstone, S. Russell, and G. Hellestrand, "A Distributed Single Address-Space Operating System Supporting Persistence," Tech. Rep. 9302, School of Computer Science and Engineering, The University of New South Wales, March 1993.

[36] N. Williams, "Linda Tuplespaces in Angel: A marriage made in Heaven?," Tech. Rep. TCU/SARC/1993/2, Systems Architecture Research Centre, City University, February 1993.