# Efficient OS Resource Management for Distributed Embedded Real-Time Systems

Neil Audsley Rui Gao Ameet Patil Paul Usher
Real-Time Systems Group,
Department of Computer Science, University of York, York YO10 5DD, UK
Email:{neil,rgao,appatil,usher}@cs.york.ac.uk

## Abstract

*Distributed embedded real-time systems provide distinct challenges for Operating Systems (OS). These systems have limited resources and inherent parallelism via multiple processors. The use of static off-the-shelf OSs for these platforms is inappropriate, without significant modification, as system efficiency is of primary concern. Within the paper we present two areas of work that combine to aid overall system efficiency within an off-the-shelf OS (Linux). Firstly, application specific resource management is provided so that OS policies can be adapted by applications dynamically to aid efficient scheduling and resource management. Secondly, efficient mechanisms to enable distributed resource management are presented. The latter provide access to remote resources using the same mechanisms as local resources.*

## 1 Introduction

Efficient implementations of distributed embedded real-time systems necessitate fundamentally different approaches to OS policy and mechanism for resource management. Within this paper, we consider both local and remote resource management in systems with limited resources, where efficient management of resources is required. We observe that standard resource management approaches do not take advantage of application knowledge of future resource requirements. Such knowledge can be used at run-time to help resource management policies to better allocate resources to applications. Also, we observe that distributed resource access is usually provided by a combination of heavyweight network protocols and remote OS servers. Within a resource limited system, such approaches are resource inefficient. Key challenges include the dynamic specialisation of OS resource management to the needs of a particular application; the provision of efficient access to remote resources. This paper describes solutions to these problems:

- *Application-Specific Resource Management Policies*: provision of reflection within the operating system to enable run-time application specific resource management policies (for scheduling, memory, power etc.);
- *Efficient Remote Resource Access*: recognition and provision of distributed resource access at low-level in the operating system to enable low resource systems to efficiently utilise remote resources;

The remainder of this paper describes these solutions within the context of Linux (including virtual memory). Firstly, application specific resource management is described that allows applications to affect both scheduling and virtual memory allocation policies of the OS. This enables more effective run-time scheduling, smarter memory management, a reduction in page-faults and reduced overall system power consumption. Secondly, efficient remote resource access within Linux is proposed. This extends the file-system interface to remote devices (not commonly supported under UNIX / POSIX implementations), and uniquely, allows access to those remote devices without significant overhead at the remote node.

The use of a standard OS (ie. Linux) as the basis of the reported work is based on the observations that standard OS APIs are often a requirement within industrial embedded real-time system development. However, the work described is also applicable to other OSs.

The remainder of this section provides background. Section 2 describes application specific resource management policies, with section 3 describing efficient remote resource access. Finally, section 4 provides conclusions.

### 1.1 Background

The Linux kernel (2.4 series) provides all of the facilities expected of a modern networked operating system and

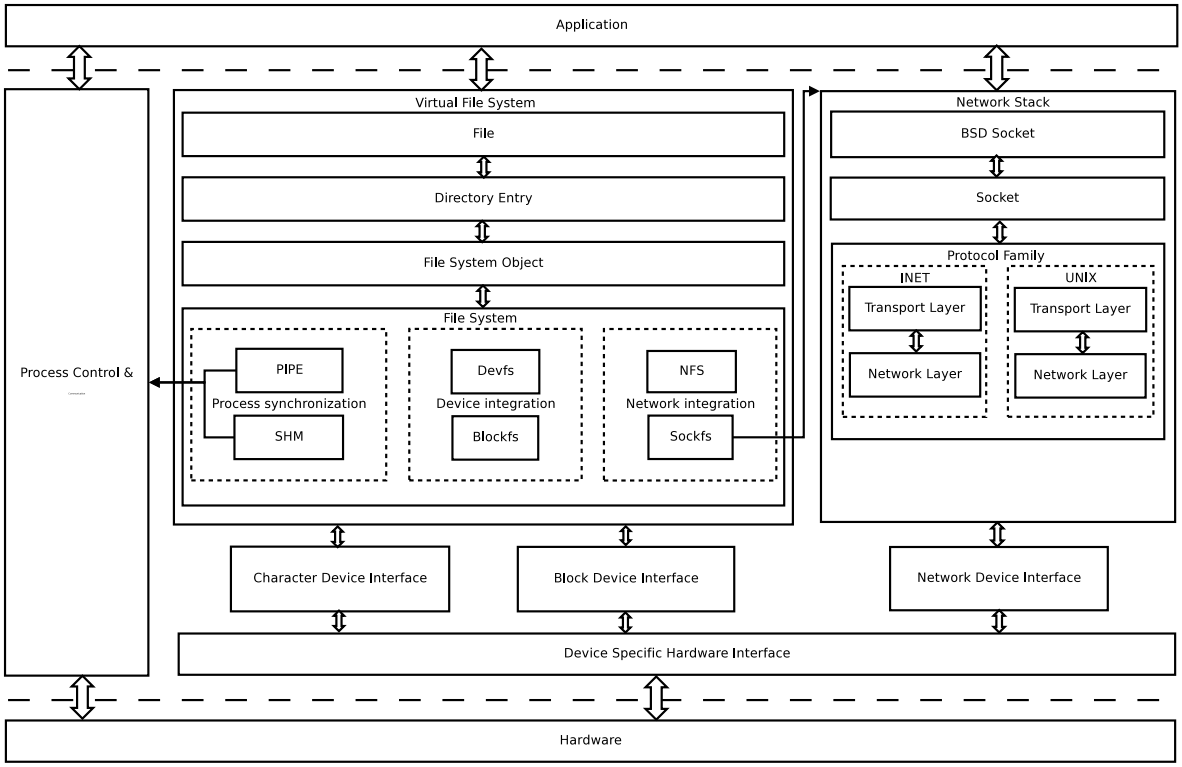


**Figure 1. Architectural overview of 2.4 series Linux kernels**

as such it does not differ radically from the norm other than in its adherence to being open-source. One advantage of this is the wealth of excellent information that is available on its design. Most text books only provide an overview of an operating systems design, but the open-source nature of Linux means that it is possible to obtain very detailed documentation of its operation [5, 20]. From such sources it is possible to ascertain that Linux is primarily a monolithic kernel that has been augmented with support for dynamically loadable modules, the basic structure is illustrated in Figure 1. This clearly illustrates the orthogonal nature of a design which concentrates separately on the three primary concerns of a modern networked OS, in order to simplify implementation and improve flexibility.

**Process control:** Mechanisms to create, destroy and communicate with processes.

**Device access:** Mechanisms to control access to devices.

**Network communication:** Mechanisms allowing communication across the network.

Like many UNIX based operating systems Linux uses the virtual file system (VFS) to provide some degree of integration with the otherwise isolated parts of the kernel. This is achieved through the use of special file systems that map VFS operations onto suitable network or process related functionality, some examples of this have been shown in Figure 1 (pipe, nfs, shm and sockfs). Unfortunately this method of integration is only partially successful as network devices cannot be accessed directly via the VFS.

## 1.2 Network Stack Functionality

The modern network stack is a complex piece of software that allows applications to communicate without worrying about whether they are located on the same machine. Unfortunately, there is little or no integration with the VFS and consequently it is possible to use a local device or to communicate with a process on a remote machine, but not to access a device on a remote machine.

Although there are a wide variety of different network stacks undoubtedly the most popular is IPv4. From the information available it has been possible to produce a graph showing the sequence of function calls made by this functionality in order to implement the TCP and UDP protocols, see Figure 2 [5, 20] . Whilst this diagram identifies a considerable number of different function calls it should by pointed out that no attempt has been made to illustrate supporting protocols such as ARP. This diagram also only illustrates the high level functionality involved in the operation of the TCP stack, in addition to omitting any support

functionality that may be called before handing off work to the lower level

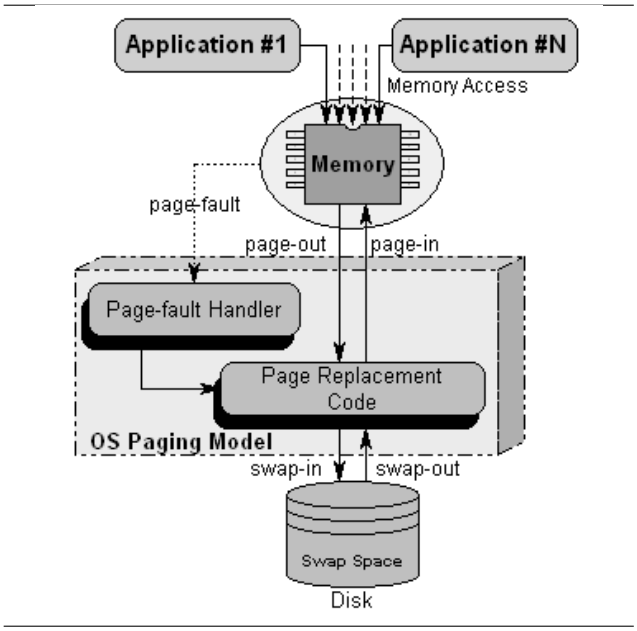


**Figure 3. OS Paging Model**

### 1.3 Memory Management

In a demand paged system the physical memory pages are allocated to application processes only when requested for the first time. Figure 3 is a diagrammatic representation of the paging model. Application processes make requests in the form of access to virtual memory addresses. The hardware traps *page-faults* to the OS *page-fault handler* routine. This *page-fault handler* routine analyses the information provided by hardware and transfers control to the *page replacement code* if needed. The *page replacement code* is responsible for all the paging activity in the system such as reclaiming unused pages from memory, bringing back the evicted pages from the *swap-space*, etc.. Depending on the type of a *page-fault* the *page-fault handler* does the following to handle it:

- a page from memory is moved to *swap-space*. This is called *page-out* or *swap-in* operation,
- a page from *swap-space* is moved back into memory. This is called *page-in* or *swap-out* operation,
- in case of demand paging, a new page is allocated to the first-time requesting process.

LRU (Least-Recently Used) and CLOCK based policies are the most widely accepted and commercially used in OSs like Linux [4][13], Mach [7]. However, due to recency based paging decisions, LRU fails to keep pages in memory that are frequently accessed over a long-term period. Improvements to LRU that were proposed include: LRFU [8][6], EELRU [26], LRU-K [9][10], 2Q [25], and more [2]. The CLOCK replacement policy is easier to implement than LRU and requires less bookkeeping. It has been shown that performance of CLOCK approximates that of LRU [23].

CLOCK-PRO [21] is an improved version of CLOCK combining the advantages of CLOCK and the LIRS [22] policy; the latter being proposed for better buffer cache performance. CLOCK-PRO maintains a circular list of pages with three clock hands. The $HAND_{hot}$ points to the hot page (page which is new allocated or recently accessed) with the largest recency. Any hot pages swept by this hand turn into cold pages (not recently accessed). The $HAND_{cold}$ points to the last resident cold page (i.e. the furthest one to the list head). $HAND_{test}$ points to the last cold page which is in its test period. This hand is used to terminate the test period of cold pages. The non-resident cold pages swept by this hand will leave the circular list for reclamation. Several other page replacement policies have been proposed along with the above mentioned policies. For example: the Working Set (WS) model [17][18], SC [3], etc.

The Linux 2.6.16 kernel implements LRU based page replacement policy which can be closely compared to LRU-2Q [13]. Memory in Linux is divided into three zones – *ZONE_DMA*, *ZONE_NORMAL* and *ZONE_HIGHMEM*. Pages in each zone are stored in two zone-wise lists – *active_list* and *inactive_list*. The *active_list* consists of most recently accessed pages or all newly allocated pages. Unlike in theory, Linux does not reclaim pages only on a *page-fault*. A special kernel process *'kswapd()'* that runs at fixed intervals is responsible to reclaim pages. *kswapd()* tries to maintain a fixed number of free pages that are available in a zone determined by the value of *zone water-mark*. It is this process that moves pages present in *active_list* that are not recently accessed into the *inactive_list*. While in *inactive_list* pages are still marked as accessed by the kernel so that the kswapd() moves them back into *active_list*. When ratio of the number of pages in the *inactive_list* and *active_list* reaches certain mark, kswapd() starts reclaiming unreferenced pages from the *inactive_list*. It is shown in practice that the performance of this replacement policy is close to LRU [23]. For simplicity, in all further discussions we refer to Linux's replacement policy as *Linux-LRU*. Note that *Linux-LRU* makes page replacement decisions solely on the basis of the recency and not using any frequency features.

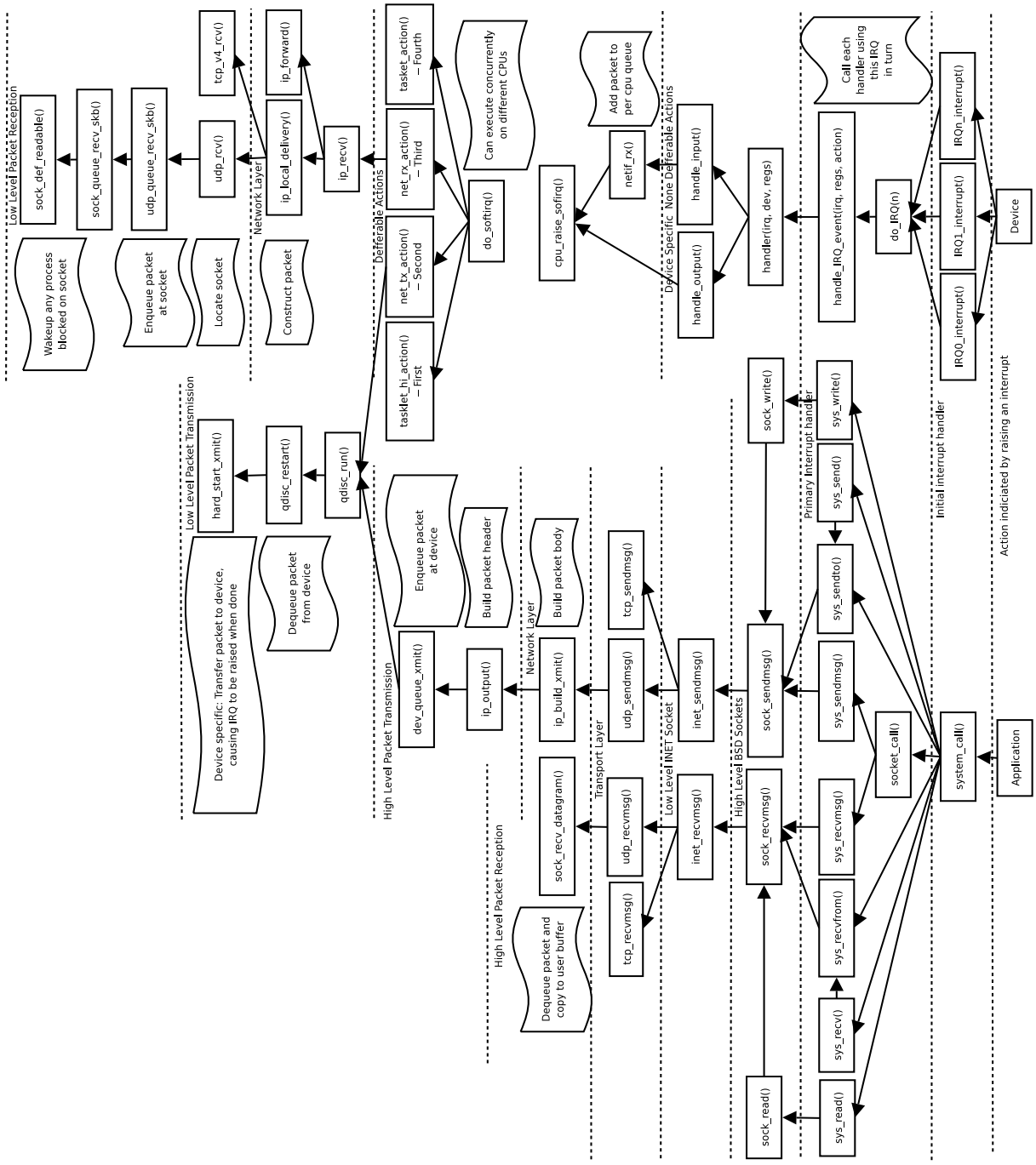


**Figure 2. Function call hierarchy for the UDP and TCP protocols**

## 2 Application-Specific Resource Management Policies

Operating systems and applications in general are an enormous source of information that could be used for each others benefit. On one hand, an operating system knows when an application process will be scheduled?, what resources are available in the system?, etc. and on the other hand an application knows exactly when it needs to use a resource?, how long it should actually execute?, etc. If we bridge the gap between the OS and the application, then it is possible for the OS to dynamically adapt to the application currently executing thereby providing better application-specific support. To facilitate the exchange of information between OS and application we chose to use the reflection mechanism.

### 2.1 Reflection within Operating Systems

A general purpose RTOS is built for the general case and without the knowledge of the applications that would execute upon it. Such systems may therefore contain some functionality which real-time embedded applications may not require (Eg. networking, graphics). To address this issue many recent systems utilize a component based architecture so that no unnecessary components are included. Although component dependencies mean that some additional support functionality may be required. Developing components that work together therefore adds several restrictions to their development methodology and thereby compromises the overall systems flexibility.

Another approach to overcoming this problem is to provide APIs that applications can use to change certain policies in the RTOS to their specific requirements. For example: in MaRTE[19] OS, the applications use the API to change the scheduling policy being used to schedule the application threads. On the other hand, SHaRK[11] provides with a similar API to develop applications that use their own scheduling policies. Evaluation results of these approaches show a considerable amount of overhead added to the system there by making the approach infeasible[19].

Alternatively, we consider reflection within OSs. Reflection is usually seen in programming languages (eg. Ada, Java, etc.) to provide the flexibility for the applications to change their behaviour dynamically. Also, OSs like ApertOS [27], Spring [24] are reflective OSs [12]. Reflection essentially is a mechanism by which a program becomes 'self-aware', checks its progress and can change itself or its behaviour [16]. This change can occur by changing data structures, the program code itself, or sometimes even the semantics of the language its written in. To facilitate this, the program has to have knowledge about the data structures, language semantics, etc. The process by which this information is provided is called *'Reification'*.

In terms of OSs, reflection is used to allow applications to access key OS data structures to obtain information pertaining to the current system performance and resource management policies (eg. scheduling). An application is then able to modify or introduce new policies into the RTOS with the help of reflective system modules that intercept certain events or function calls to change the overall behaviour of the application and the system. The Reflective OS on the other hand is able to obtain critical information from the applications and change its structure/behaviour dynamically to adapt to the application.

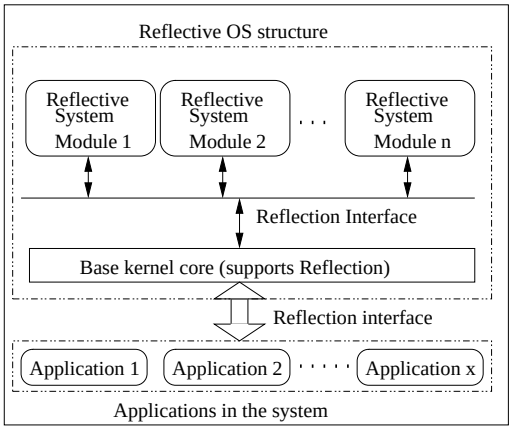


**Figure 4. Generic Reflective OS framework**

A reflective OS framework consists of a Base kernel core that provides support for reflection in the form of reflection interface for system modules/applications to reify information, introspect and intercept the base-level. The System modules (eg. scheduler) are designed to be completely reflective. A reflective system module (eg. a reflective scheduler) makes use of the interface provided to analyse reified information and take intuitive steps to intercept and change behaviour of the base-level module. Fig. 4 shows several reflective system modules as well as the applications using the in-kernel reflection interface. Similar to the reflective system modules, the applications can also be reflective. The meta-level code in the reflective applications (not shown in fig. 4) can analyse the reified information from the system and change the behaviour of the application.

The reflective system modules (see fig.5) implement a generic policy at the base-level. For example: in case of a reflective scheduler, a simple round-robin scheduling policy or an optimised scheduling policy can be implemented at the base level. Depending on the application requirement, the meta-level code can then change this base-level policy to an application specific one either at run time or statically.

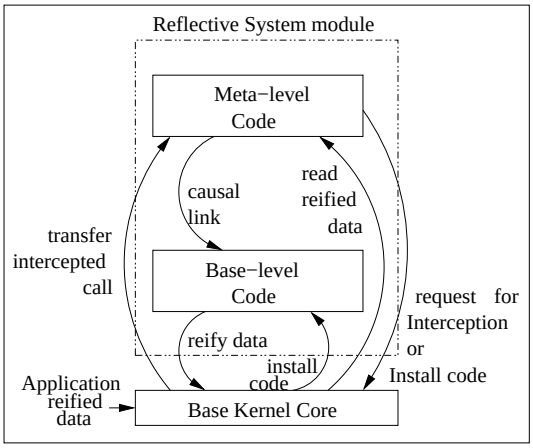


**Figure 5. Reflective framework**

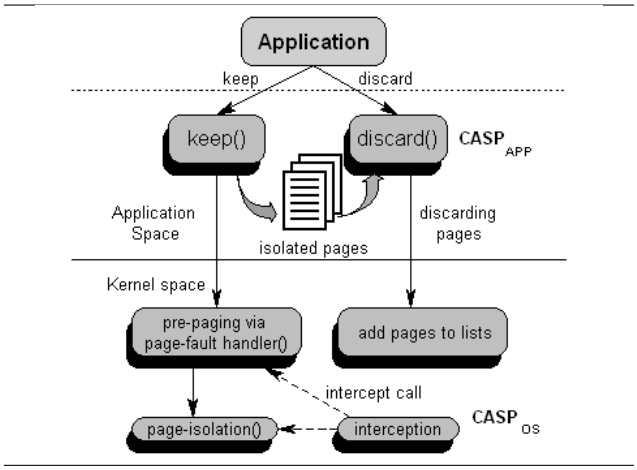


**Figure 6. CASP Design**

## 2.2 Reflection in Linux: Application-Level Scheduling

Fig. 5 shows how the applications make use of reflection interface in a system module such as scheduler. We incorporate a generic module-based reflective framework [1, 15] in Linux kernel 2.6.16. Our initial implementation involves using reflection to improve the performance of the virtual memory subsystem in Linux and also to reduce memory power consumption.

In our previous work we developed DAMROS [15], a reflective real-time operating system that implements the reflective framework and provides support for application-specific scheduling and memory management. DAMROS supports hierarchical scheduling allowing multiple threads of the same application process to be scheduled in an application-specific way. DAMROS provides reflection primitives such as '*installCode()*', '*interceptCode()*', etc. which allow applications to either change the scheduling policy or install a new scheduler in the system. The framework assures that a change brought in by one application can not affect another.

## 2.3 Reflection in Linux: Application-Level Memory Management

With no knowledge of the application's memory access patterns, the page replacement policy used in Linux shows poor paging performance in its inability to reduce the occurrence of *page-faults*. This results in unnecessary page-swap operations and increases energy consumption due to paging. By allowing applications to cooperatively hint to the OS about their access patterns well in advance, more accurate page replacement decisions can be taken at runtime thereby reducing the occurrence of any further *page-faults*.

We propose a Co-operative Application Specific Paging (CASP) mechanism that can be used upon any existing page replacement policy in the OS in order to reduce paging energy consumption. Applications hint about their memory access patterns to CASP which then makes available the application's working set. The basic CASP design is shown in Figure 6. Although CASP can be used by other application types, it is designed mainly to support *SEQ-type* (where pages tend to be accessed sequentially). It is expensive for the OS to determine an application's access pattern at runtime with no prior knowledge of the application. With co-operation from *SEQ-type* applications CASP is able to accurately determine the access pattern at runtime. CASP is divided into an application-level part called $CASP_{app}$ and an OS-level part called $CASP_{os}$ (see fig.6). Applications use $CASP_{app}$ primitives – (1) *keep()* and (2) *discard()* in their code to hint about their memory access pattern. This information is then passed onto the $CASP_{os}$ part which makes certain pages present in memory before the process actually accesses them. $CASP_{os}$ uses *pre-paging* and *page-isolation* via *interception*.

Figure 7 shows CASP in operation with an application having a separate page list containing all the *isolated* pages as a result of a call to *keep()*. CASP operates non-intrusively with the existing page replacement code and thus has no side-effects. Since the *isolated* pages do not exist in the OS page lists, they are never considered as candidates for reclamation by the OS's *page replacement code*. In fact this could speed up the reclamation process since the code now has less reclaim candidates. CASP achieves page locking without the knowledge of the original *page replacement code* making it a generic approach that can easily operate on top of any existing replacement policy. Furthermore, *keep()* and *discard()* can be automatically inserted into the appli-

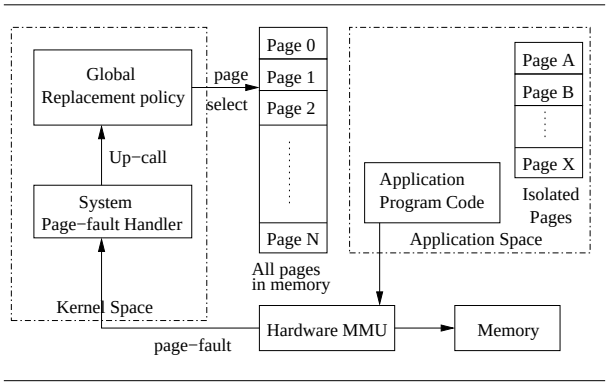


**Figure 7. CASP Paging Model**

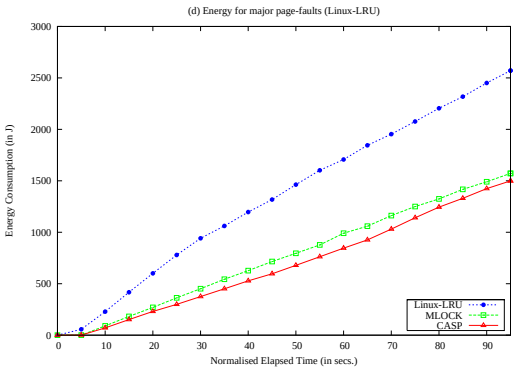


**Figure 8. Energy consumption in Linux-LRU**

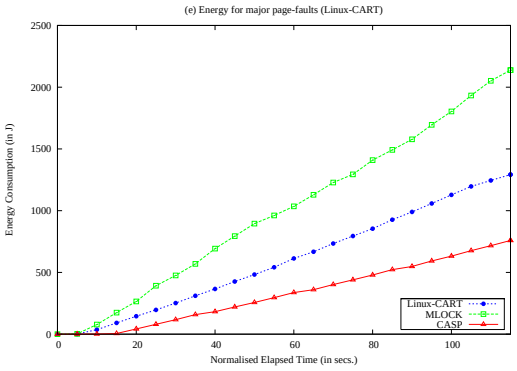


**Figure 9. Energy consumption in Linux-CART**

cation's code at compile time like the *LOCK* and *UNLOCK* methods in [14].

CASP helps applications to have their working set pages always resident in memory. The generic nature of CASP allows it to be implemented on top of any existing page replacement policy. We implemented CASP in three variants of Linux 2.6.16 kernel each implementing LRU-2Q based [13][4], CART[23] and CLOCK-PRO[21] paging policies respectively. We call these three variants; *Linux-LRU*, *Linux-CART* and *Linux-PRO* respectively. For initial testing we executed a linear scan application on top of each variant with and without using CASP and estimated the total memory energy consumption. The application, linearly scans through a 100MB file 5 times on a 300MHz Cyrix Media GX processor with 64MB RAM and a 4GB 7000 RPM hard disk. We produced three versions of scan for our experiments – (1) *scan*: the original version with no change, (2) *scan-MLOCK*: scan uses linux primitives *mlock()* and *munlock()* to lock and unlock memory location dynamically such that while scanning the pages remain in the RAM, and (3) *scan-CASP*: here we use the CASP primitives – *keep()* and *discard()* instead of the *mlock()* and *munlock()* as in the previous case.

Figures 8,9 and 10 represent the memory energy consumption of *scan*, *scan-MLOCK* and *scan-CASP* on each Linux variant. These clearly indicate that memory consumption is lowest when using CASP. On an average *scan-CASP* uses approximately 35% less energy than *scan* and 32% less energy than *scan-MLOCK* across all Linux variants. Furthermore it has been observed that reflection framework does not impose any serious time and space penalties to the original system. Our future work in this area will look at incorporating this framework with the Linux scheduler and also to do better power estimations.

## 3 Efficient Remote Resource Access

Each machine in a network of embedded systems has considerably less resources than those seen in many traditional computer networks. As a result the applications running on them are far more likely to need to access remote files and devices in order to successfully complete their task.

Networked operating systems such as Linux provide this kind of access via file systems such as NFS which essentially convert the file operations of the virtual file system (VFS) into messages sent to the remote machine via the network stack. These messages are then delivered to a proxy process which performs the access on behalf of the client before sending a reply message back to the client via the network stack, see Figure 11.

Given the increased use of this mechanism in the target system it is important to maximise both performance and the range of resources that can be accessed. Unfortunately networked operating systems where never designed with this kind of access in mind and as a result they are less than ideally suited to the task.

Networked operating systems such as Linux treat remote

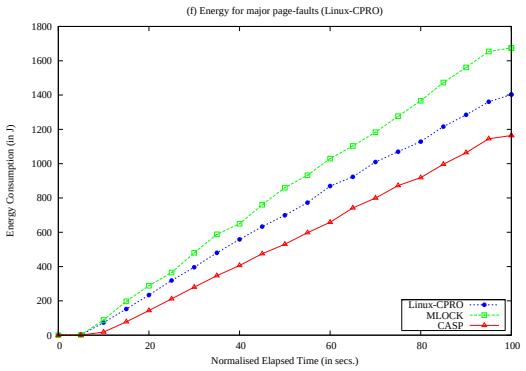


**Figure 10. Energy consumption in Linux-PRO**

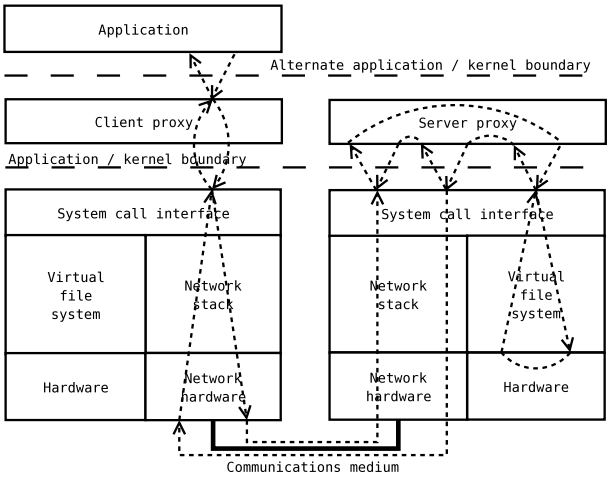


**Figure 11. Control flow when accessing remote resources**

communication and file access as separate issues, when they are clearly inter-dependant in the target system. This causes two separate stacks of software to be built, limiting the flow of information around the system. This adversely affects both performance and flexibility by ensuring that the semantics of the device being accessed and the communications medium being used cannot affect the higher level functionality.

For example the layering within the network stack insures that the higher level functionality cannot adapt if the communications medium is completely reliable. As a result socket based mechanisms exhibit poor throughput and latency in comparison to local IPC mechanisms. Similarly the VFS can access both local and remote files via a suitable file system but it cannot access a remote device. Such issues clearly demonstrate the unsuitability of networked operating systems such as Linux in the target environment.

It therefore seems clear that in order to improve performance without re-writing the OS we must reduce the volume of functionality involved in sending a request to a server, in addition to finding some way of allowing the VFS to access remote devices.

## 3.1 An Integrated Approach

To do this we have implemented a simple file system that does not differentiate between files or devices, consequently all remote resources are represented locally as ordinary files. A very simple network stack then packs the parameters used by VFS operations up into a message that is embedded directly into a single Ethernet frame. This effectively eliminates any overheads occurring because of the network stack as well as insuring message integrity via the CRC checksum attached to every frame. Each message also contains a sequence number in order to protect against dropped frames.

Since most VFS operations do not pass or return much data it is generally the case that both request and reply messages can fit into a minimum sized Ethernet frame, which is clearly beneficial to performance. Worst case performance is exhibited by the read() and write() operations but even here the request or the reply message always fits into the smallest frame, whilst 1494 bytes of data can be transferred using the largest frame.

In contrast IPv4 limits the data transferred by a single operation to 64K. This is unnecessarily large for the vast majority of device accesses, especially as file systems such as NFS default to reading and writing 1024 bytes at a time. Given that a single Ethernet frame can contain nearly 50% more data it is seems unnecessary to complicate the protocol by supporting multi-frame messages when the vast majority of devices do not require it. This limit can be increased still further without complicating the protocol provided that the Ethernet devices support jumbo frames. If this is the case then read() and write() could transfer up to 8K of data in a single message.

Whilst this mechanism still works within the confines of a traditional networked OS, reducing the volume of functionality involved in any access to a remote resource is clearly beneficial. This reduction in functionality has been illustrated graphically in Figure 12.

## 3.2 Performance

This approach has been tested on a pair of 266MHz Intel Pentium II based computers that are directly connected via a 100Mbit Ethernet link, see Table 1.

The figures shown in this table represent the average performance seen by the test application when repeatedly accessing a wide variety of remote devices and disc based

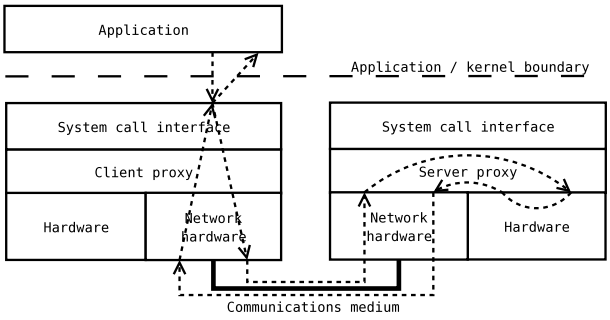


**Figure 12. Improved Control flow for accessing remote resources**

| | Latency ($\mu$s) |
|---|---|
| close() | 91 |
| open() | 107 |
| read() | 278 |
| write() | 269 |

**Table 1. Average performance for remote file operations**

files. In contrast a test application that transmits similar sized Ethernet frames via UDP without accessing any files or devices takes 147$\mu$s in the close()/open() simulation and 358$\mu$s in the case of read()/write(). Our mechanism therefore achieves at least 22-38% better performance depending upon the type of operation being performed.

## 4 Conclusions

This paper has presented two areas of work:

- Efficient application specific resource management.
- Efficient remote resource access.

The former clearly demonstrated how an OS can be made to adapt at run-time to the needs of those applications currently in use in order to improve performance and energy efficiency. Whilst the latter has shown that remote resource access can be achieved without the use of a traditional network stack.

Experimental results have also show that these techniques can provide substantial performance improvements with memory consumption being reduced by 35% and network latency reduced by 20-40%.

Work is continuing in these areas in order to improve on these results, thereby improving performance, efficiency and flexibility across a wider range of applications.

## References

[1] Ameet Patil and Neil Audsley. An Application Adaptive Generic Module-based Reflective Framework for Real-time Operating Systems. In *Proceedings of the 25th IEEE Work in Progress session of Real-time Systems Symposium*, Lisbon, Portugal, December 2004.

[2] Amos Fiat and Ziv Rosen. Experimental studies of access graph based heuristics: beating the LRU standard? In *SODA '97: Proceedings of the eighth Annual ACM-SIAM Symposium On Discrete Algorithms*, pages 63–72, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[3] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, second edition, 1997.

[4] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, and D. Verworner. *Linux Kernel Internals*. Addison–Wesley, second edition, 1998.

[5] D. P. Bovet and M. Cesati. *Understanding the LINUX KERNEL*. O'Reilly & Associates, Inc., second edition, 2002.

[6] D. Lee and J. Choi and J. H. Kim and S. H. Noh and S. L. Min and Y. Cho and C. S. Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Trans. Computers*, 50(12):1352–1361, 2001.

[7] D. McNamee and K. Armstrong. Extending the Mach External Pager Interface To Accommodate User-Level Page Replacement Policies. Technical Report TR-90-09-05, 1990.

[8] Donghee Lee and Jongmoo Choi and Jong-Hun Kim and Sam H. Noh and Sang Lyul Min and Yookun Cho and Chong Sang Kim. LRFU (Least Recently/Frequently Used) Replacement Policy: A Spectrum of Block Replacement Policies. Technical Report SNU-CE-AN-96-004, Seoul National University, March 1996.

[9] Elizabeth J. O'Neil and Patrick E. O'Neil and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. pages 297–306, 1993.

[10] Elizabeth J. O'Neil and Patrick E. O'Neil and Gerhard Weikum. An optimality proof of the LRU-K page replacement algorithm. *Journal of ACM*, 46(1):92–112, 1999.

[11] P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, June 2001.

[12] J. A. Stankovic and K. Ramamritham. *A Reflective Architecture for Real-Time Operating Systems*. Prentice-Hall, Inc., 1995.

[13] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, April 2004.

[14] Mohammad Malkawi and Janek Patel. Compiler Directed Memory Management Policy for Numerical Programs. In *SOSP '85: Proceedings of the tenth ACM Symposium on Operating Systems Principles*, pages 97–106, New York, NY, USA, 1985. ACM Press.

[15] A. Patil and N. Audsley. Implementing Application-Specific RTOS Policies using Reflection. In *Proceedings of the 11th IEEE Real-time and Embedded Technology and Applications Symposium*, pages 438–447, San Francisco, CA, USA, 2005.

[16] Patrick Rogers. *Software Fault Tolerance, Reflection and the Ada Programming Language*. PhD thesis, University of York, UK, October 2003.

[17] Peter J. Denning. The Working Set Model for Program Behavior. In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 15.1–15.12, New York, USA, 1967. ACM Press.

[18] Peter J. Denning and Stuart C. Schwartz. Properties of the Working-Set Model. *Communications of ACM*, 15(3):191–198, 1972.

[19] M. A. Rivas and M. G. Harbour. POSIX-Compatible Application-Defined Scheduling in MaRTE OS. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 67–75. IEEE Computer Society, June 2002.

[20] A. Rubini and J. Corbet. *LINUX DEVICE DRIVERS*. O'Reilly & Associates, Inc., second edition, 2001.

[21] Song Jiang and Feng Chen and Xiaodong Zhang. CLOCK-Pro: an Effective Improvement of the CLOCK Replacement. In *Proceedings of 2005 USENIX Annual Technical Conference (USENIX'05)*, Berkeley, CA, USA, April 2005. USENIX Association.

[22] Song Jiang and Xiaodong Zhang. LIRS: an Efficient Low Inter-reference Recency Set Replacement Policy to improve Buffer Cache Performance. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 31–42, New York, NY, USA, 2002. ACM Press.

[23] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association.

[24] J. A. Stankovic and K. Ramamritham. The Spring Kernel: a New Paradigm for Real-Time Operating Systems. *SIGOPS Oper. Syst. Rev.*, 23(3):54–71, 1989.

[25] Theodore Johnson and Dennis Shasha. 2Q: a low overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, 1994.

[26] Yannis Smaragdakis and Scott Kaplan and Paul Wilson. EELRU: Simple and Effective Adaptive Page Replacement. In *SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 122–133, New York, NY, USA, 1999. ACM Press.

[27] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 414–434. ACM Press, 1992.