The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

Using meta-objects to support optimisation in the Apertos operating system

Jun-ichiro Itoh
Keio University, Kanagawa, Japan

Rodger Lea and Yasuhiko Yokote
Sony CSL, Tokyo, Japan

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Using meta-objects to support optimisation in the Apertos operati system

Jun-ichiro Itoh. *Keio University, Kanagawa, Japan.
Rodger Lea. and Yasuhiko Yokote. *Sony CSL, Tokyo, Japan.*

Abstract

The Apertos OS has explored the use of the meta-object model and reflection as a means to build highly flexible operating systems. While the benefits of such a system are great, the performance cost of a clean consistent use of the meta object model is high. Our initial work accepted this because of our desire to explore the model to its fullest. Recently we have turned our attention to optimisation techniques. Our work is interesting because we have tried to use the flexibility of the model to support our optimisations. This is in contrast to many systems that restrict or even compromise their initial model for the sake of performance. This paper outlines the Apertos system and discusses the optimisations we have made.

Introduction

Recent trends in networking, multi-media and mobile computing have placed new requirements on the flexibility and adaptability of existing operating systems. It is increasingly recognised that existing OS architectures are not sufficiently flexible to meet these demands. As such there is significant interest in building more flexible, or adaptive operating systems. However, this flexibility often has a performance price and OS designers are faced with a trade off between flexibility and performance. Inevitable, performance becomes the overriding factor, and the clean conceptual model originally proposed to achieve flexibility is compromised.

In this paper we describe our work with the Apertos operating system which is based on the meta-object model. We outline the design and overall implementation of the system and show how we have explored the meta-object model to its fullest. In particular we discuss how we have used the meta-object model consiste throughout the entire system. Our initial imp mentation suffered from this desire for a cl model by having poor performance. Recer we have been exploring how to optimise system. Our work differs from much other timisation exercises in that we are endeavo ing to exploit our model to support optim tion, rather than breaking the model. We that our initial experience is promising, that the techniques we are adopting will be b eficial to other groups who are exploring object oriented paradigm for the construct of operating systems.

This paper begins with an overview of different approaches to system flexibility then briefly describes the object/meta-ob model; we present an overview of our imp mentation of this model in the Apertos s tem; finally we discuss our current work to timise our system within the framework of object/meta-object model.

*itojun@mt.cs.keio.ac.jp

OS flexibility

We can categorise flexibility in work to date as follows: flexibility in the way we combine the components of the system at build time, flexibility in the way the systems adapts to application needs and flexibility in the way the system and its interfaces evolve over time.

Flexibility though modularity: Modularity is a key technique in building flexible software since it encapsulates components and makes them easier to change or evolve. Micro-kernel architectures have popularised such modularity and have demonstrated benefits in their overall flexibility[3][2]. Object oriented operating systems have extended this modularity with a finer degree of granularity and used techniques such as abstract interfaces and inheritance hierarchies to support a high degree of flexibility[10]. Systems such as Choices[6] have exploited this software engineering model to support an extremely sophisticated mix and match approach to the initial build phase, in some cases exploiting it to specialise systems not only for hardware but for the intended applications.

Flexibility through run-time adaptation: Most operating systems have supported some degree of run time adaptation; for example scheduling policies are often adapted to deal with changing application loads. Recently there has been much interest in the use of more sophisticated techniques to not only choose between existing policy modules, but also to dynamically generate code that is optimised to a set of run-time constraints [13] [4].

Flexibility of interface evolution: Flexibility to support the evolution of system interfaces is more difficult to achieve. Some of the work on object oriented systems has attempted to exploit type conformance [5] or similar means [7] to achieve it.

It is clear that the operating system community is aware of and is actively addressing the issues of flexibility. We argue however, that although this work is moving in the right direction, it is ad hoc in that is opens up pieces of a system without an overall architectural model for making that flexibility intrinsic and so available to all OS components.

Reflection and its use by Apertos

The Apertos system (formally Muse)[15][has been investigating a flexible open OS chitecture since 1988 and has concentrated efforts on the meta-level programming mo

Simply put, a meta-level architecture is in which all system components have a m level component that holds information ab that component and how it is used; in esse the meta-level defines control and policy. T meta-level component can be viewed as an vironment for the component and provides interface to manipulate aspects of the co ponent's environment. This is traditiona called the meta-interface or meta-object p tocol (MOP)[9]. However it could equally called its management interface.

For example, a thread in an object orien operating system is represented by a simple ject that has a 'standard' interface such as s and yield. In addition, somewhere in the k nel is the scheduling code which decides wh thread to execute next. This algorithm is ten referred to as the policy associated with simple object. In our terms this algorithm part of its meta-object. In a traditional OS implementation of the policy and its appli tion to the thread was a closed a-priori de sion made by the OS designers. In a more fl ible system we would like to be able to o up that algorithm, make it accessible to ot parts of the system and perhaps even cha it. By providing an interface to the schedu meta-object, a MOP, we are able to expose t algorithm and allow system designers and e applications to use it.

Reflection can be viewed as a use of basic foundation. Reflection is the process which we step back from computation tha on behalf of, or part of the application and ca out computation on behalf of the system its To achieve this it is necessary to be able access a description of the system. When have finished this computation we will h changed the description of the system. Th

changes are then 'reflected' back to the actual system. Hence reflection gives us a way to cleanly make changes to the system. Using the example above, when we use the MOP interface to make a scheduling decision, e.g. reorder the scheduler queue, we are making a reflective computation at the meta-level.

It is clear that this ability to access and manipulate the basic features of an OS is useful; some of the work cited above is using a variety of different techniques to do just that. What is different about the Apertos system is that the entire operating system is built using this meta-object architecture and so all aspects of the system are open and can be easily changed. It is our belief that only by doing this in a completely consistent manner will we be able to fully realise the benefits of the flexibility it provides.

The Apertos operating system

As mentioned above, Apertos is an investigation into how we can apply this meta-object model to all aspects of the operating system and what are the costs and drawbacks. The Apertos system generalises some of the ideas mentioned above.

Each simple object in Apertos is managed by a set of meta-objects. The meta-objects are associated with the simple object via a descriptor object that holds a reference to all the required meta-objects. This set is referred to as a meta-space which is also represented by an object (which we call a reflector). The meta-space constitutes the execution environment for an object. At any point in time there will be many meta-spaces co-existing in the system.

Interaction between objects is achieved through MetaCore, a minimal system component that allows an object to call into its meta-space. MetaCore can be likened to a micro kernel, however rather than implementing a simple set of base abstractions, it implements only the means to move from a simple object into its meta-space.

An object that wishes to access another object does so by calling into its meta-space; the

Figure 1: The object/meta-object relations
and the reflector hierarchy

target object is located and the invocation
patched. In the cases where an object wishe
compute not at the base level, but at the m
level, the same invocation model is used. H
ever the target for the invocation is a m
object and not a base object. This comm
invocation path implies that all aspects of
invocation are also open to change includ
look-up, dispatch and invocation semantics

The last extension that Apertos support
a means of allowing objects to change t
behaviour not by using the MOP to mo
some feature of their existing environment,
by migrating themselves to a completely
environment which supports a different se
policies. Apertos supports a simple form
type conformance to allow this.

Implementation

In the previous sections we outlined the c
ceptual framework that Apertos supports,
the object/meta-object model, and discus
how the Apertos system conforms to this c
ceptual framework. In this section we disc
in more detail how the actual implementat
of these concepts is carried out.

In fig.2 the abstract meta-architecture sho
in fig.1 is visible as the separate meta-spa
mCOOP, ***m***Zero and ***m***Core.

Figure 2: Implementation hierarchy of meta-spaces

MetaCore

MetaCore acts in a manner similar to that of a micro-kernel in more traditional systems. However, rather than providing a minimal set of resource abstractions, its role is restricted to supporting the object/meta-object model, the base notion of activity and interrupt dispatching. This allows a very small core object (currently around 3K) which aids both understanding and porting. MetaCore supports the following operations:

- M(meta): Make a request for meta-computing. This causes the execution of an object to be suspended and control to pass from the object to its meta-object.
- R(reflect): Resume object execution, i.e. pass control from the meta-object back to a base object.
- CBind: Specify a recipient for an interrupt. Interrupts will be delivered to the recipient as messages.
- CUnbind: Break the association between a recipient and an interrupt.

MetaCore is shown in fig.2 as the vertical box running alongside the meta hierarchy. Since MetaCore supports the basic facilities for meta-computing, i.e. the reify(M) and deify(R) operations, it is accessible and used by all meta-spaces. In table 1 we give perf mance figures for these basic operations[1]. N that operations without trap occur when we not cross a protection boundary.

Table 1: Execution cost of ***MetaCore*** pri tives (in μsec)

primitive	*on i486*
M	21.1
M(w/o trap)	13.0
R	22.6
R(w/o trap)	8.8
CBind	4.1
CUnbind	3.5

The key meta-spaces

***m*COOP** is the first level meta-space for a b level object, it implements the scheduling p icy and provides a concurrent object progra ming model for base objects.

***m*Zero** is one of the terminal meta-spaces all other meta-spaces and is thus accessed all other meta-spaces. This allows some op ations, eg. context location. to be optimise

m*Core** is the reflector for ***mZero and c tains meta-objects to handle activity, phys memory and virtual memory. Although, mentioned above, ***m***Zero is the shared ref tors for all others, it too needs a reflector, he the use of ***m***Core. To halt to potentially finite hierarchy of meta-spaces we make meta-space for ***m***Core be ***m***Zero. Altho this introduces a circular dependency betw the two it allows us to 'bottom out' when m ing down through the meta hierarchy.

[1]All performance figures from a 486DX2-66M 16MByte PCAT compatible.

Activity

The underlying CPU is abstracted in a structure known as a **context**. Contexts[2] are created by the meta-object *exec* which resides in ***m***Core. Contexts are actually abstracted by a notion of **Activity** which is the unit of manipulation at the meta-level (see fig.3). The context structure is seen by MetaCore as it uses this to set up the hardware to support the actual execution of an activity. The context structure contains the pointer to the meta-space for that context. This enables MetaCore to move from the currently executing context to its meta-space when the M operation is called.

Figure 3: Activity and the execution path to exec metaobject

Optimisations

Our initial work on optimisation has concentrated on its use in the lowest levels of the OS, i.e. the hardware interface. We begin this discussion by outlining some of the details of the driver metaspaces.

***m*Drive and *m*System** are meta-spaces for device driver programming (see fig.2). They implement a concurrent object environment

[2]Contexts in Apertos should not be confused which the general OS concept of a context as an address space. They are simply the state of the machine registers

where device drivers are written in the sa
way as any other concurrent object applicati
i.e. we are able to hide the restrictions norma
visible to device driver programmers. ***m***Dr
supports device driver objects that have di
physical access to hardware and manage in
rupts, and ***m***System supports device driver
jects that have no access to hardware.

We implement device drivers as sing
threaded concurrent objects. The advantage
this is that we can completely hide any comp
mutual exclusion operation normally asso
ated with device driver programming. Inst
the details are managed automatically by
meta-objects in the device driver meta-spa
Device driver objects are programmed w
out recourse to shared data structure and co
munication between objects is carried out
passing messages. Service entry points and
terrupt handlers are implemented as meth
of device driver objects which are invoked
message reception. In this way we can prov
a safe programming environment for progra
mers who write very low-level system p
grams.

One complication when writing dev
driver objects is that we need to have a v
to synchronize interrupt handler methods
hardware manipulation methods so that we
cleanly switch between hardware manipulat
and high priority interrupts. We achieve
in an object-oriented fashion by using con
uation meta-objects. Continuations perfor
similar role to 'futures' as used in langua
such as Actor[1] and ConcurrentSmalltalk[1
However we use them for synchronizing me
ods of an object, not to synchronize betw
two objects.

Table 2 gives some brief details of execut
costs for interrupt handling and using conc
rent objects in the driver meta-spaces. Det
can be found in [8].

As can be seen our approach has been
maintain the concurrent object oriented mo
supported throughout Apertos even at the
vice driver level. We believe that this appro
provides a cleaner way to write device driv
which has always been something of a bl
art, and hides some of the inherent complex

Table 2: Costs of mDrive services, and interrupt operations (in μsec)

metaoperation	*Apertos*	*BSD/386*
Interrupt message delivery	25.0	11.5
Null interrupt handler execution	44.2	16.2
Send metacall overhead on mDrive	108.6	—
Call-Reply roundtrip on mDrive	207.8	—

within the meta-space mechanisms.

Context hand-off

Because our system uses the object/meta-object model throughout, the general case execution path constantly moves from base to meta. As discussed above, we are using a concurrent object model so that each object (including meta-objects) has it's own execution context. Thus our system is forced to constantly switch contexts as it traverses the meta-space hierarchy. Obviously our first target for improvement is in the implementation of this context switching. We have implemented a variant of thread hand-off which we refer to as context hand-off.

Context hand-off will occur when an object issues a synchronous (Call-and-Reply style) method call to another object. In our basic implementation, there will be at least 4 context switches[3] when issuing synchronous method call, i.e. switch to reflector for call, go up to callee, switch to reflector for reply, go back to caller. If it is safe to share the execution context the context hand-off mechanism will use a simple function call for the synchronous method call. This means that the caller's context will be used for executing the callee's method.

The following conditions should be satisfied to use context hand-off mechanism:

- caller and callee share the same address space, and are in the same protection boundary.
- various state information, such as in rupt masks or CPU execution mode, k in the execution context are the same.
- it is acceptable to skip the scheduler w calling into the reflector. If the times for the caller has expired, or will soon pire, then skipping the scheduler is not ceptable.

It is the job of the reflector to decide if we ar use the context hand-off mechanism or not manages various state values on behalf of ba level objects and monitors behaviors (every ternal event from an object will go through flector).

This use of state information in the ref tor is a classic case of the use of meta-data a basis for a reflective operation. The reflec uses the data that describes the state of the b object to make a decision about how a syst operation, in this case synchronous call/re is implemented. When it decides that a c text hand-off is acceptable, it manipulates implementation of the mechanism to optim the code path. I.e., it carries out computat at the meta-level and reflects the changes b into the system.

To ensure sufficient application control meta-level method which allows context ha off to be explicitly disabled by a request fr the base-level objects is available if needed

The performance benefits of this techni come from the avoidance of context-switc and are bought at the small cost of the de sion making code held in the reflector. The p formance benefits from this simple techni are impressive, in table 3 we show the be fits gained within the driver environment w

[3]It is important to bear in mind that we use the term context switch here to mean a lightweight switch between two concurrent objects, not as a heavier weight notion consiting of a complete address space switch.

the use of context hand-offs. In this evaluation, context hand-off avoids fourteen context switches since we normally need context switches between objects and between schedulers which are also implemented as single-threaded concurrent objects with reflectors. It should also be noted that in the case of the context hand-off decision failing, i.e. deciding that the four criteria mentioned above are not met, then the performance is not affected.

To confirm that these benefits are consistent at a macroscopic level we have used the context hand-off mechanism in the implementation of the IP protocol handlers. Based on our design principle, the network protocol handlers are implemented onto an appropriate system service layer for protocol handlers [12]. We have implemented IP network protocol handler objects onto the system service layer mSystem. There is a concurrent object per each network protocol, five in total, passing IP packets as message between them. We have measured the execution time after reception of ICMP echo packet through the transmission of ICMP echoreply packet. Table 4 shows the results, indicating that optimised execution is 10 times faster than the non-optimised execution[4] In the normal case there will be at least 54 context switches (dependent on timing conditions such as timer interrupts) during the execution of the non-optimised execution. In the optimized execution we avoid all the context switches.

Message batching

We have also implemented a technique which exploits message asynchrony to allow us to batch messages based on message send patterns.

Again we use the meta-object model by using reflectors to gather statistics on message communication patterns. As discussed above, this is possible because all message communication goes through the reflector. We then allow the meta-space to re-configure itself to change the semantics of the send from a simple

[4]We do not provide performance figure for the UNIX variant because the implementations are completely different.

send per invocation to a batched send mo
As before, the meta-space can be configure
not batch asynchronous messages by invok
a meta-level method from the base-level
ject.

Fig.4 gives a graph of the initial perform
speed up using this batched message send te
nique. The horizontal axis indicates the nu
ber of asynchronous messages to be actu
batched into one packet; the vertical axis i
cates the time to process all the message se
ing requests. As you can see, if the bat
ing is enabled and there is only one mess
in a packet, there is a slight overhead.
source of the overhead is the processing t
for batching. We can reduce execution tim
we can batch 2 or more messages into sin
packet.

Figure 4: Effect of simple message batchi

While the use of message batching is in it
not a new technique, the ease with which
can add this technique to our system is a fu
tion of the meta-space architecture. Since
system implements message send as a m
operation, and since we have a meta ob
that describes how that operation is perfor
it becomes simple to use reflection to cha
the way we send messages. Again, at the
plementation level, because our model mo
from base to reflector on all meta-operation
provides a convinient place to monitor sys
operation. In this case we simply instrum
the reflector to monitor message traffic, tha

Table 3: Costs of mSystem services (in μsec)

metaoperation	*Apertos*
Call-Reply roundtrip on mSystem(context hand-off not implemented)	772.8
Call-Reply roundtrip on mSystem(context hand-off succeed)	7.8
Call-Reply roundtrip on mSystem(context hand-off failed)	693.0

Table 4: Costs of network protocol handlers on mSystem (in μsec)

metaoperation	*Apertos*
ICMP echo-echoreply roundtrip	3466.0
ICMP echo-echoreply roundtrip(optimised)	362.3

make message traffic statistics into meta-data, and use this meta-date as the basis for the reflective operation.

Dynamic object reallocation, and context handoff

A related technique we have adopted is the use of address space change combined with context hand-off. The context hand-off technique only works when two objects share the same protection domain limiting its use. However, Apertos implements every system entity using fine-grained concurrent objects allowing several objects to be allocated onto a single address space. In Apertos, unlike UNIX and other operating system with fixed design policies, various aspects of the system such as protection, activity (CPU thread allocation), object persistence, and message passing semantics can be tailored to meet the needs of the system designers. Therefore, if performance is the critical issue for the objects and protection is not an issue, (for example they are written using a language system with strong type inference) the system designer can use a meta-space which allocates objects onto the same address space and always utilizes the above described optimisation techniques. If performance is not an issue and protection is important, one can provide a meta-space which implements st
protection policies. An object can choose
of these policies on the fly, by migrating
tween two meta-spaces.

Again Apertos utilizes fine-grained conc
rent object, these objects can be moved fr
the current address space to other addr
spaces. By doing so, we can tailor object
location on a per address space basis allow
us to maximize the utilization rate of the o
misation techniques. This technique resemb
the load balancing techniques used between
mote CPUs.

Run-time code generation

The last area of work that we have explore
the use of run time code generation and par
evaluation. It is often the case that we can o
mise the general case execution path provi
at initial system load based on application
quirements.

Again we use the meta-space to gather
time information about the execution char
teristics of the application. By using the m
object interface we are able to adapt the c
execution to the application requirements.

This work is based on our initial inve
gation in the Cognac language [11]. Cog
uses static analysis and type information

drive run-time binding of an object to its meta-objects. By dynamically changing these run-time bindings we are able to maintain the same base programming interface, but change the semantics of the system level operations that support the application object. Also we are investigating ways to utilize partial evaluation techniques to flatten out meta-object hierarchies, which can then be statically linked into the base-level object.

Conclusion

We believe that operating system flexibility is the most important goal of current research. However we would argue that flexibility must be provided within the framework of a clean and consistent architecture. Our approach, the Apertos operating system, uses the meta-object model and is targetted toward implementing the whole operating system within a reflective programming model. This approach has produced an operating system which is highly flexible at all levels, however, it has some performance drawbacks. In this paper we have discussed ways to minimize the execution cost by using the meta-object model in full and have shown that it is possible to exploit meta-object to gain significant performance advantages without breaking the model.

Current and future work

We are currently working on the use of Apertos in a distributed multi-media environment. In our initial set up we have ported Apertos to a number of Sony News 5000 workstations with MIPS R4000 processors and a video server using the R3000 processor. These machines are interconnected with a small experimental ATM network. We are experimenting with Apertos in two areas; the first uses the meta-object model to support QofS constraints and exploits our flexible architecture to adapt the resource scheduling policies to varying simulated system loads and failures. The second area of work uses Apertos in a proprietary graphics engine attached to the above network where we are experimenting with protocols for d tributed shared virtual environments. In par ular we are looking at the issues of distribu consistency where we are exploring the of migration between meta-spaces to dyna ically change the consistency associated v shared objects. Also, we plan to investig the use of object migration (to the remote ho for downloading the whole system code incl ing device drivers, to the graphics engine. ing meta-spaces as a clear abstraction for v ious visibility control, such as host bound persistent objects and volatile objects, is a planned and will be carried out in the next velopment phase.

In addition we are continuing to analyse performance of our system and hope to ext the techniques discussed above to all m spaces. We hope to report in the future on success of this work.

Availability of Apertos Operati System

The Apertos system is available for resea purpose and information, along with a col tion of papers can be found by accessing following URL:

```
http://www.csl.sony.co.jp/
ftp://ftp.csl.sony.co.jp/
```

References

[1] Gul A. Agha. Actors: A Model Of C current Computation In Distributed S tems. Technical Report 844, MIT, J 1985. Technical Report 844.

[2] Tevanian Avadis Jr. and Richard Rashid. MACH: A Basis for ture UNIX Development. Technical port, Department of Computer Scien Carnegie Mellon University, June 198

[3] Nariman Batlivala, Gleeson Barry, Ha rick Jim, Lurndal Scott, Price Darr Soddy James, and Abrossimov Vad

Experience with SVR4 over CHORUS. In *Micro-kernel and other kernel architectures*, pp. 223–241. USENIX, April 1992.

[4] Brian Bershad. SPIN - an extensible micro-kernel for application specific operating system services. In *Proceedings of the 6th ACM-SIGOPS European Workshop: Matching Operating Systems to Application Needs*, September 1994.

[5] C Bryce, V Issarny, G Muller, and I Puaut. Towards safe and efficient customization in distributed systems. In *Proceedings of the 6th ACM-SIGOPS European Workshop: Matching Operating Systems to Application Needs*, September 1994.

[6] Roy H. Campbell, Nayeem Islam, Ralph Johnson, Panos Kougiouris, and Peter Madany. *Choices*, Frameworks and Refinement. In *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 9–15. IEEE Computer Society Press, October 1991.

[7] Graham Hamilton and Panos Kougiouris. The Spring nucleus: A microkernel for objects. In *USENIX 1993 Summer Technical Conference Proceedings*. USENIX Association, June 1993.

[8] Jun-ichiro Itoh, Yasuhiko Yokote, and Mario Tokoro. SCONE: A New Execution Model for Low-Level System Programming based on Concurrent Objects. Technical Report SCSL-TM-95-005, Sony Computer Science Laboratory Inc., 1995. submitted to OOPSLA95.

[9] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[10] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL: System support for distributed programming. *Comm nications of the ACM*, Vol. 36, No. September 1993.

[11] Kenichi Murata, Nigel Horspool, and suhiko Yokote. Design and specificat of cognac. Technical Report SCSL-T 94-006, Sony Computer Science Labo tory Inc., 1994.

[12] Kenichi Murata and Yasuhiko Yok A Reflective Network System Us Concurrent Objects and Meta Archit tures. Technical Report SCSL-TM- 010, Sony Computer Science Laborat Inc., July 1993.

[13] Carlton Pu and Jon Walpole. study of dynamic optimisation te niques: Lessons and directions in ker design. Technical Report CS/E 93-0 Oregon Graduate Institute (OGI), A 1993. Technical Report CS/E 93-007

[14] Yasuhiko Yokote. *The Design and Imp mentation of ConcurrentSmalltalk*. Wo Scientific Publishing, 1990.

[15] Yasuhiko Yokote. The Apertos flective Operating System: The C cept and Its Implementation. In *P ceedings of Object-Oriented Progra ming Systems, Languages and Appli tions in 1992*. ACM Press, October 19 Also appeared in SCSL-TR-92-014 Sony Computer Science Laboratory I

[16] Yasuhiko Yokote, Gregor Kiczales, John Lamping. Separation of Conce and Operating Systems for Highly H erogeneous Distributed Computing. *Proceedings of the 6th ACM-SIGOPS ropean Workshop: Matching Operat Systems to Application Needs*, Septem 1994.