INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Matching Micro-Kernels to Modern Applications using Fine-Grained Memory Protection

Ciarán Bryce & Gilles Muller

N° 2647

Août 1995

PROGRAMME 1

ISSN 0249-6399

Matching Micro-Kernels to Modern Applications using Fine-Grained Memory Protection

Ciarán Bryce* & Gilles Muller

Programme 1 — Architectures parallèles, bases de données, réseaux et systèmes distribués
Projet SOLIDOR

Rapport de recherche n° 2647 — Août 1995 — 16 pages

Abstract: Scalable distributed systems, systems whose processing power remains proportional to the number of component processors, require a programming methodology where an application developer may take existing software modules and plug them together to form a new application. To allow mistrusting modules to interact, the underlying kernel support must offer protection barriers which do not impede performance. The wide-ranging nature of modern applications used on larger scale systems means that existing kernel functions may not necessarily be the most efficient for an application. The kernel must therefore allow an application to dynamically install a function in the kernel; this is one aspect of **customization**. This paper argues that customization support is one aspect of fine-grained protection for modules needing CPU supervisor privilege. We describe the kernel support required for fine grained protection. Basically, our approach relies on the assignment of a single address space to an application with application modules having their own **domain of protection**. An experiment was made by modifying the Mach kernel; results show that inter-domain communication by **protected procedure call** is up to 5 times faster than Mach 3.0 IPC.

Key-words: micro-kernel, fine-grained memory protection, protection domain, system customization

(Résumé : tsvp)

To appear in the procedings of the seventh IEEE Symposium on Parallel and Distributed Processing

*Ciarán Bryce's current address is: GMD - German National Research Center for Information Technology, 53754 Sankt Augustin, Germany.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Intégration d'un mécanisme de protection mémoire à grain fin dans un micro-noyau : prise en compte des besoins des applications distribuées modernes

Résumé : Les systèmes distribués extensibles sont caractérisés par une puissance de calcul qui croît avec le nombre d'éléments processeurs. Étant donné la complexité de ces systèmes, leur programmation s'oriente vers une méthodologie de développement reposant sur la composition et la réutilisation de composants logiciels existants. Comme ces composants sont par nature hétérogènes, le noyau doit offrir un mécanisme de protection qui permette une coopération entre des composants de niveau de confiance différents sans pour autant sacrifier les performances. Par ailleurs, la diversité des applications rencontrées sur des grands systèmes entraîne une multiplication des politiques de gestions des diverses ressources qui doit être mises en œuvre dans le noyau. Il en résulte une inefficacité du noyau soit parce que les politiques existantes sont trop généralistes soit parce qu'elles sont trop spécifiques. Une solution repose sur l'installation dynamique, par application, de fonctions systèmes ; cette fonctionnalité est un aspect de la **spécialisation** de systèmes d'exploitation. Nous montrons que le support système de la spécialisation relève de la protection mémoire à grain fin, i.e., les fonctions systèmes sont des composants logiciels possédant le privilège superviseur. Cet article présente un mécanisme de protection à grain fin permettant de satisfaire ces besoins de protection. Notre approche repose sur l'association à chaque application d'un espace d'adressage unique, chaque composant de l'application possédant son propre **domaine de protection**. Notre mécanisme a été intégré au sein du micro-noyau Mach 3.0 ; les résultats de nos expérimentations montrent que les communications entre domaines par utilisation d'un **appel de procédure protégé** peuvent être jusqu'à cinq fois plus rapides que l'IPC de Mach 3.0.

Mots-clé : micro-noyau, protection mémoire de grain fin, domaine de protection, spécialisation de systèmes

1 Introduction

A distributed system is said to be *scalable* if its processing power remains proportional to the number of its component processors. High bandwidth networks and increased processor speeds means that scalability is now viable, even for large scale architectures. Larger architectures attract more users and consequently more wide-ranging applications. To reduce software development costs, a convenient programming methodology for applications in such systems is to build the application from several existing user-level and kernel-level software modules (e.g., window managers, databases).

1.1 On the Need for Fine-Grained Protection

The modules available to a programmer are heterogeneous in several ways. They may be written in different programming languages, they may come from different development sources (e.g., different development groups, public domain, software vendors) or they may be at different stages of development (e.g, debugging, final testing). One implication of this is that any application will be composed of modules of differing trust levels. Consequently, the underlying system must be able to *isolate* the software components without penalizing inter-module *communication performance*. More specifically, we need to be able to construct a trusted application from non-trusted components. Moreover, in-built operating system components are not always able to match the requirements of a given application [2]; consequently, it should be possible to *safely install* an application module which can access the system's low-level resources and therefore implement an application-specific policy. The replacement of an existing system function with an application specific function, or installation of a new one, is known as *customization*. Customized functions do not need to have access to all of the kernel's address space; rather, they run in a protection unit within this space. We thus argue that customization support is one aspect of fine-grained protection for modules needing CPU supervisor privilege. Consequently, both should be handled by the kernel within a single paradigm.

In order to provide efficient support for fine-grained protection, we propose to introduce multiple *domains of protection* (DP) within a task, thereby preserving the association of a unique address space with an application and consequently easing data sharing, since all entities of the application name each other with the same names. A DP defines a set of memory code and data regions with the property that the data may only be accessed by a thread executing the code of the domain; a thread changes protection domains by executing a *protected procedure call* (PPC). This paper describes the extension of a standard micro-kernel, Mach 3.0 [1], incorporating this paradigm, while preserving binary compability, thus allowing the reuse of existing software.

1.2 Why Enrich the Kernel with Domains of Protection

Application programmers formerly relied on language compilers to detect protection violations, when this was possible, though the presence of modules written in different programming languages complicates this issue. Another approach is software-based fault isolation, where the compiler inserts instructions into the code which verifies the legality of addresses used [16]; this however leads to an overhead for each address dereference. Consequently, due to the lack of a convenient protection mechanism, the only solution is to place mistrusting application components in different tasks. Tasks as protection units have successfully allowed the implementation of protected sub-systems (e.g., OS personalities, file systems, communication layers) using the micro-kernel technology and the client-server paradigm. However, using multiple tasks within a single application complicates programming and makes inter-domain communication costly. As a result, application programmers often choose to ignore protection for their applications.

In kernels like Mach, communication is heavy partly due to the necessity of packaging the message into a port, switching threads and then switching address and port spaces. We adopt the procedural model of inter-domain communication and avoid task switching. The mechanism also takes advantage of the fact that DPs share a common address space to optimize for cases where DPs overlap and where the TLB contains entries visible in the called and calling DPs, thereby avoiding the "cold-start" overhead of a switched domain. These optimization techniques, described in a later section, mean that a PPC can be up to 5 times faster than a Mach 3.0 IPC.

Finally, our kernel model also simplifies the application development process. During the debugging phase, modules under test must be placed in different fault domains at least until a sufficient degree of mutual confidence is reached [16]; this usually means different tasks. Since an application prototype using DPs runs in the unique address space that the completed application will eventually use, the transition from the prototype to the running system is simplified since only the linking procedure is altered.

1.3 Plan of Paper

The remainder of this paper is organized as follows. The next section outlines the kernel model. Section 3 describes important points of the implementation, in particular, PPC and the modifications made to the Mach kernel. Section 4 presents a performance evaluation for a PC/i486 platform. Section 5 looks at related work and our conclusions are presented in section 6 along with a description of current work.

2 Introducing Domains of Protection into the Micro-Kernel

2.1 Basic Principles

In our kernel model, an application is associated with a *task*. The task is almost the same abstraction as in Mach; it provides a unique naming environment, containing an address and port space; in addition, it includes a set of *domains of protection* (DPs). A DP is a subset of a task's memory space. A thread executes in a single DP at any given moment and can only access the memory regions belonging to, or *visible* in, that DP. Like all Mach abstractions, a DP is named by a port.

The change in a thread's DP visibility is effected by a protected procedure call (PPC), a mechanism similar to LRPC [3], where a thread traps into the kernel and comes back out executing in the called domain. On a PPC, there is no need to change the name environment or any task related kernel structure. Entry points into a DP for PPCs are specified by a *dp_register()* system call: entry points are the addresses from which a thread executes on entering into a domain. There are several other extra primitives in the model; these are for creating and destroying DPs as well as for rendering memory regions already mapped into a task visible and invisible to a DP.

The kernel model has two special kinds of DPs:

- DP_ROOT: of which there is one per task and which has visibility over all of the task's memory regions. DP_ROOT is the default DP and is automatically created on task creation. Standard Mach applications execute in DP_ROOT without any modification to their binaries.
- *Supervisor DPs*, are DPs in a task address space in which threads execute with supervisor CPU privilege, thus allowing customization to be implemented on an application basis. That is, since a thread executes in a supervisor DP with high privilege, an application can use its own trusted operating system routine. One such example is a specialized driver.

Access to ports is also controlled on a DP basis. As is the case with addressing, ports are uniformly named by threads of a task, no matter what their current DP; a *dp_pass_port_right()* kernel operation is used to transfer a port right from one DP to another.

2.2 Protected Modules

From an application's point of view, a DP contains a *protected programmed module*. This is an encapsulated entity, that is, it can only be accessed through its procedural interface. Protected modules are associated with stubs which push the parameters and calls the *dp_call()*

trap; this is the kernel traps which changes DPs and retarts the thread in the called DP. A PPC possesses the following signature:

ppc (dp_port, entry_point, proc_id,[capability], procedure parameters)

The *entry_point* parameter identifies the protected module; it is the value of the entry point address for the DP specified in the *dp_register()* call; there may be several entry points to a DP. The *proc_id* parameter names the procedure of the protected module to be invoked; it is an integer which offsets into a jump table for the protected module. The *capability* parameter is optional; its goal is to control access by other protected modules to it. In our model, a capability is a sparse password capability (a random number), kept in user space, which is created for the called module on initialization; the module must validate the capability itself on each PPC. The fact that the capability mechanism is implemented in the protected module means that the module can implement its own access control policy: algorithmic based access control mechanisms are more expressive than traditional data structure based mechanisms such as the old capability systems [10]. Other systems using this password capability approach are Opal [6] and Amoeba [12]; related work is discussed in section 5.

Protected modules contain a header code which is invoked on each PPC; this code is placed at the enclosing DP's entry point. The header manages parameter passing data structures and verifies the validity of the capability parameter. Protected modules are mainly created by the operating system loader; a loader not only creates tasks and maps memory regions, but it must also create the DPs for the module within the task and assign the entry points using *dp_create()* and *dp_register()* respectively. It then makes some of the task's memory regions visible in the DPs. Finally, after having installed the code and memory segments of the module, it can choose a capability.

A loader is just a task and any other task may also initialize a protected module. This delegation of DP creation is important in applications requiring dynamic module creation such as object-based systems. Supervisor DPs are installed, by necessity, by a special operating system loader task, LOADER. Only LOADER can set entry points or change the memory visibility of supervisor DPs. To enforce this requirement, the operating system creates a 512-bit password capability on booting; this capability is needed for all modifications to the entry point set and memory visibility of supervisor DPs. LOADER, one of the first tasks created on booting, can read the value of this capability. This approach requires that LOADER be trusted not to divulge the value of the capability to other tasks, but, in any case, a user has no business using an operating system whose loader is not trusted. Finally note that calling a supervisor module is no different from calling a normal module from the application programmer's point of view: the signature given above is also used.

3 Implementing Domains of Protection

This section is devoted to the implementation of DPs. We first describe modifications made to the Mach 3.0 kernel for managing DPs and then the PPC implementation and its possible optimizations.

3.1 Modifying the Mach 3.0 Micro-kernel

Modifications to Mach mainly concern the memory subsystem even if some minor changes have been made to the port and thread subsytems. The memory subsystem is made up of the following entities: *vm_map*, *pmap*, *kernel k_objects* and *pages*.

- *vm_map*: This structure represents the task's address space. It contains a list of *vm_map_entry*s, one for each of the task's memory region. A *vm_map_entry* defines the region's inheritance values, memory manager, size and a pointer to the kernel k_object containing pages of the region currently mapped into memory. The *vm_map* module implements Mach virtual memory operations such as *vm_protect()*, *vm_allocate()*.
- kernel k_object: This structure is used for managing (those parts of) regions currently mapped into physical memory. A k_object principally contains the port of the region's external pager and a list of physical pages.
- Page: The page lists contain entries for all pages of physical memory. They indicate if a page has been read or modified, if it is in the process of being written or read from disk, if the page must be returned to the memory server or if the physical page is free or non-existent. A page descriptor (entry in a page list) contains the physical address of the page as well as a pointer to the external pager to which the virtual page belongs.
- PMAP: This structure contains the virtual address to physical address bindings of the task. The pmap module is processor dependent and there is a single PMAP structure per address space. The *pmap* module's role is to implement the following operations: find the physical address bound to some virtual address, remove this binding or add new ones, find the virtual addresses associated with some physical address, change the protection rights associated with some address binding, and to verify if a given virtual memory page has been read or written.

The kernel in Mach has its own address space with its own PMAP structure; the kernel space is mapped into all task address spaces at the same virtual address in the high part of each task's virtual address space.

One of our implementation goals was to keep the modifications as much as possible to the *vm_map* and *pmap* modules. The PMAP structure now references an address space containing several DPs. Several operations have been added to the *pmap* module in order to create, destroy, activate and deactivate DPs as well as change the visibility of a memory region in the DP.

The *vm_map* module has been modified as follow: (i) each *vm_map_entry* contains a list of the DPs in which a region is visible; (ii) an additional structure, *dp_map*, is allocated per task and contains a list of *dp_map_entry*s, one for each of the DPs of the task. A DP's *dp_map_entry* contains a list of the regions visible in the DP, the *rwx* rights, a flag indicating if the DP is supervisor or not, port, parent task pointer, inheritance values for cases where a DP is a clone of its parent.

As presented in section 4, our approach has been evaluated on a PC/i486 platform. Typical CISC processors such as the MC68000 and Intel x86 families do not support the notion of protection domain in their virtual memory subsystem. An address space is linear and is represented by a translation tree. A thread executing in a task has direct access to all memory pages currently resident in physical memory and referenced by the task's memory tree. Since each memory reference must be validated, the default implementation of a DP is to assign it its own translation tree, which will obviously be a subset of the task's translation tree. Changing DPs thus involves changing the page table root pointer.

When a page fault occurs in Mach, the kernel consults the *vm_map* of the task to determine to which memory region, and thus to which memory object, the page belongs. The kernel verifies that the page is not actually in memory before sending a request to the page's external pager. The entry in the translation tree is updated once the page is brought into main memory. In our case, only the PMAP of the DP which generated the fault is updated; if the page is visible in other DPs, then their translation trees will be updated whenever they generate a page fault. Since the page can be in memory, fault resolution in these latter cases is less costly. This approach corresponds to the Mach philosophy of *lazy evaluation* [1].

Whenever a region is mapped into a task, only the page tree of DP_ROOT is updated since only this DP can initially access the region until a *dp_make_visible()* is executed. Whenever the protection rights of DP_ROOT on a memory region are changed, the kernel verifies that no other DP has more rights for the region than DP_ROOT. All the information necessary for this call can in any case be found in the *vm_map* structure of the task.

Currently in Mach, whenever the kernel must verify if a page has been references or modified, it consults the translation tree of each task which has mapped the page. Thus, by assigning one pmap per DP, there is no need to alter neither this algorithm nor its implementation.

3.2 Basic PPC Execution Schema

The PPC implementation is based on an optimization of LRPC [3] for a single addressing environment. LRPC is itself an optimization of RPC for address spaces situated on the same processor and where the parameters transferred are non-complex. The essential difference of

PPC to (L)RPC is that, in PPC, there is no need for special procedures which package complex pointer structure parameters (like linked lists) since these entities are named universally by the calling and called domains via the application's address space.

There are three basic data structures in a PPC: *A-stack*s, *E-stack*s and *linkage segments*. An A-stack is a memory region mapped into the calling and called domains and is used to hold the transferred parameters. An E-stack is a memory region allocated for the stack of the executing thread; a thread thus possesses a sequence of E-stacks, one for each of the domains that it has traversed. However, a thread can only access the E-stack of its current domain. The linkage segment is a kernel visible entity which, for each PPC, holds the thread's return program counter and the stack pointer address for the calling domain.

Phase 1 - Compilation and Loading During compilation and loading, communication stubs are generated for the calling and called modules - there being a generator for each language. To avoid (in C) the double copy of parameters on a procedure call - a copy on the thread stack followed by a copy onto the A-stack: the stubs are implemented as assembly code macros. The code of the stub chooses an A-stack which is free. In the case where the calling and called DPs are not disjoint, then the use of A-stacks is not always necessary. This is particularly the case where the called DP is an extension of the calling since the caller can read parameters directly. The return parameters must then be placed in caller visible memory. On DP creation, the kernel allocates memory in the DP for the E-stacks. The number of E-stacks in a DP represents the maximum degree of parallelism of the DP.

Phase 2 - Connection This phase installs the communication channels between the DPs. The A-stacks are mapped into both domains using the *dp_make_visible()* primitive. The number of A-stacks represents the maximum number of concurrent PPCs to the domain. The connection is normally made on the first PPC between the two domains.

Phase 3 - Call & return When a module makes a PPC, its communication stub: chooses a free A-stack, stacks the parameters onto the A-stack and calls the system trap *dp_call()*. In the *dp_call()* trap, the kernel, saves the stack pointer and the return address in the thread's linkage segment, chooses a new E-stack, changes DP memory visibility, verifies the validity of the *entry_point* and of the procedure identifier and then and resumes the thread execution at the procedure address. On returning from a PPC, the called module executes the *dp_return()* trap which, recovers the old stack pointer and return address, changes the DP memory visibility, resumes thread execution and the calling stubs then frees the A-stack.

3.3 Optimizing PPC

There are two possible optimizations for PPCs arising out of the unity of the application's address space: (i) optimizing calls to supervisor DPs, (ii) selectively invalidating TLB entries.

3.3.1 Optimizing PPC for supervisor DPs

The PPCs to supervisor modules, i.e., those modules which run in supervisor DPs, also give rise to optimizations. TLBs generally contain an attribute for each entry indicating if the entry is visible only in user-space or also in supervisor space. Thus, the MMU has a functionality similar to that of a protection lookaside buffer [9] and region visibility is fully managed by the MMU when changing the cpu privilege within the *dp_call* trap and return. Moreover, there is no need to allocate an A-stack since the supervisor DP has kernel visibility; the routine is forcibly trusted so that there is no need to allocate a new E-Stack.

3.3.2 Optimizing PPC with selective TLB invalidation

In the basic PPC schema, a DP switch is implemented by changing the MMU translation tree root pointer, thus invalidating all TLB entries on some CISC processors (e.g., i486). However, when the calling and caller DPs overlap, some TLB entries are unnecessarily invalidated - an entry which has just been flushed may be refilled. For instance, about 25% of the overhead in a nil LRPC call is due to the initial TLB misses in the called domain [3]. Thus, on a DP switch, it is sometimes better to modify the translation tree and then to selectively invalidate TLB entries than to perform a global flush. The TLB overhead in the case where the TLB is flushed equals the cost of invalidating the TLB plus the cost of the misses on the entries in F, where F is the set of addresses used by the called DP which would lead to an entry of the TLB being loaded for the first time were the TLB initially empty. Subsequent loads do not interest us as the cost of these misses is not influenced by whether the TLB was globally or selectively invalidated at DP switch time. tlb_{glob} is the cost of globally invalidating the TLB, tlb_{miss}, the cost (in terms of supplementary processor cycles) of accessing a memory word when its virtual address is not in the TLB, # denotes set cardinality. Thus, the overhead of a global TLB invalidation is:

$$tlb_{glob} + (\#F * tlb_{miss})$$

In the general case, the overhead of the selective invalidation caller is the cost of invalidating entries in the TLB not visible in the called DP, plus the cost of the TLB misses on the set F for its elements not already in TLB, plus the cost of altering the page table entries (PTE) to reflect the new visibility:

$$\#(N \setminus C) * tlb_{sel} + ((\#F - \#(F \cap C)) * tlb_{miss}) + (D * y)$$

Parameters are: tlb_{sel}, the cost of selectively invalidating an entry in the TLB. N, the set of entries in the TLB following a DP switch but before execution starts in the called DP. C is a subset of N, containing those TLB entries which are valid in the called DP. y is the cost of writing a new value to a PTE. D is the difference in page size between the two DPs, that is, the number of pages visible in the called DP though not in the calling. The term $\#(F \cap C)$ depends on the behavior of the called function: how many caller's entries are reused. Its value falls between [0 .. #F]. Let us rename it in $\alpha\#F$ with α in [0 .. 1]. Consequently, a selective invalidation is cheaper when:

$$(\#(N \setminus C) * tlb_{sel}) - (\alpha\#F * tlb_{miss}) + (D * y) < tlb_{glob}$$

Analysing PPC to an englobing DP Englobing DPs are useful for implementing trusted sub-systems which do not need to run in a supervisor DP. Examples include safe object runtimes and secure execution environments where the application behavior is controlled. We now analyse the gain of selective invalidation when calling an englobing DP with the i486 as a target processor.

On the call, there is no need for TLB invalidations since addresses of the caller are also visible in the called DP. $(N \setminus C)$ is therefore zero (N will contain kernel entries after the trap though these obviously cannot be used by the called DP). In the i486, y equals 2: an instruction for loading the PTE value to a register and then a second instruction which writes it to the PTE address. tlb_{glob} equals 4. For a page in physical memory, tlb_{miss} is 13 processor cycles in the fastest case (worse for us), and 29 in the slowest case (where the referenced and dirty bits of the page's table descriptor must be set by the processor). Consequently, selective invalidation is better when:

$$D < ((\alpha \# F * 13) + 4)/2$$

The worst case, is when α is 0 ; D must be equal to 1. This is unlikely since there is sharing between the two DPs due to parameter passing. The best case is when α is 1; For a typical #F equal to 20, D must be less than 132 for selective invalidation to be more efficient. In practice, α depends mainly on the size and memory mapping of parameters. In an i486, the size of the TLB is 32 and kernel takes few entries during trap execution and PTE modification, thus reducing the value of α. Also, it should be said that we must use absolute addressing to invalidate a single TLB entry and so know the address beforehand from analysis. This information can be retrieved from link and binding phases.

For the return from call, F now applies to those entries loaded by the calling DP (i.e., results return by the procedure). The only difference to the above formula is that the first term for the selective invalidation $(\#(N \setminus C))$ becomes D ; the kernel cannot know what DP addresses visible in the called DP have been loaded into the TLB during the procedure call so it must remove all (a selective TLB invalidation is 11 cycles when the entry is there and 12 when it is not). For maximum and minimum α and an F-set of size 20, a selective invalidation turns out to be more efficient when D is less than 20 and 0 for minimum and maximum α respectively. The following subsection contains a more applicative evalution of the mechanism.

4 Performance Evaluation

In order to evaluate the performance of protection domains, we extended an OSF Norma MK 78 version of Mach 3.0. Most of the modifications are done in C; only a few assembly code routines were needed in the *dp_call()* and *dp_return()* traps for stacking the parameters to the C-implemented trap routines. For our test we have consider the execution of a small procedure Max (), which returns the maximum of two integers, in order to evaluate the overhead of our mechanims against other solutions. The results are shown in the first table below for a system running on a PC/i486 66 Mhz.

No.	Implementation	Execution Time
1	PPC (no TLB optimization)	24.2 μsecs
2	PPC (with TLB optimization)	16.2 μsecs
3	Separate Tasks	100.9 μsecs
4	User Library	0.8 μsecs
5	System Call (Trap)	6.7 μsecs
6	System Call (Port)	48.8 μsecs

The first row gives is the cost of the Max() procedure when executed in a DP disjoint from its calling module's. The cost is broken down as follows:

Task	Execution Time
A-Stack Management	1.5 μsecs
dp_call() trap	11.1 μsecs
dp_return() trap	11.1 μsecs
Max () Procedure	0.5 μsecs
Total	24.2 μsecs

As we mentioned, the TLB is invalidated in the *dp_call()* and *dp_return()* traps in the i486 implementation. Thus, the Max procedure is executed from a cold TLB; the cost of executing Max() with a warm TLB (in a loop) is about 0.2 μsecs.

The second row is the cost of executing Max () in a DP which is an extension of the calling DP. The TLB is selectively invalidated. The procedure is called in a loop and the TLB is still warm for the caller when the procedure returns. Moreover, the execution times of the two traps are reduced since the TLB invalidation that normally happens also invalidates kernel entries. The third row corresponds to an emulation of DPs using multiple tasks. PPC is implemented by a message on a port and *handoff scheduling* [5] is used. IPC is thus more than 5 times slower than an optimized PPC which it is still 20 times slower than a user level library implementation (see row 4) though there is no protection between the caller and the calling modules. The fifth row gives the cost of Max() when it is implemented as a trap based system call. The thread enters into the kernel via a processor gate; a jump table is then consulted which maps a system call number to a kernel routine. Mach system calls implemented in this way include *mach_thread_self* which returns the port of the executing thread. The execution time is very low and the called function is protected from the caller. The call (overhead) here is very similar to a PPC to a supervisor DP. However, this approach does not allow dynamic loading - the kernel must be recompiled, relinked and rebooted. The sixth row gives the execution time when the procedure is implemented by a system call which uses ports: the call is translated into a stub which sends the request to a port; the request is eventually read and processed by the kernel and the result is passed back in the same way. Finally note that as the duration of the called DP execution increases, the effect of the TLB-optimized/non-optimized tradeoff trickles out; there is still a performance gain since even a non-optimized PPC performs better than the traditional IPC.

5 Related Work

The ideas forwarded in this paper stem from three domains: protection, optimized IPC systems and customization.

Protection in Operating Systems Capability systems [10], and Hydra in particular, are probably the most achieved systems with protection features. Hydra provides capability-based operating system support for abstract data types. Protection domains in Hydra are heavyweight due to the complexity of the protection mechanisms. Moreover, although each entity is uniquely named, within a domain these entities are named by their index into the domain's capability list. Thus the system must convert names exchanged between domains on the fly. Not only do these mechanisms make inter-domain communication more costly, but the system programming model is not used by general purpose applications.

Domains of protection have also been introduced in operating systems based on a single 64-bit virtual address space such as Opal [6]. These systems resolve the problem of unity of addresses without compromising protection. The main advantage of single address space operating systems is to simplify memory management. However, this approach does have its own set of problems. Firstly, it requires processors with larger address spaces. Moreover, compatibility with standard libraries using absolute and PC-relative addressing modes is not ensured. In a single address space system, these modes designate a single address in a system whereas in a multi-space operating system, these addresses denote the private data of the modules. Notably, Unix cannot be run on these systems because of the semantics of the fork command where one address space is copied into another [6]. Finally, address contiguity for applications is more difficult to achieve since an application's naming environment is dependent upon others; this is important for data structures like records, objects and arrays belonging to low level languages such as C.

Optimized IPC Systems The best known works on optimized RPC is by Bershad *et al.* on *Light-weight RPC* (LRPC) [3] and L3 [11] with the latter kernel being designed from scratch having fast IPC as its dominant objective. The LRPC model has been adopted in several other systems. The Spring kernel [8] uses LRPC to support cross address space method invocations for an object-based system. Ford & Lepreau have integrated a migrating thread model into Mach 3.0 [7]; like our kernel, IPC is much faster and there is no loss of compatibility for existing OSF/1 binaries. Nevertheless, all these systems use tasks as protection units. In the DP approach, the primary goal was protection, though this eventually led us to optimise the IPC.

Customization in Operating Systems Customization in operating systems is currently one of the hotter topics in operating system research [2]. One of the most interesting works is SPIN [4]. In this kernel, application-specific code sequences called *spindles* are dynamically linked into the kernel; the interface and implementation of spindles is left open to the application programmer. A system allocator manages basic resources such as page frames and communication bandwidth; user allocators, implemented as spindles, rent the resources thus allowing applications to implement their own resource management strategies. SPIN's benefit relies in how to build more modular kernel. Our interest is seeing how customization should be handled by the kernel from a protection and address space point of view.

One way for kernels to allow a system function to be redefined for a particular application is to implemented the new function in the kernel. The Chorus micro-kernel [13] permits to dynamically add servers (e.g., *supervisor actors*) which are able to run with a supervisor privilege and share the kernel address space. In [15], the kernel dynamically synthesizes efficient RPC stubs from procedure signatures and links the code to itself. The major advantage of these two approaches is their efficiency - the customized code is directly accessible through a kernel trap. On the other hand, the major drawback is that they are not scalable: the size of the kernel's address space bounds the number of procedures that can be integrated within the kernel. Moreover, they introduce the need for a kernel space reclamation strategy and if the customization is on a per application basis, protection problems arise since customized code is globally accessible. The SunOS system [14] system has allowing hooks allowing users to install their own drivers in user space (and to install kernel modules), even dynamically without having to reboot the system. This is restricted though since user programs using them should run with supervisor privilege and some modes of device activity (e.g., DMA) are not possible.

6 Conclusions and Future Work

In this paper, we have discussed a method for implementing fine-grained protection domains within a Mach task's address space with a view to allowing modules of differing trust levels, and different development origins, to be integrated into a unique naming space ; this allows a trusted application to be built from non-trusted components.

Our results show that PPC provides fast intra-application RPC (e.g., 24.2 μsecs on a PC/486 66 Mhz); PPC outperforms Mach 3.0 IPC by a factor of 4. Furthermore, it is sometime possible to optimize PPC by selectively invalidating TLB entries, thus reducing PPC overhead (e.g., 16.2 μsecs). Concerning customization, our approach allows certain system functions to be tailored to a particular application's needs ; we are particularly interested in implementing application specific device drivers which run in supervisor DPs. Finally, it offers compatibility for Mach-based applications.

Our initial results indicate that protection domains can be implemented in Mach without extensive kernel modification. Though we have chosen Mach as a workbench to implement our approach, we believe that the approach is equally applicable to other micro-kernels such as Chorus [13]. The model is currently being developed into a complete versioning prototype on a Chorus platform where we would also like to evaluate our approach on modern RISC processors.

Acknowledgements The authors are grateful to the anonymous referees for their useful suggestions in making the presentation of this paper clearer. We would also like to thank J-P. Routeau for help on the implementation as well as V. Issarny and P.A. Lee from the University of Newcastle for helpful discussions on the content of this paper.

References

[1] Accetta (M.), Baron (R.), Bolosky (W.), Golub (D.), Rashid (R.), Tevanian (A.), Young (M.), "Mach: a New Kernel Foundation for Unix Development", in *Proceedings of the Usenix 1986 Summer Conference*, pages 93-112, July 1986.

[2] Anderson (T.E.), *The Case for Application Specific Operating Systems*, Technical Report no. UCB/CSD 93/738, University of California Berkely, 1992.

[3] Bershad (B.N.), Anderson (T.E.), Lazowska (E.D.), Levy (H.E.), "Lightweight Remote Procedure Call", in *ACM Transactions on Computing Systems*, vol 8 (1), February 1990, pages 37-55.

[4] Bershad (B.N.), Chambers (C.), Eggers (S.) *et al.*, "SPIN: An Extensible Microkernel for Application-specific Operating System Services", in *Proceedings of the 6th ACM Sigops Workshop on Matching Operating Systems to Application's Needs*, Warden, Germany, September 1994.

[5] Black (D.L.), "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", in *IEEE Computer*, 25 (3), May 1990, pages 35-43.

[6] Chase (J.S.), Levy (H.E.), Feely (M.J.), Lazowska (E.D.), "Sharing and Addressing in a Single Address Space System", in *ACM Transactions on Computing Systems*, vol 12 (3), November 1994.

[7] Ford (B.), Lepreau (J.), "Evolving Mach 3.0 to a Migrating Thread Model", in *Proceedings of the Usenix Winter Conference*, California, United States, January 1994, pages 97-114.

[8] Hamilton (G.), Kougioris (P.), "The Spring Nucleus: a Micro-kernel for objects", in *Proceedings of the Usenix Summer Conference*, Ohio, United States, June 1994, pages 147-159.

[9] Koldinger (E.J.), Chase (J.S.), Eggers (S.J.), "Architectural Support for Single Address Space Operating Systems", in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, United States, October 1992, pages 175-186.

[10] Levy (H.) *Capability-Based Computer Systems*, Digital Press, Mass. 1984.

[11] Liedkte (J.), "Improving IPC by Kernel Design", in *Proceedings of the 14th ACM Symposium on Operating System Principles*, North Carolina, United States, December 1993, pages 203-215.

[12] Mullender (S.J.), van Rossum (G.), Tanenbaum (A.S.), van Renesse (R.), van Staveren (H.), "Amoeba: A Distributed Operating System for the 1990s", in *IEEE Computer*, May 1990, pages 44-53.

[13] Rozier (M.), Abrossimov (V.), Armand (F.), Boule (I.), Gien (M.), Guillemont (M.), Herrman (F.), Léonard (P.), Langlois (S.), Neuhauser (W.), "The Chorus Distributed Operating System", in *Computing Systems*, vol 1 (4), pages 239-253, October 1991.

[14] *Driver Development Topics*, The SunOS Operating System Reference Manuel.

[15] Thekkath (C.A.), Levy (H.E.), "Low-Latency Communication on High Speed Networks", in *ACM Transactions on Computing Systems*, vol 11 (2), May 1993, pages 179-203.

[16] Wahbe (J.), Lucco (S.), Anderson (T.E.), Graham (S.L.), "Efficient Software-Based Fault Isolation", in *Proceedings of the 14th ACM Symposium on Operating System Principles*, North Carolina, United States, December 1993, pages 203-215.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399