# Extending Mach NORMA IPC and XMM to SMP Nodes

Eric PAIRE

OSF Research Institute
2, Avenue de Vignate
38610 Gières
FRANCE
Tel.: (+33) 76 63 48 92
Email: paire@gr.osf.org

October 1993

Abstract

This paper presents the design of the parallelization of the NORMA Mach3.0 XMM (eXtended Memory Management) and IPC (Inter-Process Communication) components, and describes its implementation. The purpose of this work was to modify the NORMA code so that it could run on a multiprocessor node. This includes not only the work to make it safe on a multiprocessor architecture (MP-safe design), but also modifications so that the code can run in parallel on different processors (MP-parallel design), leading to an efficient utilization of a multiprocessor architecture.

## 1. INTRODUCTION

NORMA Mach3.0 is a special branch of the Mach 3.0 microkernel code. This branch has been set up in order to run Mach3.0 on NO Remote Memory Access architectures (such as multicomputers). In addition to the standard Mach3.0 components, the NORMA Mach3.0 microkernel provides two specific components called **NORMA IPC** and **NORMA XMM**.

NORMA IPC (Inter-Process Communication) is responsible for the transparent localization of ports among a set of nodes called a NORMA domain. When a thread sends a message to a port, NORMA Mach3.0 is able to deliver this message and to respect Mach IPC port semantics even if the owner of the port receive right is located on a remote node. NORMA XMM (eXternal Memory Management) is responsible for the data coherency of memory objects that are shared between multiple nodes. This component is pushed between the interfaces of the memory manager and the VM (Virtual Memory) kernel component.

Unlike standard Mach3.0 components, NORMA XMM and NORMA IPC are not MP-safe, which means that they are unable to run safely on a multiprocessor node. The purpose of this paper is to explain the strategies developed to provide MP-parallel code for NORMA XMM and MP-safe code for NORMA IPC.

Since the current NORMA XMM component is now sufficiently stable and robust, it was decided to make this code MP-parallel, rather than MP-safe. An MP-parallel component means that most parts of the code can run in parallel on different processors of the same node, in order to get the maximum efficiency from the hardware architecture. Although it is not MP-safe, the NORMA IPC component is about to be rewritten for many other reasons, which explains why only work needed to make it MP-safe has been done, with the hope that this experience will contribute to the development of an efficient MP-parallel NORMA IPC component.

# 2. MP-PARALLEL XMM SPECIFICATION

## 2.1 XMM stack layout

Figure 1 shows a diagram of the layers that constitute the XMM stack. For each memory object referenced in the kernel and initialized, a stack of XMM objects is created locally with links to the upper and lower layers. If the memory manager of a local object is remote, then only XMM objects concerning the SERVER and IMPORT layers are created locally; otherwise, only those concerning the SERVER, KSVM, MSVM, SPLIT and USER layers are created. If there are remote kernels using a locally managed memory object, then an instance of the XMM object for the KSVM and EXPORT layers is created for each of these remote kernels. At the MSVM level, a request object is created for each MSVM object on which work has to be done. In fact, the MSVM and KSVM layers are part of the same layer, called SVM (Shared Virtual Memory) which is the heart of XMM because it is responsible for distributed coherency.

## 2.2 General parallelism policy

On a uniprocessor, each layer reaches a stable synchronization point before calling the next procedure of its adjacent layers, since it may block. Therefore, it is forbidden to hold any XMM lock during a call to a procedure of another XMM layer.

The ordering of page work is managed at the MSVM layer by a list of requests, each concerning the same page, and split into lists of kernel and pager ones. It seems obvious that such parallelism has to be maintained in the multiprocessor design. An XMM object reference count already present in the uniprocessor design will insure that no XMM object is destroyed while being used. In this way, each XMM object will have its own lock, at least for protecting its own reference count access. It will also be utilized to protect access to some other XMM object fields, depending on the layer class.

Since each request represents what has to be done on **one** page of a memory

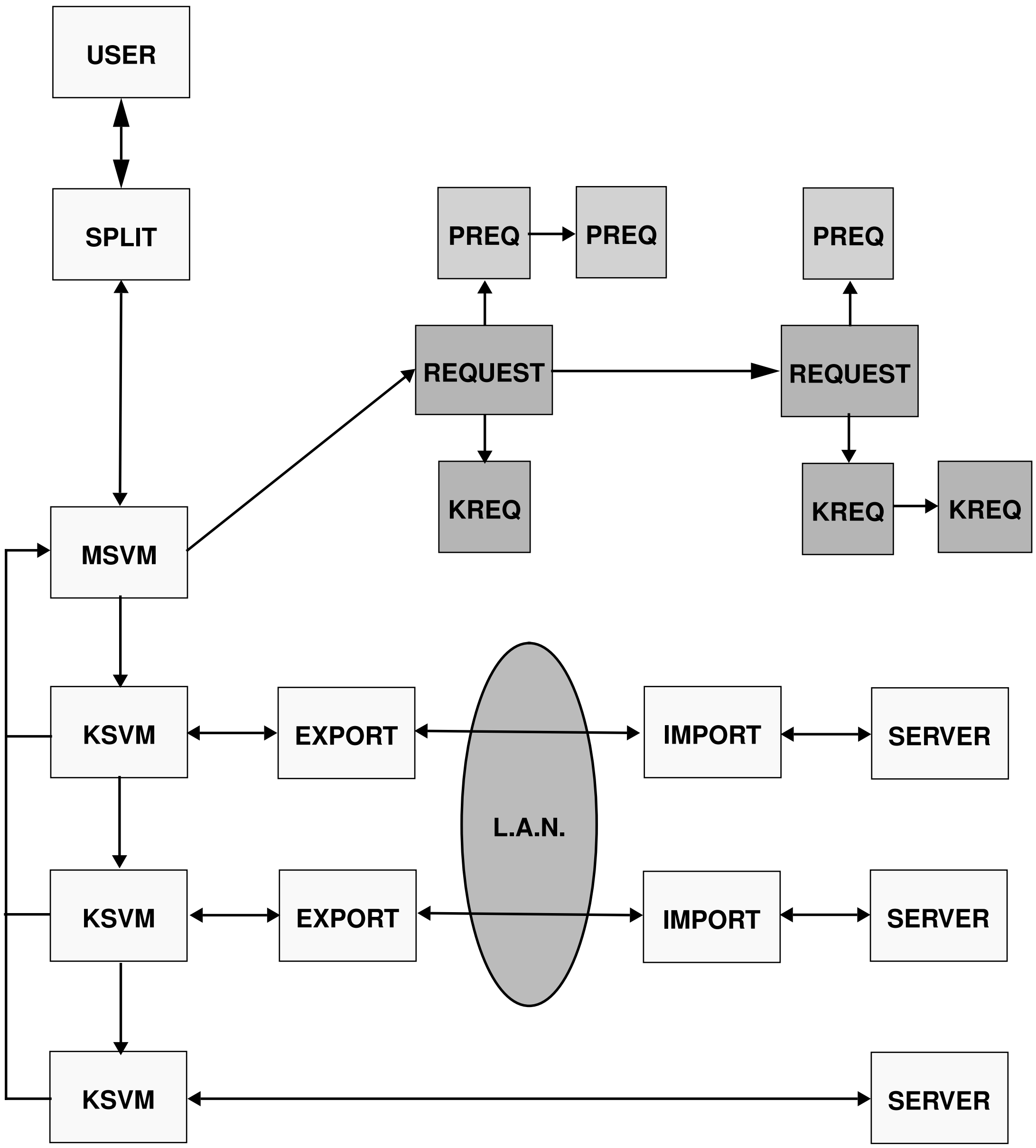


**Figure 1** XMM layers graph

object, it seems very fruitful on a multiprocessor to be able to handle different requests for the **same** XMM object in parallel. Therefore, when working on a request, a thread should not handle an XMM object lock for too long a period, in order to permit other threads to work on other pages (i.e. other requests) of the same object. Distributed servers should be able to take advantage of this strategy because a writable shared memory object will issue many requests to its memory manager at page granularity.

At the request level, preserving the sequence of work is the most important aspect. On a uniprocessor, a flag is set when a thread is currently managing a kernel or pager request. Even if this thread is blocked, no other thread will be able to do any work for this particular page. To protect the request state, a lock has to be introduced within each request structure. This lock will insure that no thread will modify the request state until the current thread has finished working with it.

## 2.3 Lock ordering

Since each layer is entered without any lock held, no lock ordering needs to be defined at the XMM layer level. At the MSVM layer, the request lock has to be held while handling the lock that protects access to the MSVM request list. Since the MSVM object lock has to be released while working for a given request, a deadlock may occur if the thread which handles the current request lock needs to take the MSVM object lock. It is clear that MSVM objects need a lock for the protection of their structure fields (the generic XMM object lock) and another lock for the protection of their request list. This lock has to be taken before any request lock, and the MSVM object lock will be taken after the request lock.

Each layer has in fact one and only one XMM object associated with a given VM object, except at the SVM layer. Each object is called in a different way, depending on whether it is used when going through layers from the VM component to the memory manager (called M object), or from the memory manager to the VM component (called K object). At the SVM layer, for a given VM object managed by a memory manager, there is one M object and as many K objects as existing kernels that have access to the pages of the original VM object. Most of the time, an M object is accessed before any K object, and this is the reason why the MSVM object lock will be taken before the KSVM object lock.

At the MSVM layer, there is another lock ordering difficulty to overcome. As for VM objects, M objects are part of a (sometimes doubly) linked list, the shadow list in one direction, the copy list in the other. If it has a shadow, an M object always holds a reference on it. So it is safe to lock the M object, and then its shadow. Unfortunately, accessing the shadow from a given M object occurs only once, when the M object has to be terminated. However, accessing the copy object of a current M object is much more frequent. This situation may lead to a deadlock if there is not a correct and efficient lock strategy.

The correct order is to first take the M object lock, and then its copy lock (i.e. the opposite of what is implemented in the VM component). The only conflict that may arise is between an M object that is being terminated, and the creation of a copy for its shadow. This is resolved by the fact that, before releasing the reference of its shadow, the terminating object is marked as terminated. Consequently, during the creation of a copy for a given object, if the copy exists and is marked as terminated, then the current terminating copy is just being ignored.

The final instance of lock order identified at the USER and SERVER layers defines the way to take locks when a port is translated to an XMM object, or reciprocally. The port lock is always taken first, and then the associated XMM object lock if there is one, so that it is always easy to dismantle safely the association between a port and an XMM object, by first taking the port lock and then nullifying the XMM associated object without taking the XMM object lock.

## 2.4 Termination mechanism

Locking each level independently of the others leads to a termination race condition. Requests from the kernel and from the memory manager are held completely independently at any layer, except at the MSVM layer where they are synchronized. When the kernel wants to release an XMM object stack, it starts by calling `memory_object_terminate()` that releases all layers between the SERVER and KSVM ones. Then, if there are no more kernels that use this object, an `M_TERMINATE(FALSE)` operation is issued by the MSVM layer to the USER layer. The USER layer is then responsible for refusing any operation initiated by the memory manager. Finally, when the MSVM layer wants to dismantle the upper XMM object stack, it issues an `M_TERMINATE(TRUE)` call to the USER layer that will unreference USER, SPLIT and MSVM layer objects in order to free them.

However, there is a problem if a request from the memory manager is issued from the USER layer to the MSVM layer before a `M_TERMINATE(FALSE)` call has arrived at the USER layer, and if the `M_TERMINATE(TRUE)` call arrives before this request has returned from lower layers. The XMM object stack cannot be destroyed since another request still uses it. A general synchronization mechanism has to be set up to solve this problem.

The mechanism must insure that XMM objects between the USER and MSVM layers will be released only when no memory manager request is being executed in the XMM stack. The obvious solution is to block the `M_TERMINATE(TRUE)` call at the USER layer until the last memory object request returns from lower layers; this solution has not been adopted because sometimes, the thread that issues the memory object request also issues the `M_TERMINATE(TRUE)` call, thus leading to a deadlock.

The solution implemented is the following: a memory manager ‘operation in progress’ counter is managed at the USER layer, incremented each time an operation is started from the USER layer to the MSVM layer, and decremented each time it returns, except for the `K_RELEASE_ALL` operation. At the USER layer, when a `M_TERMINATE(TRUE)` operation arrives, the memory manager is notified by a call to `memory_object_terminate()`, and if there are pending memory manager operations (the counter is not zero), the object is marked as released and the operation will return a `KERN_FAILURE` error, thus preventing the XMM stack dismantling to take place. At the end of any memory manager operation, if the USER object is marked as released and if the memory_manager operation counter decrements to zero, then the XMM stack has to be destroyed. This is achieved by issuing a `K_RELEASE_ALL` operation from the USER layer to the MSVM layer.

## 2.5 Global XMM locks

Within the SERVER layer, there is a timeout function for handling temporarily cached XMM objects. Upon a timeout, a thread regularly checks if there are such objects to remove asynchronously. There is a global queue that needs to be protected from simultaneous access by the normal thread and by the timeout thread. So a new global lock has been defined to protect access to this queue.

There are also some statistics counters that need to be protected. However, they used only for some internal statistics without any relationships to XMM functionalities. They will not be protected, because it is assumed that strict accuracy of these counters is not very important.

## 2.6 KSVM objects list protection

Within the SVM layer, it happens that some K_ operations have to be issued for each KSVM object of a given MSVM object, without any lock held since these operations call a procedure from another XMM layer. The MSVM and KSVM object locks must be released, so that no other thread may modify the list, or worse remove the current KSVM object. The most noticeable feature concerning the MSVM list of the KSVM objects is that list element insertion takes always place at the first or second position in the list, deletion occurring anywhere in the list.

The protection of the list must insure that the current element will not be released during the time a thread is working on it. Since new elements may be added just after the first one in the KSVM list, a possible optimization could be to skip those elements that were not part of the list when its processing began. Both have been achieved by writing some macros to hide the KSVM object list management.

# 3. MP-PARALLEL XMM IMPLEMENTATION NOTES

## 3.1 General considerations

The same code with minimum modifications should be compilable for uniprocessor and multiprocessor machines. There should be the least possible number of "`#ifdef`" statements in the code. For this reason, locks are accessed via the following macros that are defined differently for uniprocessor and multiprocessor machines (... stands for 'type of lock', and `a` for 'pointer to the object'):

- **`xmm_..._lock_addr(a)`** : Address of the specified XMM lock.
- **`xmm_..._lock_held(a)`** : Is the specified XMM lock being held?
- **`xmm_..._lock_init(a)`** : Initialization of the specified XMM lock.
- **`xmm_..._lock(a)`** : Lock the specified XMM lock.
- **`xmm_..._unlock(a)`** : Unlock the specified XMM lock.

**`xmm_..._lock_held(a)`** should be used carefully. On a uniprocessor machine without the MACH_LDEBUG option, it always returns TRUE. Therefore, it should always be used as "assert(xmm_..._lock_held(a));". On a multi-processor machine, if the kernel is compiled without the MACH_LDEBUG option, it tests if the lock is taken or not; if the kernel has been compiled with the MACH_LDEBUG option, then it will also test if the owner of the lock is the current thread. This macro usage is to specify assertions at the beginning of functions, in order to show which locks should be held at their call time.

## 3.2 XMM object modification

Since all XMM objects begin with the same "struct xmm_obj" generic structure, XMM object simple lock is added to it, so that with some minor #include modifications, all XMM objects have their own simple lock declared. This simple lock is initialized directly in the "xmm_obj_allocate()" function with the other object fields. Its type is obj, and the lock macro called "xmm_obj_lock(xmm_obj_t)".

## 3.3 MSVM object additional modification

MSVM objects are the only XMM objects that need some internal modifications in order to correctly manage parallelism for request management. As explained above, the modification consists of the addition of a local simple lock to protect the request queue against concurrent accesses. The type of this simple lock is list, and the lock macro called "xmm_list_lock(xmm_obj_t)".

## 3.4 REQUEST modifications

REQUEST structures also needs some modifications to achieve serialization of many kernel or pager requests on the same page of the same MSVM object. As before, a new local simple lock is added to each request structure. The type of this lock is req, and the lock macro called "xmm_req_lock(request_t)".

There is another modification that needs further explanations: when working for a request, a thread may eventually want to destroy it. For this purpose, it will call "xmm_release_request(request_t)" with the req lock taken. Unfortunately, it cannot take the list lock, since the correct order is first the list lock, and then the req lock. To respect the correct lock order, it has to first release the req lock, and in a second phase, take first the list lock and then the req one. However, during this time, another thread may have put in this particular request another kernel or pager request. Additional code is required to handle this situation, as described below.

In "svm_release_request(request_t)" after having got back the req lock, additional code must be added to test if the request is still to be destroyed. If it is not the case, then the thread must release the req and list locks and return without having destroyed the request; another thread has provided a new life to the current request and its deletion will be deferred. This is the first piece of code to add to this function.

Unfortunately, a more subtle case may occur: what happens if the second thread enters “`svm_release_request(request_t)`” while the first one is still waiting to take back the `req` lock? Each of them will find the request ready to be destroyed, and the “`struct request`” will be released twice.

It is mandatory to add a new flag in the “`struct request`” type in order to mark that a thread is currently destroying the request. This flag will be tested and set in “`svm_release_request(request_t)`” before the first release of the `req` lock. If it is found to be already set, then another thread is about to destroy the current request. So, it is not worthwhile for the current thread to continue the destruction process, since someone else is already doing it, and this corrects the last problem. This flag is useless on a uniprocessor, since the thread that executes “`svm_release_request(request_t)`” will not be preempted. However, it is important to keep it, in order to have the same code for all architectures.

### 3.5 Timeout lock

The only global simple lock that has been introduced for XMM is the one that protects the object caching queue against concurrent accesses from normal threads and the timeout thread. The type of this lock is `queue` and the `lock` macro is called “`xmm_queue_lock(lock_t)`”. When managing the XMM object caching queue, functions will have to take the XMM object lock. The lock ordering is quite obvious here: first the `queue` lock, and then the `obj` lock.

For kernels subsequent to NMK15, it is no longer necessary to mask procedures that handle locks against other calls to “`timeout()`”, since timeout functions are now handled by a special thread, and not directly under hardware clock interrupt.

## 4. MP-SAFE NORMA IPC DESCRIPTION

### 4.1 IPC internal structure

NORMA IPC is split into two main parts: the distributed IPC semantics layer and the transport protocol layer. Unfortunately, these layers are not independent of each other, so it is not easy to set up an MP architecture as for XMM. An alternative approach is to examine procedures and try to discover under which context they are executed. This approach leads to the following analysis:

- **Hardware Interrupt:** NORMA packets are received by a network/mesh driver under hardware interrupt. In most cases, this context delivers packets up to the complete or incomplete message receive queue. The other case where a hardware interrupt is received is at the end of packet transmission. Under this context, NORMA IPC code looks for other fragments to be transmitted, either from the same message or from a new message waiting to be sent.

- **Asynchronous Trap:** A netipc asynchronous trap context is responsible for managing complete and incomplete message receive queues. If the destination port is in the kernel space, then the message is put in the kserver queues handled by the kserver threads. Otherwise, the message is delivered to the local IPC module.
- **Retransmission Timeout:** Under this timeout context, NORMA code retransmits packets for which no protocol acknowledge has been received in the assigned time.
- **Resume Timeout:** Under this timeout context, the stop/resume protocol regularly sends packets asking the receiver if it is ready to receive more packets from the local node.
- **netipc_slow_page thread:** If the amount of out-of-line memory of the current packet is too large to fit into a vm_map_copy structure, then the pages must be gathered into a special VM object. Since the VM object path may block, it is mandatory to use a thread context to run this code, whereas the hardware interrupt context is sufficient within the vm_map_copy path.
- **netipc thread:** Most of the time, it is forbidden to block on packet reception contexts. The role of the netipc thread is to refill asynchronously the NORMA input and output resources pools.
- **kserver threads:** When the destination port is in the kernel port space, a message is delivered to a queue managed by a pool of threads. Their context is used to do the in-kernel work requested by the message.
- **kserver_pageout threads:** In the case of kernel pageout resources, a special kserver path has been set up with its own kserver_pageout threads to manage critical resources, instead of going through ordinary kserver threads.
- **netipc_vm_map_copy thread:** This context is used to send out-of-line pages that have been put in vm_map_copy objects.
- **user threads:** NORMA procedures also use sending/receiving thread contexts. Under sending ones, a message is put in the send queue and sending occurs if the NORMA code is not currently in this state. Under receiving ones, threads are waiting for a message to arrive, except if one of them is running inside `norma_ipc_kmsg_accept()`, where it is spinning for the reception of a packet, instead of being idle, thus saving one context switch on NORMA packet reception.

The uniprocessor synchronization between all these contexts is achieved by the global variable **`netipc_lock_owner`**, which contains the context that is currently being run, and by the fact that most of the code is run completely masked against hardware interruptions. The greatest difficulty to overcome for setting an MP-safe design will come from the fact that the same parts of the code may be run either under thread context, timeout context, asynchronous trap context or even under hardware interrupt context.

## 4.2 NORMA IPC MP-safe general architecture

As explained in the introduction, the goal here is to set up an MP-safe NORMA IPC which is as efficient as possible, so that NORMA IPC can be used on MP architectures with some parallelism. The general idea is to have an interrupt thread that will serialize all asynchronous events such as hardware interrupts and time-outs, so that the ordering required by the current NORMA IPC implementation is preserved, and most of the code will run without being masked. This interrupt thread must not block (otherwise no more packets will be received from the network nor any timeout handled), and it will run different procedures depending on two counters (number of packets received or sent) and on two flags (timeout or resume_timeout request). There is a simple lock to synchronize counter/flag accesses, to be handled while masked at `splimp()` level.

One of the major problems in the MP-safe NORMA IPC work is to add missing port locks and to define the relationships between port locks and NORMA IPC locks. The natural order is to first take the port lock, and then the NORMA IPC lock, whatever it is. This new lock ordering implies two types of changes in the code: new simple locks must be added to solve potential deadlocks, and some algorithms must be changed. These will be explained in separate sections.

In the standard NORMA IPC implementation, it is impossible to receive more than one packet at a time from remote nodes until the current packet has been acknowledged or discarded. One important improvement that has been introduced in the MP-safe version is the ability to receive one packet from more than one remote node at a time, in order to parallelize packet reception from different nodes. The structures used, and problems encountered for this enhancement will also be explained in a separate section.

Finally, the global variable `netipc_lock_owner` has been replaced by a simple lock `netipc_lock` that keeps on playing the same role in the NORMA component general synchronization, but is no longer handled while being masked.

## 4.3 NORMA IPC context description

Figure 2 shows the relationships between all NORMA contexts potentially used for message reception, and all parallel work that could be done at the NORMA level. The new interrupt thread adds one more context switch. The asynchronous trap utilization has been kept as a NORMA IPC packet reception flow control. It will be run only if there are no more kernel threads to run, that is input packet processing will be delayed until there is no more kernel work.

## 4.4 NORMA IPC new port management

The main part of NORMA IPC that has been changed for MP-safe purposes is

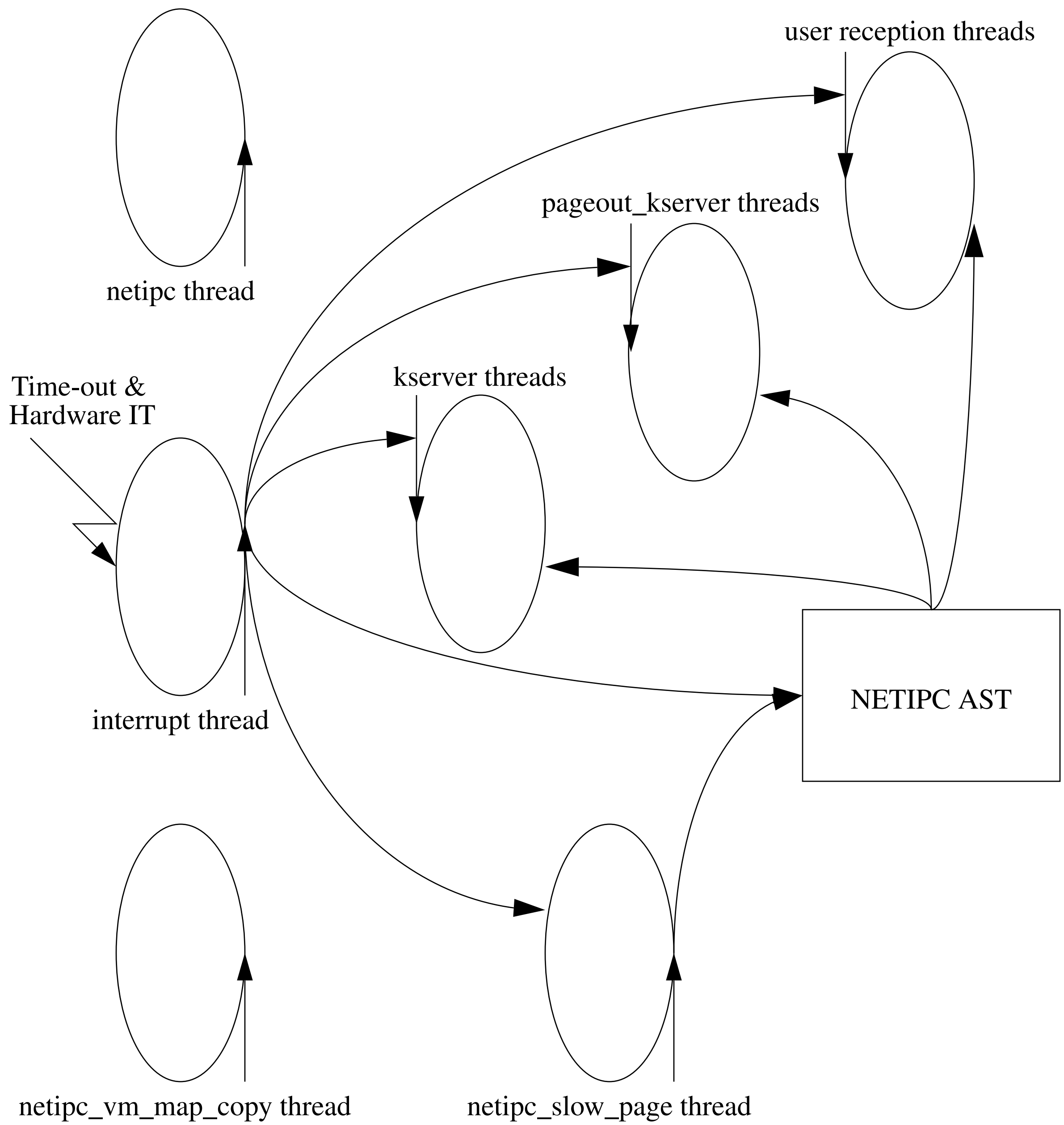


**Figure 2** NORMA IPC contexts

the NORMA port management. There were two problems: first is the fact that a lookup may be followed by an insert, in which case other lookups for insertion should be blocked. Second is the lock order to respect between `netipc_lock` and the port lock. Both have been addressed the same way: NORMA ports are hashed and all ports that have the same hash value are put in a list whose root is `norma_port_table[]`. This array element is no longer a pointer to a port, but a structure that contains four fields: the pointer to the port list, a flag that insure the lookup and insert sequence mutual exclusion, a counter of the number of threads waiting for a lookup and insert sequence, and a simple lock that protects accesses to the structure fields.

NORMA port management is no longer protected by `netipc_lock`, but by a per hash value list lock, which improves potential parallelism, and allows the standard lock ordering to be respected: first take the port lock, and then the NORMA lock. The `norma_port_lookup_locked()` procedure has been replaced by a new general `norma_port_lookup()` procedure with a new argument to indicate if it is a lookup and insert sequence start point. If not, then the lookup will not block; otherwise, the thread will be blocked until no other thread is currently running a lookup and insert sequence. It the port already exists in the NORMA port list, the thread returns the port. If the port does not exist, then the thread will mark the port list blocked for any subsequent lookup and insert sequence, and return a NULL port. The procedure that initiated this lookup and insert sequence is then responsible to call `norma_port_insert()` to insert the port for which the lookup failed, and to unblock any other thread waiting to start a lookup and insert sequence.

Of course, since the new lookup and insert sequence may block, such a sequence must not be run under the interrupt thread or AST contexts.

## 4.5 NORMA IPC new reception architecture

As mentioned before, the goal of this architecture is to add some more parallelism in the receive strategy of NORMA packets. The idea is to have a list of receive contexts that is searched at each new received fragment, with a status associated to each context.

- If there is already a complete packet from the same remote node, then throw away the current fragment (current packet has not yet been managed).
- If there is no context from the remote node, then allocate (if possible) a new context to store the received fragment. If the packet is completely received, then mark it and wake up the interrupt thread for further packet handling.
- If there is a context holding an incomplete packet (mostly from a medium that sometimes fragments packets such as Ethernet) and if the current fragment is the next one, then add the current fragment to the incomplete packet; if the packet is completely received, then mark it and wake up the interrupt thread for further packet handling.
- Otherwise, this is an error and the current fragment will be discarded, as well as any other context that matched that of the fragment.

Any completely received NORMA packet will be removed from the received list either because internal information was wrong, or when it has been acknowledged. There are three types of NORMA packets: IPC messages, out-of-line pages and control packets. IPC messages and ool pages are acknowledged by a control packet, while control packets are not. Each packet always starts with a header of type `struct netipc_hdr`, followed by data. This data is just one longword for control packets, whereas for other types, it could be up to a page in length. There will also be a simple lock inside each context, to protect it against concurrent accesses, as well as a global simple lock to protect context list management. It is obvious that these locks have to be managed under `splimp()` mask.

A simple memory usage optimization is to allocate a page with a context only when receiving an IPC message of an ool page, and to reserve a longword in the context to receive control packets. This optimization is particularly useful under certain circumstances: if a local node receives two packets one after the other from the same remote node, then the second one will be thrown away if the local node has not got enough time to manage the first one. This is particularly undesirable if the first one is a control packet (such as an acknowledge), and the second an IPC message or an ool page, because the control packet will be quickly managed by the local node without consuming a lot of resources, and because the next packet will be retransmitted after a timeout (as in a NULL rpc). Latency may become very high with such a mechanism.

For this reason, a small window has been opened at the reception level: when receiving a fragment from a remote node, if there are contexts from the same remote node only holding control packets, then the received fragment will not be thrown away; it will be kept in a separate context (if it is possible to allocate such), but managed after previous ones by the interrupt thread. However, if the current fragment is the first one from an IPC message or from an ool page, and if there is already a context that holds such a fragment type, then the current fragment will be released. This optimization does not kill the 'one packet window size' paradigm of NORMA IPC, and should decrease latency and increase throughput between two nodes exchanging packets, since it may reduce loss of exchanged fragments.

Concerning resources, this optimization allocates contexts and pages, but within a limit that depends on the total number of nodes. There will always be at least one allocated context and one reserved page able to receive fragments continuously from other nodes; another advantage is the ability to receive and manage more control packets under memory shortage, since control packets just need a context structure and no extra page to be managed.

## 4.6 NORMA IPC new simple locks

`norma_ipc_kmsg_accept()` is a procedure that spins waiting for further work to do, or for a NORMA message reception. The idea is to replace the idle thread, in order to save a thread context if a NORMA message is received. In the current implementation, only one processor is allowed to run this procedure. One possible improvement would be to allow more than one processor to spin inside this special procedure. In any case, a new simple lock was required here, because all `norma_ipc_handoff` variables were protected by `netipc_lock`, and there was a potential deadlock between it and the message queue lock.

Pages may now be allocated under hardware interrupt at the same time as contexts, so that it becomes mandatory to handle the `netipc_page_list_lock` under `splimp()`. The main consequence of this allocation is that a new simple lock `netipc_vm_page_list_lock` has to be created to specifically protect `netipc_vm_map_copy` structures, and this simple lock does not need to be handled under `splimp()`.

## 5. RESULTS AND PERSPECTIVES

The first result was to correct many MP-independent bugs hidden in the uniprocessor architecture, and to detect areas where work had to be done to fit MP constraints. Work was started on the MP-parallel XMM component, because it was easy to test it on a multiprocessor single node NORMA domain. The second phase was the introduction of the MP-safe NORMA IPC component. All this work has been integrated into NMK16 kernel base and has been tested with different servers (OSF/1 AD, OSF/1 MK and OSF/1 1.3 from the OSF engineering team), on different systems (PC/AT, 2 processor COMPAQ/PRO, 4 processor DEC433 and 16 processor Sequent).

At the time of writing, no real performance work has been done on these new NORMA components. A performance tool is being written, but is not yet MP-oriented. A very interesting study to be done is to profile this MP kernel to find bottlenecks, and also to enable lock monitoring to measure lock usage and contention.