Exploiting Large Memory using 32-bit Energy-Efficient Manycore Architectures

Mohamed Lamine Karaoui, Pierre-Yves Péneau, Quentin L. Meunier, Franck Wajsbürt, Alain Greiner

Exploiting Large Memory using 32-bit Energy-Efficient Manycore Architectures

Mohamed L. Karaoui, Pierre-Yves Péneau*, Quentin Meunier, Franck Wajsbürt, Alain Greiner
Sorbonne Universités, UPMC Univ Paris 06, UMR 7606
Laboratoire d'Informatique de Paris 6
F-75005, Paris, France
firstname.lastname@lip6.fr

***Abstract*—Recent advances in processor manufacturing has led to integrating tens of cores in a single chip and promise to integrate many more with the so-called manycore architectures. Manycore architectures usually integrate many small power efficient cores, which can be 32-bit cores in order to maximize the performance per Watt ratio. Providing large physical memory (e.g. 1 TB) to such architectures thus requires extending the physical address space (e.g. to 40 bits).**

This extended physical space has early been identified as a problem for 32-bit operating systems as they can normally support at maximum 4 GB of physical space, and up to 64 GB with memory extension techniques.

This paper presents a scalable solution which efficiently manages large physical memory. The proposed solution decomposes the kernel into multiple units, each running in its own space, without the virtual memory mechanism (directly in physical mode). User applications, however, continue to run in the virtual mode. This solution allows the kernel to manage a large physical memory, while allowing user space applications to access nearly 4 GB of virtual space. It has been successfully implemented in ALMOS, a UNIX-like operating system, running on the TSAR manycore architecture, which is a 32-bit virtual 40-bit physical manycore architecture. Moreover, the first results show that this approach improves both the scalability and the performance of the system.

I. Introduction

Moore's law suggests that the number of transistors per core doubles approximately every two years. Until the early 2000s, this increase has been used by designers to improve the performance of microprocessors by increasing the instruction level parallelism (ILP) along with increasing frequency. However, both the ILP and frequency improvements have reached a limit. The first limit is due to weak ILP in many commercial applications [6], and the second due to the power dissipation problem [8]. To overcome this limit and keep improving the performance of processors, manufacturers integrate multiple cores into a single processor rather than trying to improve a single core. Currently, systems containing tens of cores are a reality and this trend is expected to continue in order to meet the performance demands. This kind of processors are called manycore and could contain hundreds if not thousands of cores. To keep the power consumption within an affordable envelope, the cores need to be smaller and simpler as the number of cores is increased [8].

*Pierre-Yves Péneau is now with the Montpellier Laboratory of Informatics, Robotics and Microelectronics (LIRMM) at Montpellier, France

It is in this context that the TSAR (Tera-Scale ARchitecture) architecture has been developed [11]. This manycore architecture was designed to integrate up to 1024 cores. It supports a CC-NUMA (cache coherent - non uniform memory access) paradigm since its memory is physically distributed on the chip, and logically shared: any core can transparently access any memory location, although at different costs. To minimize the power consumption, TSAR uses small single-issue 32-bit MIPS cores. To avoid limiting the supported memory to 4 GB, the physical address space is extended to 40 bits, which allows the architecture to support up to 1 TB of memory.

In addition to the energy and space efficiency, the use of 32-bit cores has two main advantages over 64-bit cores. First, the application memory footprint is minimized, as most data types are limited to 32-bit: on 64-bit architectures, more data types are 64-bit wide, like pointers. As a consequence of this first advantage, the second is that the cache utilization is improved. The main disadvantage, of course, is the space limitation for the user applications.

The use of 32-bit cores with an extended address space is not new to the processor industry. It has been used in many x86 processors and in some ARM processors. However, current operating systems cannot efficiently support physical address spaces larger than 64 GB. In practice, this limit is even smaller to avoid memory pressure on kernel data structures (see section III). Moreover, current operating systems offer at maximum 3 GB of virtual address space to applications although the cores can address up to 4 GB.

To overcome these limits and allow the operating system to support large extended physical memory while offering nearly 4 GB of address space to user applications, we designed a new solution. This solution is based on two principles. The first consists of structuring the kernel as a distributed system, composed of several cooperating kernel units, each one managing a private segment of the total physical address space. The different kernel units cooperate to present a single kernel image to user applications. This structuring strategy is not new since it has been used to improve the scalability [18], [16], to contain hardware and software faults [10], or to support heterogeneous architectures [7], [5], [14], yet the motivation here is new: to support a large physical address space. The second consists of executing each kernel unit in physical mode, while running user applications in virtual

mode.

In the rest of the paper, section II presents some background; section III presents related work; section IV presents the TSAR architecture with its features that fully address the physical space; section V presents our solution; section VI presents the experimental evaluations; finally, section VII concludes and presents future work.

II. Background

In this section we present the hardware and software components used to manage memory spaces.

A. Physical Page Descriptor

Most modern operating systems manage the physical memory with the granularity of a page. The size of a page depends on both the operating system and the hardware. However, most 32-bit operating systems use a 4KB page size. To manage these pages, the kernel represents each physical page with a page descriptor data structure. This descriptor holds much information, like the address of the page and the number of its users. The size of the descriptor depends on the operating system, but is typically 32 Bytes.

B. Process Virtual Address Space on 32-bit Architectures

Existing operating systems split the address space in two parts. The first part is used to map user data, while the second part maps kernel data. The size of each part depends, generally, only on the operating system. We find two common configurations: the first is to give half of the virtual address space (2 GB) to the user and the other half to the kernel; the second is to give 3 GB to the user and 1 GB to the kernel. The problem with this split configuration is that it limits the address space for both the user and the kernel to 2 (resp. 3) GB and 2 (resp. 1) GB.

C. Memory Manager Unit

Abbreviated to *MMU*, this hardware component translates virtual addresses to physical addresses. The necessary information for these translations are extracted from software initialized table, called *page table*. To avoid accessing memory at each translation (i.e. at each load/store instruction), the MMU caches most recently used entries of the page table in a small cache called *TLB* (Translation Look-aside Buffer).

On some architectures, the MMU does not access the memory page table if an entry is not in the TLB. It simply raises an exception and it is up to the operating system to access the page table in memory and update the TLB cache. These are called *software-managed* TLBs. Analogously, MMUs that fill the cache transparently to the software are called *hardware-managed* TLBs.

In each context switch the TLB is flushed. This is necessary to avoid reusing entries of the previous context (process) that stayed in the TLB. However, an optimization can be used to avoid these costly flushes. It consists of assigning to each context a unique identifier, called *ASID* (Address Space IDentifier). This identifier is stored in the TLB entries each time they are filled. Thus, the TLB can distinguish between entries in different contexts, avoiding the need to flush the TLB. There is also another optimization which avoids flushing entries that are shared between contexts. This is generally the case for entries that are part of the kernel virtual address space since all contexts share the same kernel. This functionality is generally implemented by adding a bit called the *global bit* in the page table entries, which is later cached by the TLB, that marks the entry as shared. Thus, the TLB can avoid the unnecessary flushing of such entries.

III. Related Work

A. Processors with Extended Physical Space

There are three brands of industrial 32-bit processor which use an extended address space. We find this functionality in the `SPARC V8` [17] processor and in many `x86` processors, like the `Pentium Pro` processors [13]. In these processors the physical address space is extended to 36-bit, allowing up to 64 GB of memory. The extra physical space is accessible (only) through the virtual memory. In the x86 world, this functionality is called PAE for Physical Address Extension. We also find this functionality in the `ARM-v7` architecture profile [4]. The address space is extended to 40-bit (similar to TSAR). This functionality is called LPAE for Large PAE.

B. Software Support for Extended Physical Space

1) Dynamic and Static Mapping: To support the extended address space, modern monolithic operating systems use a *dynamic mapping* technique. It consists of reserving part of the kernel virtual space to dynamically map the data in the extended address space. Analogously, the other part, permanently mapped, of the virtual address space has a *static mapping*.

This solution has two limitations. First, the use of a dynamic mapping is only done on big data with no pointer, like the file system data cache. The second is that critical kernel structures which must stay in the static mapping become constrained by the available space for it, like process descriptors, file system metadata and more importantly physical page descriptors. The size of the latter increases with the size of the physical memory (there is one descriptor per page) while the size of the static mapping does not increase. For example, with 64 GB of physical space, the collective size of page descriptors will reach nearly 512 MB, which leaves little space for the rest of the kernel structures, supposing that the kernel space is 1 GB. The system is hardly functional with more physical space.

2) Different Spaces for the Kernel and the User: Another solution has been suggested in the Linux community to overcome the limitation of the previous solution [1]. This one consists of allocating different virtual spaces for the kernel and the user. This offers to both the kernel and the user a virtual space of 4 GB. The problem of this solution is that it requires switching address spaces at each system call, which is costly on architectures without the ASID functionality, since the TLB must be flushed before and after every system call.

Even if this cost is not considered, 4 GB of address space for the kernel is still small compared to 1 TB of memory.

3) Address Windowing Extension: To allow user applications to access more than their address space (2 or 3 GB), Windows defines the `AWE` (Address Windowing Extension) API [15]. This API allows the user application to implements a dynamic mapping strategy to access more memory than their virtual space. To use this API, a user application first allocates physical memory, then maps it into part of its virtual space. When the access to the data is no longer needed, the physical memory can be unmapped. In addition to its complexity, the default of this solution is that it requires modifying user applications.

Another solution for user applications to access more memory will be to use the file system. This solution consists of storing the data on a file. This data can be accessed either using the *read*/*write* system calls or the *mmap* system call. For the latter, the user application must use a dynamic mapping strategy. This is similar to the previous solution, but using a POSIX compliant system call.

4) Micro-kernel based operating system: We are unaware of any solution used to support extended physical space in a micro-kernel based operating system. Using the dynamic mapping technique in such operating system may be more costly as the modification of address space must be requested from the memory server through message passing.

IV. Tera-Scale Architecture (TSAR)

The TSAR [11] architecture is composed of clusters. Figure 1 presents this architecture with 4x4 clusters. Each cluster contains four cores with their private first-level cache (L1), a shared second level cache (L2) and an interrupt controller (XICU). Each L2 cache maps a different segment of the external memory (RAM). All segments have the same size. Outside the processor chip, we find the external peripherals. The main three are the TTY (text output), the IOC (block device) and the IOPIC (interrupt controller). This last peripheral transforms wired interrupts to memory writes. These memory writes are generally programed to point to mailboxes of the XICU. Mailboxes are XICU registers that, when written to, interrupt one of the cores of a cluster. Thus, the couples of the IOPIC and the XICU components can be used to transform an external peripheral wired interrupts to a core wired interrupt.

A. Physical Space Segmentation

The 40 bits of a physical address are split into two parts. The first part (bits 32 to 39) encodes the cluster number. Each cluster is represented by an (x,y) coordinate, x being the abscissa on the chip plane, and y being the ordinate. This part is called the *extended address*. The second part (bits 0 to 31) allows access to the components of a cluster (L2 or XICU). This part is called the *local address*. With this encoding, the physical space grows linearly with the number of clusters.

B. Means for Accessing the Physical Space

In TSAR, each core can access the 40-bit physical address space:

1) in virtual mode, using the MMU component. This mode can be used to translate 32-bit virtual addresses to 40-bit physical addresses. The used MMU component has a hardware managed TLB. To avoid flushing kernel entries between context switches, this MMU implements the global bit. Moreover, the coherence between the TLB content and the software defined page tables, is ensured by the hardware. This hardware coherence is absent in modern industrial processors, where it is up to the software to flush the TLB entries when the page table is modified.
2) in physical mode, using the *IEA*/*DEA* registers. When the MMU is deactivated, these 8-bit wide registers extend 32-bit physical addresses to 40-bit. The IEA register extends all addresses used to fetch instructions from memory. The DEA register extends all addresses of data load/store instructions. At reset, both registers are initialized to zero. Their content is not erased between two accesses, thus the same extension can be used for multiple accesses. The virtual memory can be deactivated for either the data load/store or for fetching instructions with no additional cost, in a single instruction.

   For the kernel to access data using this mode, it first needs to set the appropriate extended address in the DEA register, then executes the usual 32-bit load/store instructions. The processor will then extend all 32-bit addresses with the content of the DEA register, as long as the MMU is off for data accesses. A similar procedure may be used to fetch instructions using physical addresses.

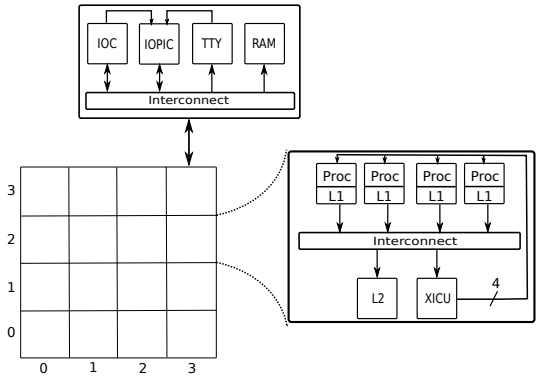

Fig. 1. A TSAR architecture with 4x4 clusters. The right block describes the content of a cluster. The top block describes some external peripherals.

V. Principles of the Solution

Our solution was implemented in ALMOS [2], [3], a UNIX-like operating system. This operating system was concurrently developed with the TSAR architecture to efficiently exploit manycore architectures.

Until recently, this operating system was developed in an early version of TSAR which had a 32-bit physical address space. With the full development of the TSAR architecture it was necessary to add support for the extended address space.

A. First Approach

To introduce support for 40 bits physical addresses, we first tried to use a solution inspired from commercial UNIX/Linux or Windows operating systems. We used the virtual mode, with static mapping to access the kernel code and some of the kernel data, while using physical mode to access kernel data that may not fit in the virtual space. In contrast with existing operating systems, our goal was to exploit the physical mode of TSAR to access not only simple data with no pointer such as the file system data cache but also complex data structures, such as the page descriptors.

However, implementing this solution was difficult. This is due to the fact that once a structure is placed in physical space, all the code handling this structure needs to be carefully analysed to use specific functions when dereferencing a physical pointer. These functions increase the cost for accessing the memory since they add to each physical access three additional instructions: two instructions to activate/deactivate the virtual memory and one to set the extended address in the DEA register. This problem is particularly difficult knowing that most accesses need to be done in physical mode since the virtual space represents only one thousandth of the physical space. For these reasons this solution was abandoned.

B. Proposed Solution

From the failure of the first solution, two guidelines were developed:

i) Only one address space must be used: the physical address space as it allows access to all the memory space; *ii)* The necessity to change the value of the extended registers must be kept minimal to avoid increasing the cost of memory accesses.

This solution restructures the kernel as a distributed system, where we have a complete independent kernel units in each cluster.

Each unit runs directly in physical mode. Both extended address registers are set to point to the local cluster: cores of cluster 0x00 have both extended address registers set to 0x00; cores of cluster 0x10 have both register set to 0x10, etc. (see Figure 1). Thus, all local accesses do not require changing the value of the extended registers.

Remote data accesses are done by temporally changing the value of the DEA register; there is no need to change the IEA register, since the kernel code is replicated in all the clusters. To further keep minimal remote accesses, almost all inter-cluster communications are done using a message passing service, implemented as RPCs (Remote Procedure Calls). Thus, all kernel subsystems are free of remote accesses. Only the message passing service uses them to post messages. However, for performance reasons, remote accesses are also used to move big data between clusters. This optimization avoids redundant copies of the same data. We call this optimization the *direct copy* functionality. It is for example used in file I/O operations (such as *read()*/*write()*) to copy the data of a file placed in the first cluster to the user buffer located in a second cluster.

This structuring allows the kernel to handle all the physical space of the TSAR architecture. Indeed, since each kernel unit handles the 4 GB of physical space local to a cluster, the aggregate space handled by all the units corresponds to the amount of physical space in the architecture.

Structuring a shared memory operating system as a distributed system is not new to the operating system world. We find many systems which use this type of structure with different motivations, like [18], [10], [7], [19], [5], [14], [16]. However, all past works sustain the scalability of this kernel structure. This is a second motivation for our solution since one of the most important goals of ALMOS is to ensure scalable performance.

Yet, we are not aware of any modern operating system which uses physical accesses at the kernel level while using virtual addresses for user space applications. This hybrid solution offers 4 GB of virtual address space to the user, with the exception of one page which is reserved for the kernel. This page maps the first instructions of the kernel entry code which deactivates/reactivates the virtual memory when entering/exiting the kernel code. Another case which must be handled is the transfer of data between the kernel and user space, which is necessary for system calls when the data cannot fit into the registers. To solve this problem, two solutions were considered. The first is to find the physical address of each page of the user buffer and then copy the data per page. The second is to activate temporarily, in the kernel, the data virtual space to access user data. Currently, we use both solutions. The first is used when a user buffer needs to be accessed directly from a remote cluster, typically for I/O file operations (see above paragraph). The second is used when a thread executing in the kernel needs to copy kernel data to its own user space buffer. A deeper study needs to be done to compare the two solutions.

Using physical addressing in the kernel presents two performance advantages. The first is that the kernel accesses perform better since no physical to virtual memory translation is necessary. The second is that user applicatons also perform better since the TLB is not shared with the kernel. The only real disadvantage that we found until now is the hardness to find kernel bugs which modify kernel code or read only data. When the kernel runs in virtual mode, this type of bug is easily detected since the virtual memory protection can be used to control the access type.

C. Implementation

Implementing the proposed solution was aided by the fact that ALMOS was internally organized in clusters. For example, all remote memory allocations must be explicitly requested, making it easy to track remote structures and thus remote memory accesses. However, restructuring the operating system required many small sparse changes in almost all subsystems of the kernel: the memory manager, the file systems and the process manager.

We describe the implementation by following the four main steps of the development process. For the purpose of this paper, some parts are briefly described.

1) Replicating Independent Kernels: This step consists of replicating, in each cluster, independent kernel units that do not communicate between them: memory cannot be remotely allocated, processes and threads cannot migrate across clusters, and the file system stack is replicated (this is safe since the file system is read-only, otherwise the disk content can become incoherent).

Running kernel units directly in physical space requires three main modifications, two of which have been discussed in section V : deactivating/reactivating the virtual mode when entering/exiting the kernel and the handling of data transfer between user and kernel spaces. The last modification is done in the kernel boot code which constructs the virtual space. This latter is modified to map only one page for entering/exiting the kernel, to deactivate the virtual mode, and to set the DEA and IEA registers to point to the local cluster. The rest of the kernel works as if the virtual memory was active and statically mapping the space of the local cluster.

Access to external peripherals is synchronized by hardware dependent code, which uses locks placed in the first cluster. The routing of the IOPIC interrupts to one of the cores is dynamically programed at each lock acquisition. This allows the interrupt to be treated locally to the core which requested a peripheral service.

2) Establishing Communication Channels: A static inter-cluster communication mechanism was established between cores using a per cluster circular buffer (Figure 2). These buffers are used only to receive message requests. The accesses to these buffers are synchronized with a lock-free mechanism, similar to [5]. The response messages are sent using a buffer allocated by the sender. Finally, all the message passing complexity, such as the pre-allocation of the response buffer, is encapsulated by an RPC mechanism.

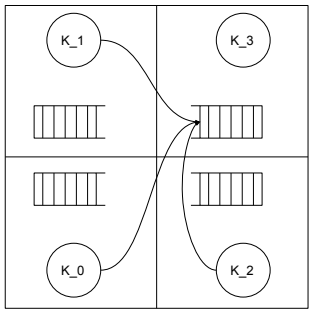

Fig. 2. An example of the new ALMOS structure on a 2x2 cluster architecture.

a) RPC Types: There are two types of RPC: blockable and non-blockable. Blockable RPCs can sleep, while non-blockable RPCs cannot sleep. The first type requires that the RPC must be treated in a separate context (i.e. by a dedicated thread), in order to avoid unnecessary halting of user or kernel threads. The second type can be executed in any context, which avoids the cost of the context switch. Currently, all RPCs are treated as blockable. Eventually, it will be better to treat non-blockable RPCs separately, thus saving the cost of a context switch.

b) Notification Mechanism: In a way similar to previous work [5], [7], the notification mechanism is hybrid, and uses both polling and IPIs (Inter-Processor Interrupts). When there is at least one core (i.e. thread) of a cluster executing kernel code, no IPI is sent, and it is up to the cores of the cluster to poll the circular buffer to check for incoming messages; this polling is done each time a core releases a lock. If all cores are executing in user space, an IPI is sent to notify one of the cores. The goal of this strategy is to avoid useless context switches when executing in kernel space while minimizing the latency when executing in user space.

c) Deadlock Avoidance: Since blockable RPCs can send other RPCs, it is important to ensure that there is no deadlock.

Our deadlock avoidance strategy is based on two properties. First, it must always be possible to send a response to a sender. This property is met since the response buffer is pre-allocated by the sender. Second, it must always be possible to treat an RPC request. Our implementation meets this property by (1) periodically checking for incoming messages and by (2) dynamically creating threads to handle incoming messages. Incoming messages are treated by specialized kernel threads, called *handlers*. There is one pool of handlers per core. The pool has initially only two handlers. New handlers are created only when a handling thread is going to block and the pool is empty. If a new thread cannot be created, the message is still treated by a special thread which sends back an error code indicating the lack of memory.

3) Unifying the File System Stack: The file system stack is composed of two layers. The first layer, called VFS (for Virtual File System), abstracts the difference between file systems of the second layer. It also caches in memory both the metadata, composed of *inodes* and *dentrys*, and the data of the file system, called data cache. An inode contains all the information relative to a file or directory, such as the size of the file. The only exception is the name of the file or directory. This latter is contained in the dentry structure. The second layer implement functions that are specific to a file system. The current implementation supports two pseudo-file systems (*devfs* and *sysfs*) and one disk file system (FAT).

This stack is distributed per inode, across the clusters. To keep this distribution uniform, the placement is decided by a hash function. The goal of this distribution is to avoid potential bottlenecks that may arise if the stack was placed in one cluster. The dentrys are placed in the same cluster as the inode (directory) to which they belong. The pages of the data cache are also placed in the same cluster as the inode to which they belong.

As an example, Figure 3 presents the placement of the different file system structures on a 2x2 cluster architecture.

Accessing a file or directory that is local to a kernel unit is done directly as in a classical monolithic kernels, with the usual synchronization techniques : locks, reference counters, etc. Accessing remote structures requires the use of RPCs.

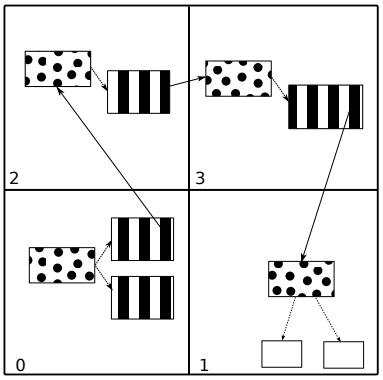

Fig. 3. Example of the possible placement of the file system structures. Inodes are represented by dotted boxes. Dentrys are represented by boxes with stripes. The remaining two structures represent the pages of the data cache belonging to the corresponding inode.

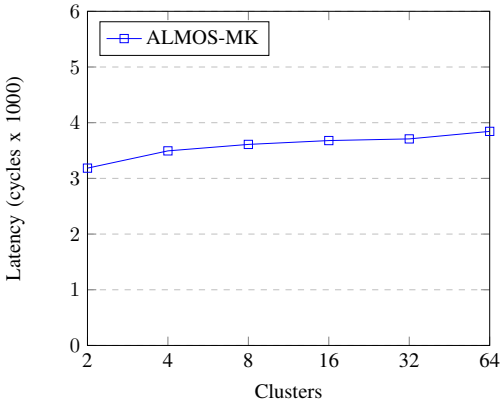

Fig. 4. Inter-clusters messaging cost

This placement strategy is similar to the one used in *Hare* [12], although the latter uses a per unit local cache, which caches recently accessed metadata. This optimization is useful to improve the performance of the path lookup operation, to avoid sending an RPC for each path component.

4) Migrating Processes across Clusters: Processes are migrated at the execution of an *exec*-like system call (in UNIX systems, this system call loads a new executable file into a process). This choice limits the data to be transferred across clusters. The main argument sent to the remote cluster is the path to the executable file.

Placement Strategy: The current implementation uses a per cluster round robin strategy. This strategy uses a per cluster counter. This counter is incremented each time a process is placed. It is the value of this counter, modulo the number of cores, which specifies the target core.

In addition to this strategy, user applications can explicitly specify a core in which a process is to be placed.

5) Remaining steps: Two more steps are yet to be implemented: the support for thread migration and the support for a general process migration service (i.e. independent of *exec*-like system calls). These two steps will allow ALMOS to execute highly parallel applications. In the current implementation, both threads and processes of the same application are bound to one cluster.

D. Portability

Restructuring the kernel as a distributed system does not necessarily depend on the use of physical accesses, it can use the virtual space. In this case, each kernel units virtual space statically maps the space local to a cluster except for few pages. These pages will be used to implement the message passing service. Such a solution allows the kernel to handle approximately 1 GB of memory per cluster without using the dynamic mapping technique. This size can be further increased by increasing the decomposition of the kernel. For example, to handle all the available memory in the case of TSAR using the virtual memory, we can set one kernel unit per core rather than per cluster. This allows the kernel to handle 4 GB of physical space per cluster, considering that we have four cores per cluster. However, increasing the decomposition increases the memory footprint of the kernel as more data are replicated.

The direct copy functionality can also be implemented without using the DEA register of TSAR. It can be implemented using the dynamic mapping technique, or an architecture specific mechanism, e.g. using a DMA peripheral to copy the data.

VI. Experimental Evaluations

In these preliminary evaluations, we try to answer four questions: *i) What is the cost of sending a simple RPC? ii) How does the new structure of ALMOS scale when accessing a shared resource?* We restrict this question to a shared resource since both kernel structures (monolithic and multikernel) can handle the scalability of private resources. *iii) How does the new structure affect the performance when accessing private remote resources?* We restrict this question to remote resources since the performance for accessing local resources should be similar to both kernel structures. *iv) How much do we gain in performance for executing the kernel directly in physical space?* The goal of this question is to evaluate the number of TLB misses.

To answer these questions, we conduct three experiments. These experiments are applied to the old version of ALMOS as well as the new multikernel ALMOS, which we call ALMOS-MK.

All the experiments are done using the Cycle-Accurate-Bit-Accurate (CABA) SystemC [9] TSAR simulator. Although, an FPGA implementation exists, the simulator allows us to reach a higher number of clusters/cores.

A. Message Passing Cost

Figure 4 presents the average time for passing a message to a remote cluster. This message has one argument and one return value. The sent argument is an integer. The return value is the incrementation of the argument. As we can see, the cost of passing such messages lays around 3,500 cycles and varies little when the number of clusters is increased. Although these results were obtained after many optimizations, we believe that they can still be improved.

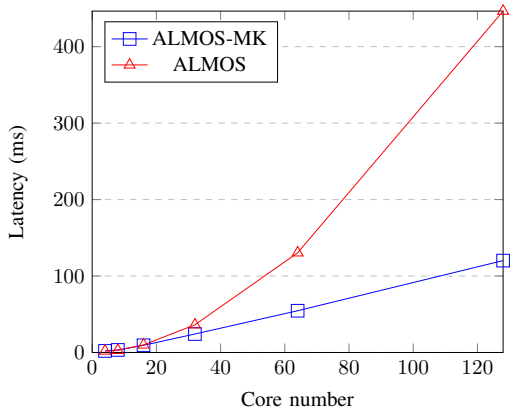

Fig. 5. Parallel read cost

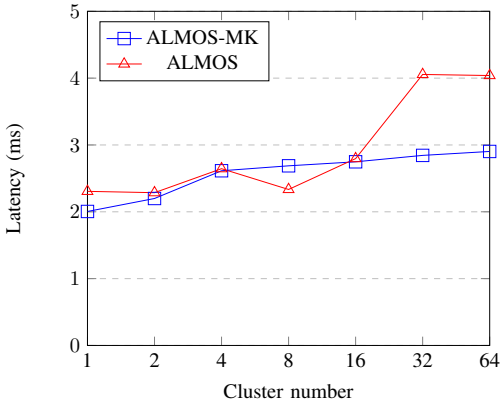

Fig. 6. Migration cost

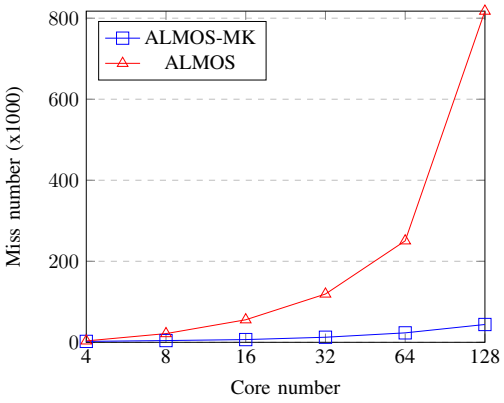

Fig. 7. Number of TLB misses for the parallel read benchmark

B. Performance Scalability

This experiment measures the average time for accessing a single file in read only. We choose to experiment on files, since they are the only memory objects which are shared between processes across clusters. We restrict access to read only to avoid disk accesses, which may bias the results. Indeed, read accesses do not require disk accesses once the file is cached in memory. In this experiment, the number of cores varies between 8 (2 clusters) to 128 (32 clusters). There is one process per core. Each process reads 256 times a portion of a file composed of 128 bytes. To synchronize the start of the processes, we use a barrier. This barrier is implemented using a file which is used as a shared memory buffer. This buffer contains a bitmap of N bits, where N is the number of cores. At the initialization, all bits are reset. Each time a process reaches the barrier, it sets the bit corresponding to the core on which it executes. Once all bits are set, processes start executing the read operations.

Figure 5 presents the results of this experiment for both ALMOS and ALMOS-MK. As we can see, starting from 64 cores, the performance gap becomes very important. Both operating systems use the same synchronization mechanism to access the file: a read-write lock. The difference is that ALMOS-MK restrains the number of processes contending for the lock to four, i.e. the number of cores per cluster. All other processes access the file through the message passing service. This experiment clearly shows the scalability of ALMOS-MK when accessing shared resources.

C. Remote Access Performance

In this experiment, we measure the cost of migrating a process from the first cluster to the last cluster of the platform. The number of clusters varies from 1 (4 cores) to 64 (256 cores). We use more clusters in this experiment because the gap of performance appears only at 32 clusters.

Figure 6 presents the results of this experiment. As we can see, ALMOS-MK handles better the NUMA effect of the architecture. This is explained by the fact that, in ALMOS-MK, all accesses are local, except when passing messages.

D. TLB Performance

To measure the impact of physical addressing on the TLB misses, we introduced instrumentation counters in the virtual prototype of the TSAR architecture. Only the counters of active cores (not idle) are taken into account.

Figures 7 and 8 show the total number of TLB misses for the two previous experiments: the parallel read and the process migration experiments.

For both experiments, the results of ALMOS-MK are better

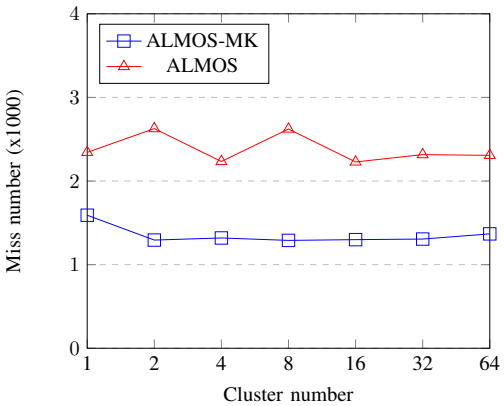

Fig. 8. Number of TLB misses for the migration benchmark

since the TLB are used only by the user space. This exclusive use of the TLB by user applications avoids both types of misses: capacity misses and conflict misses. The first since the user has more space to store recent translations. The second since there is no conflict with kernel cached translations.

Moreover, the resulting difference is clearer in the first experiment. This is due to two reasons. The first reason is that the first experiment is longer than the second leading to the appearance of more misses, considering that the miss frequency is constant over time. The second is that the amount of data accessed is more important in the first experiment. These data increase with the number of cores, while in the second the number of accesses is constant, and most cores are idle.

These results also explain why the cost of remote communication of ALMOS-MK, around three thousand cycles per RPC, does not affect its performance. This is because the cost of the RPCs is compensated by the decrease of TLB misses.

VII. Conclusion and Future Work

Memory management is the central problem of many-core architectures. More cores require more memory. It is therefore necessary to manage a physical space well beyond 4 GB, meaning physical addresses much larger than 32-bit. However, to integrate hundreds of cores in a chip, it is necessary to minimize energy consumption by reducing the footprint of each core. It is also necessary to save the cache space in order to reduce the miss rate, thus reducing accesses to external memory. For these two reasons, 32-bit cores can be a better choice than 64-bit cores.

In this paper, we have shown that it is possible for the operating system to efficiently manage a 40-bit address space with 32-bit cores. Our solution decomposes the kernel into multiple kernel units, each handling the space of a cluster. This allows us to exploit as much memory as the number of clusters. Each unit runs directly in physical space, and communication between units is ensured by a message passing service. This decomposition of the kernel is transparent to user space applications, except for the larger virtual space that they can access: nearly 4 GB.

To run directly in physical space, the solution depends on the TSAR architecture physical address space. However, the solution can still be implemented using the virtual space, but with a smaller user virtual space. Moreover, the physical address space of the TSAR architecture is simple enough to be implemented in other architectures.

The solution is evaluated using three experiments. The first one characterizes the cost of an RPC to 3500 cycles, whatever the number of cores. The second analyzes the scalability of the system when accessing a shared resource. The third analyzes the performance for migrating a process. These experiments were done on the same operating system (ALMOS) but with and without our solution. The results show a good scalability of the system, good performance and good handling of the NUMA effect.

This work is still in progress, and future work includes incorporating more services, such as the thread migration service. However, it is enough to demonstrate the feasibility of using 32-bit cores to handle large physical space on manycore processors.

VIII. Acknowledgements

This work was supported by the European CATRENE project: SHARP CA109; and by the French ANR project: TSUNAMY ANR-13-INSE-0002.

References

[1] 4G/4G split on x86. Accessed: 2015-11-26.

[2] Ghassan Almaless. Almos : un systeme d'exploitation pour manycores en memoire partagee coherent. In *Proceedings of the 8th French Conference on Operating Systems (CFSE), the French chapter of ACM-SIGOPS, GDR ARP, Saint-Malo, France*, 2011.

[3] Ghassan Almaless. *Conception d'un système d'exploitation pour une architecture many-cores à mémoire partagée cohérente de type cc-NUMA*. PhD thesis, Université Pierre et Marie Curie (UPMC), 2014.

[4] ARM. *ARM Architecture Reference Manual*. ARMv7-A and ARMv7-R edition, 2004-2012.

[5] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated kernel OS based on Linux. In *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[6] Luiz André Barroso, Kourosh Gharachorloo, Robert McNamara, Andreas Nowatzyk, Shaz Qadeer, Barton Sano, Scott Smith, Robert Stets, and Ben Verghese. *Piranha: a scalable architecture based on single-chip multiprocessing*, volume 28. ACM, 2000.

[7] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[8] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[9] Richard Buchmann and Alain Greiner. A fully static scheduling approach for fast cycle accurate systemc simulation of mpsocs. In *Microelectronics, 2007. ICM 2007. Internatonal Conference on*, pages 101–104. IEEE, 2007.

[10] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. *Hive: Fault containment for shared-memory multiprocessors*, volume 29. ACM, 1995.

[11] Alain Greiner. TSAR: a scalable, shared memory, many-cores architecture with global cache coherence. In *9th International Forum on Embedded MPSoC and Multicore (MPSoC09)*, volume 15, 2009.

[12] Charles Gruenwald III. *Providing a Shared File System in the Hare POSIX Multikernel*. PhD thesis, Massachusetts Institute of Technology, 2014.

[13] Intel. Intel Architecture Software Developer's Manual. Order Number 243192, 1999.

[14] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: a mobile operating system for heterogeneous coherence domains. In *ACM SIGARCH Computer Architecture News*, volume 42, pages 285–300. ACM, 2014.

[15] Mark E Russinovich, David A Solomon, and Jim Allchin. *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*, volume 4. Microsoft Press Redmond, 2005.

[16] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with os clustering. In *Proceedings of the sixth conference on Computer systems*, pages 61–76. ACM, 2011.

[17] SPARC International Inc. The SPARC Architecture Manual (Version 8). Revision SAV080SI9308, October 1991-1992.

[18] Ronald C Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *The Journal of Supercomputing*, 9(1-2):105–134, 1995.

[19] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.