apple core

SEVENTH FRAMEWORK PROGRAMME

FP7-215216

Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs

Specific Targeted Research Project (STReP) THEME ICT-1-3.4

Report on memory protection in microthreaded processors

Deliverable D5.2, Issue 1.0

Workpackage WP7

Author(s):	J. Masters, M. Lankamp, C. Jesshope, R. Poss, E. Hielscher		
Reviewer(s):	J. Masters, M. Lankamp, C. Jesshope, R. Poss		
WP/Task No.:	WP7	**Number of pages:**	10
Issue date:	2008-12-12	**Dissemination level:**	Public

Purpose: The purpose of this deliverable is to describe a memory protection scheme for microgrids of SVP cores.
Results: The main results of this deliverable are a mechanism for partitioning and hierarchical protection suitable for SVP on a microgrid.
Conclusion: The main conclusions are as follows: A single 64-bit address space is used and divided into statically partitioned, hardware supported Thread Local Storage and Global Storage. Protection is implemented in hardware by defining Protection Domains from families and storing their permissions in a global Protection Table. Elevation of privileges is possible through system creates, which index a System Call Table, defined per Protection Domain.

Approved by the project coordinator: Yes **Date of delivery to the EC:** 2008-12-12

Document history

When	Who	Comments
2008/11/20	J. Masters	Initial version
2008/11/25	M. Lankamp	Restructured
2008/11/27	C. Jesshope	Minor Corrections
2008/11/29	J. Masters	Minor Corrections
2008/11/30	J. Masters	Formatted to Apple-CORE template

Project co-funded by the European Commission within the 7^{th} Framework Programme (2007-11).

Table of Contents

1 Overview

The Apple-CORE project is developing many-core chip multi-processors (that we call microgrids), and compilers and operating systems that provide support for a range of applications from embedded systems through commodity computer systems to large-scale distributed systems. The model we adopt is based on SVP, which supports the hierarchical creation of families of threads [2]. The approach we adopt in Apple-CORE is to implement the actions defining the SVP model as instructions in the core's ISA [3]. This means that issues such as memory protection must also be considered in the design of the ISA. This report proposes an addressing scheme and extensions to an SVP core's ISA in order to support memory protection in a microgrid.

1.1 Requirements

There are a number of different requirements and constraints on the implementation of this model. The most fundamental is that we have the potential to have threads from many different software components running on many SVP cores concurrently accessing different areas of protected memory, as well as having such threads being interleaved on a cycle-by-cycle basis in a single SVP core. To avoid the costly overhead of switching address spaces on every thread switch, we adopt a single 64-bit address space that is shared between all software components executing on the microgrid. This concept is supported by the cache-only, on-chip memory system (COMA) we have proposed for the microgrid [1, 4]. This memory system requires an off-chip backing store, which provides the address translation between microgrids or indeed between microgrids and a conventional distributed system. Currently we have not considered the detailed requirements of the backing store.

The implementation of the core language μTC—which provides an interface to the SVP model—on SVP cores also places requirements on the memory protection and partitioning scheme. Each thread that executes in a microgrid requires its own dedicated region in the address space as Thread Local Storage (TLS) for both stack and local heap management. Having a dedicated region available avoids the problems of a parallel stack implementation and ensures that independent dynamic memory allocations can be performed in parallel. To avoid dynamically allocating and deallocating these TLS regions every time a thread is created and destroyed, a static partioning is required.

Within a microgrid, we also aim to support a partitioning of resources (the cores and memory) coupled with a partitioning of system services based on that resource partitioning. This is in order to avoid scalability issues in resource management on a many-core microgrid. It also implies a multi-rooted, distributed system, where each root (which may be user code) in some sense owns a set of resources, which it manages for its subordinate families. The SVP model is purely hierarchical; every thread can create a subordinate family, with or without new resources attached to the create action. The memory protection scheme must therefore support this hierarchical delegation of authority.

Note however, that not all systems that might be run on a microgrid fully conform to the hierarchical model of communication supported by SVP. Hence, we have a requirement for passing protected areas of memory between siblings rather than between a parent and its descendants in the SVP concurrency tree. An example of this can be found in the implementation of the coordination language S-Net [1], where multiple asynchronous records may need to be synchronised in a synchroniser before continuing on as one record to the synchroniser's successor. Records are by definition independent sibling concurrency trees and the synchroniser component can have only one output record (which is actually the last record to arrive at the synchroniser). All other records must be passed to the successor component as protected areas of memory. Currently, such sharing between components of a large distributed application would be performed either by sharing files and exploiting file-system protection or by plumbing components together and performing synchronisation as a part of the user or component code, e.g. using MPI. We abhor such plumbing techniques, as any internal synchronisation state in such components is a huge barrier to flexible and dynamic resource management [4].

Finally, note that this memory protection scheme will not apply for the 32-bit SPARC prototype. The prototype applies more to embedded environments and is not the target platform for memory

protection. Instead, we consider a 64-bit address space on a general-purpose system.

1.2 Legacy code

All code run on a microgrid can be divided into two categories: legacy and non-legacy. Non-legacy code is code built and compiled for the SVP model, able to support and use everything SVP has to offer, and able to work with the protection mechanisms described in this report. Legacy code, on the other hand, is code compiled for the unextended instruction set of the microgrid, but not for the SVP model and, as such, does not utilize families or threads. Supporting legacy code—a requirement of the commodity processor market—can be done in two ways:

1. Virtualizing the operating system interface to allow legacy user-space applications to run as one or more families of one thread on a microgrid.

2. Virtualizing the hardware to allow a legacy operating system including legacy application to run as a family of one thread on a microgrid.

The former method is the easiest, as all that it needs is emulation of the operating system call interface. The only thing required from hardware is to support whatever system call method is used (e.g., interrupt instructions), and hook them into custom system calls that emulate a legacy operating system by using SVP mechanisms. This method can be used to run legacy applications on a microgrid.

The latter mechanism effectively allows an entire legacy system to be virtualized and can be compared to current virtualization techniques. Since the legacy operating system has no knowledge of the SVP system, the virtualization mechanism must somehow virtualize access to resources such as system registers, memory structures and I/O, some of which may not even physically exist on an SVP implementation. While supporting this method of virtualization has applicable use cases, it still requires further investigation to see if it is feasible on a microgrid.

2 Protection Domains

2.1 Definition

In order to provide memory protection on a microgrid we define *Protection Domains* (PDs) that are conceptual entities that contain a list of *Address Ranges* and permissions on those ranges. Every family running on a microgrid is associated with exactly one PD. All memory accesses made by threads in a family are validated, by hardware, against the list of Address Ranges and their permissions in the family's PD. In many ways the PD is the analog of the address space of a Unix process. The primary difference is that it defines a private set of access privileges to globally addressable pages, instead of a private virtual mapping environment [5].

To identify a PD, we recognize that in theory every family on a microgrid can be associated with a unique Protection Domain. For a typical microgrid, that would consistute 1024 (2^{10}) cores and 32 (2^5) families per core. A *Protection Domain ID* (PDID) can therefore be defined as a 15-bit integer, equal to the ID (CPU ID and Family ID) of the first family associated with that PD (Table 1).

Root CPU	Root FID
10	5

Table 1: Protection Domain ID

A PD is created by specifying, at the creation of a new family, that the new family is the root of a new PD. This operation can be done by any family at any level. If this is not specified, the new family will inherit the PD from its parent (Figure 1). All families associated with a specific PD are called a *component*, as this mechanism is typically used to seperate or isolate different software components. The PD of the first family that is started on a microgrid is the *root authority* (PDID 0). It is the operating system of the entire microgrid; the ultimately trusted authority. Initially, it has full access to all resources, which it can partition and give to sub-authorities, thus starting the hierarchical partitioning. While sub-authorities have full control over the resources they have been given, some operations, as described later, should only be done by the root authority to ensure system security.

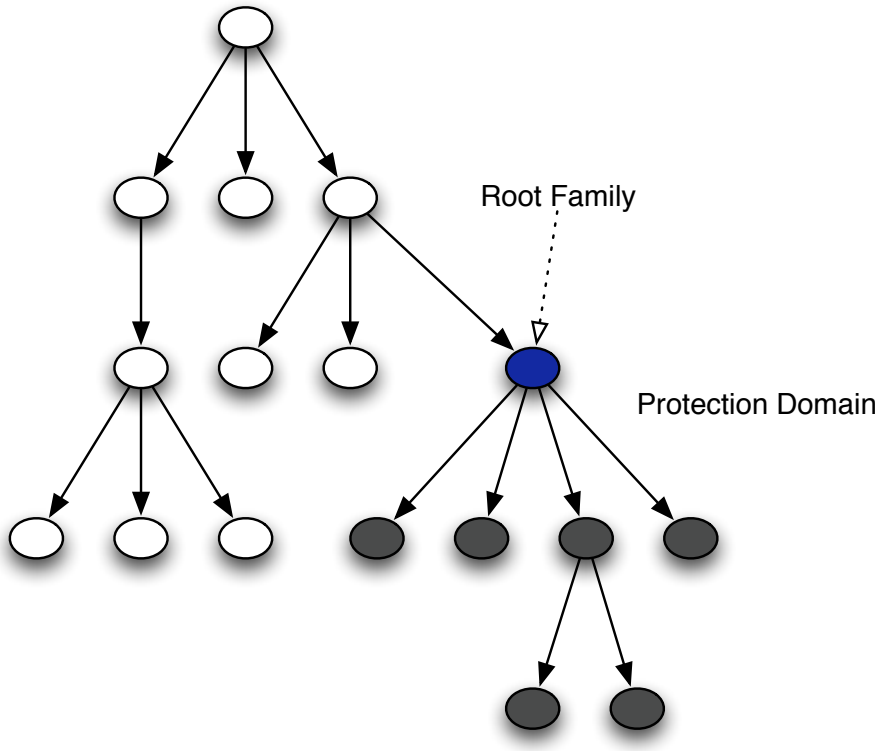

Figure 1: Protection Domain Tree

This mechanism, along with the guarantee by hardware that the first family in a PD will not be reused until all families in that Protection Domain have terminated, is both sufficient for protection and efficient to implement.

2.2 Protection Table

To implement protection, a global structure is defined, the *Protection Table* (PT). While the exact implementation of the PT is not yet fully defined, it can be indexed by PDID and address to obtain the privileges of said PDID for said address. This is done by hardware on every memory access, although PT entries may be cached in *Protection Lookaside Buffers* (PLBs), as described later.

To avoid authorities modifying the PT and giving themselves or their children access to sensitive memory, the root authority should never give out access to the PT to sub-authorities. If sub-authorities wish to give sub-sub-authorities access to their memory, they will have to request the root authority to make the change in the PT via a system call (see section 2.4).

Address Ranges may be as small as a cache line (e.g. 64 bytes), and are variably sized with the restriction that the size must be a power of two.

An Address Range in the PT is described by a virtual page number (VPN), and an integer to identify the number of don't care bits (i.e. the log_2 of the range size). If the smallest Address Range may be the size of the cache line, say, 2^6 bytes, we need 58 bits for the VPN. We use 6 bits for the size to store the number of VPN bits to be masked (0 - 63 for Address Ranges of size 2^0 - 2^{63} bytes, respectively). If there is a PT entry whose masked VPN matches the masked address of the memory request, as well as the PDID, then the access rights of that PT entry are used to determine whether the request is allowed. Overlapping Address Ranges for a single PDID are not allowed in the PT.

Protection Lookaside Buffer

Protecting global memory accesses naively is a very costly operation, involving not just an additional memory reference, as required in an address-translation table, but potentially a search of an unbounded (although in practice limited) structure of Address Ranges. Just as in address translation where a translation look-aside buffer (TLB) provides a fully associative cache onto the address-translation table, so in an SVP core we propose to implement a PLB to cache accesses to the Protection Table. Because Address Ranges may be specified in more than one PD, the PLB is accessed by PDID and address. A single entry in the PLB will therefore describe the access to a single area of memory for a given PD.

On each non-local[1] memory access request the processor sends the address to the cache and the address + PDID to the PLB. As all on-chip memory comprises one single, shared virtual address space, this can be done in parallel.

2.3 System Calls

Above we have described a mechanism to inherit protected memory from a parent family and to draw a security line when creating a new subordinate component, so that no memory is shared below in the concurrency tree, i.e. those creates for which a new PDID has been set as described above. This however, does not give us a secure mechanism for creating system threads, which must access protected data in some upper-level Protection Domain. It should be repeated here that we wish to create a distributed and multi-level collection of authorities. This can include multiple operating systems running on different cores or multiple layers of system control such that trust can be passed down the SVP concurrency tree. Thus, we do not want to fix to what level a system call should go.

The solution to execute system code is a *system create*. Any thread, at any level, can request the root authority to create a *System Call Table* (SCT), which is an array of (address, PDID) pairs. Each pair will define a service such as malloc, fopen, and so on. The index of each service in the SCT is a system-implementation dependent convention. Each PD will include a pointer to its SCT, which the system create uses to determine the address of the code, as well as the (presumbly more authorized) PDID to assign to the new family. This table will be located in memory accessable only to the root authority, to prevent threads from corrupting or abusing SCTs, even their own. All requests to create, modify and remove these SCTs should go through the root authority, which can verify if the requesting component is allowed to define the specified services. If allowed, the root authority sets the PDID of an overriden service entry to the PDID of the component requesting the override. Thus, components can only define services into its own Protection Domain. The system create will ensure that the created family gets the caller's PDID.

This system create mechanism allows threads at any level to define services to manage the resources they own, or to filter requests to higher authorities (e.g., file system virtualization).

[1] Local accesses are described in section 3.1

2.4 Memory Sharing

Since families in a new PD have no access to any data outside of their PD, they cannot access data from their parent. If this is required, which is very likely for parameters, the parent family has to explicitely share its data with the new PD after creating the new family by making a system call to request the sharing. This request will eventually (or immediately) result in a system call to the root authority, which will update the Protection Table for the new PD. Once the system call completes, the parent can send a pointer to the shared data to the new family via a shared parameter.

Sharing of memory does not necessarily have to occur between parent and child. For example, a thread may allocate memory, which is automatically added to its Protection Domain, but be required to share that memory with another thread, which is not its child (e.g., synchronisation in S-Net). This requires a reference to that memory to be passed in shared memory. This, in turn, requires the access rights to be set on a PD unknown to the allocator of the memory. The allocator is able to specify what permissions it is willing to grant when sharing memory with another PD. Ultimately, the Address Range must be added to the other thread's PD using a system call made by that thread. This call sets the rights granted by the allocator.

However, this method of sharing requires an additional protection mechanism. This mechanism is similar to the one used when allocating places in a microgrid [3]: shared areas of memory are protected with a capability. When memory is shared a capability is generated and stored in the authority's memory management structure (MMS). The capability is returned to the requesting thread as part of the reference to the memory. This reference may be passed to another thread and that thread will perform a system call to request the addition of the Address Range to its PD. This system call will consult the authority's own MMS to see if the Address Range was allocated from its own memory (i.e., both threads are in different components, but both children of the same authority). If it was, it can check the capability and request the root authority to change the component's PT to allow the share. If it wasn't, it must perform a system call to its authority for verification of the share reference. This process continues until the request arrives at an authority that recognizes, from its MMS, that the Address Range is part of memory that it owns. It can then continue the verification down the authority tree, until it arrives at the authority that allocated the memory, which can verify the capability. A response is then passed back up and down again until it reaches the original authority.

Reference counts

With memory sharing possible, allocated memory should be reference counted by authorities in their MMSs, as a memory region's owning PD can have terminated before all PDs that share that memory have terminated. When memory is allocated, it is allocated with a reference count of one. When memory is shared, the reference count must be incremented and when memory is freed, decremented. The memory is only released (invalidated in the COMA memory system) and available for reuse when the counter decrements to zero. Details of the data structures that represent the MMS need to be investigated further, as does the flexibility in sharing access to memory. It may be required for example, to allow memory to be passed many times between components where the permissions granted are monotonically decreasing. This requires more experience in implementing system software for the microgrid.

3 Memory Partioning

As a result of the requirements listed above, we've split the 64-bit address space into two equal regions: one for Thread Local Storage, used to store or spill frame variables and do thread-local heap allocations and one for Global Storage, used for everything else.

3.1 Thread Local Storage

A value of 1 in the most significant bit of the virtual address (the "local bit") is used to identify an address as belonging to thread local memory. To accomodate the required static partioning, we divide the remaining 63-bit address space equally among all thread entries on the micro-grid. With 2^{10} cores and at most 2^8 threads per core, we reserve 18 bits in the address for this purpose. Since the data in TLS is, by definition, thread-local, we can improve performance by removing the protection-check for memory accesses to a thread's own TLS. However, this introduces a security issue. A thread entry can be re-used by a thread belonging to a different PD, and that thread should not have access to the old thread's memory. Rather than clear or invalidate TLS memory on thread creation/termination, we solve this by putting the PDID in the address as well.

We thus partition the 63-bit TLS memory space into 2^{15} memory regions, one for each PD. Each of those memory regions is partitioned into 2^{18} memory regions, one for each thread, leaving 1 GB of protected memory available to each thread (Table 2). Both the PDID and the CPU and Thread ID of a newly created thread are trivially known by the hardware, so the resulting TLS pointer can be trivially composed and made available to the new thread as a pointer to its TLS (e.g. in a register in the thread's context). The performance of accesses to TLS memory is maintained, by checking the PDID in the address against the PDID of the executing thread (Figure 2).

Local	PDID	CPU	TID	Offset
1	15	10	8	30

Table 2: TLS base address

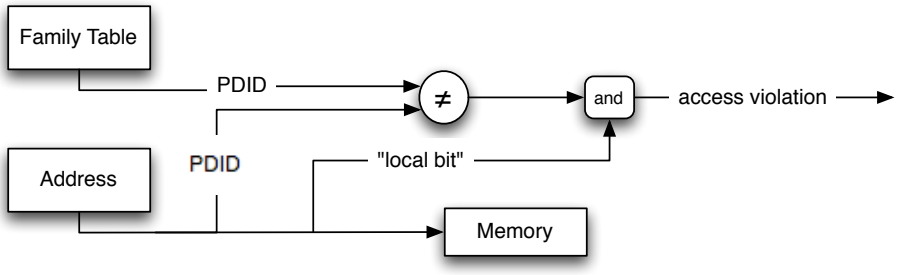

Figure 2: Implicit TLS protection mechanism

Note that for legacy code, which is executed by creating the code as a single-thread family as a new PD, there is no need to differentiate between threads in a PD, thus it can ignore the thread partioning, leaving it with 2^{48} bytes (256 TB) of local memory.

3.2 Global Storage

A value of 0 in the most significant bit of the virtual address (the "local bit") is used to identify an address as belonging to Global Storage (Table 3). This memory is not statically partitioned in any way and can be allocated by the top-level system however it wants. Global Storage will likely contain all code, memory management structures and shared data in a system.

If a thread wants to extend its Protection Domain beyond TLS, it may request Global Storage through a system call which will allocate a region of memory from the authority's Global Storage, if possible, and request the root authority to update the Protection Table for the caller's PD. By overriding the memory allocation system calls, memory partioning can be achieved by having authorities serve memory requests from their own area of global memory.

Local	Offset
0	63

Table 3: GS address

4 Conclusion

In this report we have established a mechanism for partitioning and hierarchical protection suitable for SVP on a microgrid. A single 64-bit address space is used and divided into statically partitioned, hardware supported Thread Local Storage and Global Storage. Protection is implemented in hardware by defining Protection Domains from Families and storing their permissions in a global Protection Table. Elevation of privileges is possible through system creates, which index a System Call Table, defined per Protection Domain.

We have established that best practices for a general purpose Operating System on a microgrid involve never handing out direct access to the Protection Table or System Call Tables, while allowing applications and other untrusted[2] authorities to override system services to provide localized allocation, protection or virtualization.

These hardware mechanisms and software conventions allow for memory sharing between arbitrary cooperative threads in the concurrency tree while keeping most communication localized to the partition in which they reside.

[2]untrusted from the operating system's point of view

APPENDIX A - Example

To illustrate the overall picture of the design, consider the example illustrated in Figure 3. The example shows a system with a root PD having created various applications. Each of those applications runs in its own PD. Application A (in PD A) created various families and threads, two of which created two sub-components, B and C, each in their own Protection Domain. Both B and C also created sub-components of their own; D and E, respectively, both also in their own Protection Domain.

A schematic overview of the hierarchy is shown to the top-left, and the layout of the various memory allocations and data structures is shown to the right. The Memory Management Structures of the Root, A, B and C Protection Domains are shown to the bottom-left and the contents of the Protection Table is shown at the bottom, along with which authority added or requested the addition of the entries. The SCTs are also shown, as well as the *System Call Table Pointers* for each Protection Domain.

As can be seen from the figure, only the Root PD has access to the System Call Tables and Protection Table by virtue of them only being writable in PD 0's PT entry. When PD A was started, the Root PD allocated memory for it, which it keeps track of in its MMS. A wants to manage that region of memory itself, so it creates an MMS in its allocated region, to manage the rest of that region. It requests the Root PD to create a SCT for it (SCT A), with memory management functions redirected to its code. After creating components B and C, they both request memory from A through their SCTs. A allocates the memory and requests the Root PD to update the PT to give B and C Read/Write access to their allocated memory. B and C also want to mange their own memory, so they proceed akin to A and create components E and D, respectively. E now wants to share some data with D. The steps that are taken are as follows:

1. E allocates memory from B and requests R/W share access
2. B allocates the memory (S), creates a capability and updates its MMS to reflect the share.
3. E passes the reference (address, size and capability) to a thread D.
4. D uses the reference to request access to the memory. This is a system call and goes into PD C.
5. C checks its MMS to see if it owns the address. It doesn't.
6. C sends a validation request to its parent, A.
7. A checks its MMS to see if it owns the address. It does, but notices it has been allocated to B.
8. A hands the validation request down to B.
9. B checks its MMS to see if it owns the address. It does, and the capability matches.
10. B increases the reference count of the Address Range to 2 and returns a success status and the allowed access rights (Read/Write).
11. A increments the reference count for B's memory and hands the reply back down to C.
12. C, now having validated the share request, updates its MMS with the new memory information and requests the root to update the PT to allow D to Read/Write S. C's MMS now also serves as a cache for future requests to the same reference, with the same capability.
13. The Root PD updates the PT and returns.
14. The original system call from D completes succesfully and D can now access the memory in the reference.

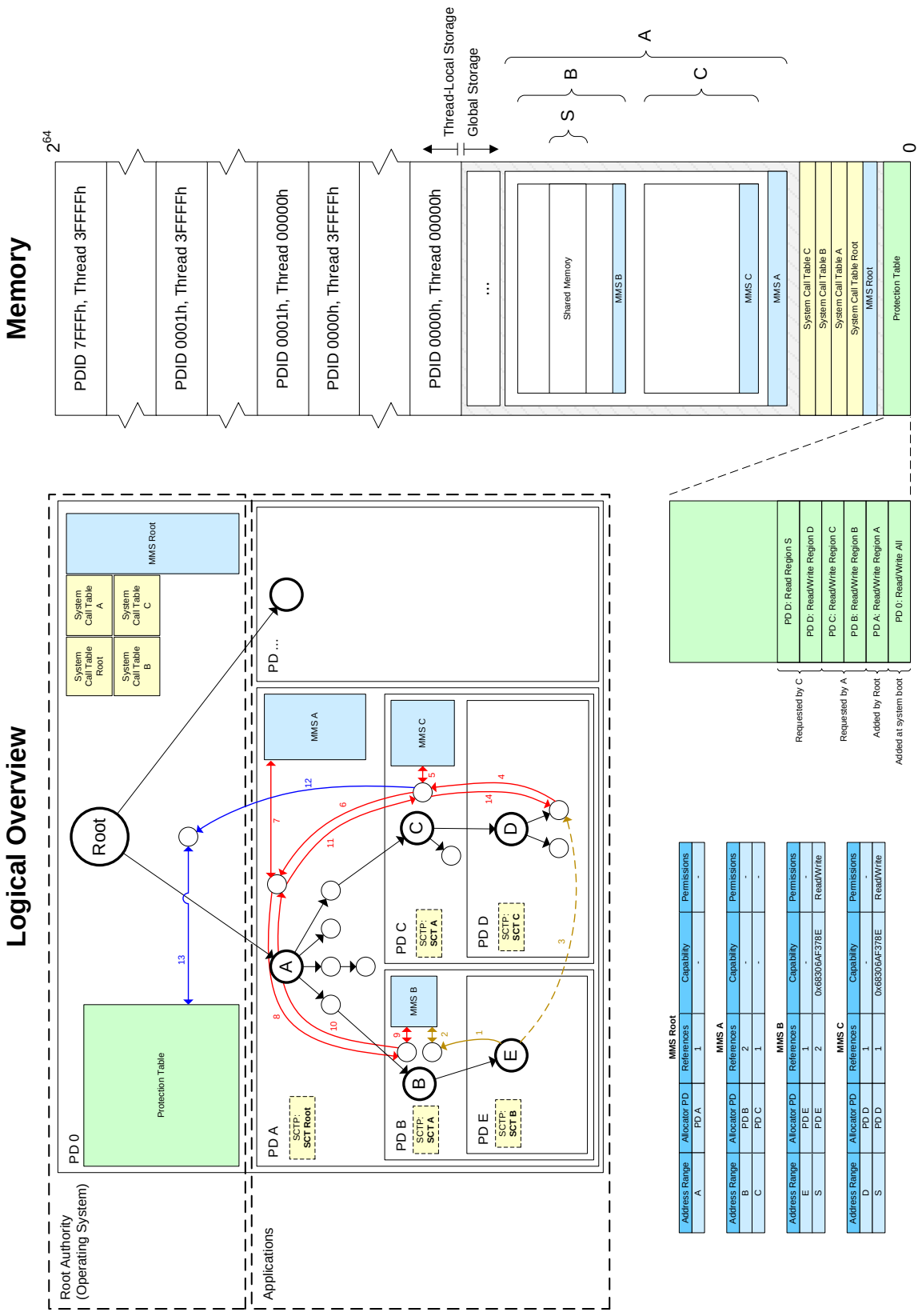

Figure 3: Memory sharing example

References

[1] C. Grelck, S.-B. Scholz, and A.V. Shafarenko. A gentle introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[2] C.R. Jesshope. A model for the design and programming of multi-cores. *Advances in Parallel Computing*, 16:37–55, 2008.

[3] C.R. Jesshope. Operating systems in silicon and the dynamic management of resources in many-core chips. *Parallel Processing Letters*, 18(2):257–274, 2008.

[4] C.R. Jesshope and A. Shafarenko. Concurrency engineering. 2008.

[5] E.J. Koldinger, J.S. Chase, and S.J. Eggers. Architecture support for single address space operating systems. *SIGPLAN Not.*, 27(9):175–186, 1992.