From *Proceedings of the 25th Hawaii International Conference on System Sciences (HICSS-25)*, Vol. I, pp. 720–729.

A Model for Address-Oriented Software and Hardware

Walter R. Smith
Robert V. Welland
Apple Computer, Inc.
Cupertino, California 95014
`wrs@apple.com`, `welland@apple.com`

Abstract

We introduce the concept of "address-oriented software", which is software that assigns particular meaning to the values of memory addresses. Such software is often not well supported by the services of the operating system in which it operates. We provide some examples of address-oriented algorithms, and propose a general model of the operations they use, called "address management", which we intend to use as the basis of a new operating system. We derive a hardware model from our formal model, and suggest implementation techniques. A partial implementation of our hardware model is an integral part of the ARM 600 processor, currently in development.

1. Introduction

Programs that manipulate pointers usually do so without using knowledge of their actual values. This tendency is reflected in "safe" languages such as Pascal and Lisp that admit the concept of an opaque pointer to a data object, but provide no operations to get the memory address from a pointer. However, there is a class of algorithms, which we call *address-oriented,* that do rely on the values of memory addresses. For example, a generational garbage collector may determine the age of an object by examining its address, an object-oriented language implementation may define an object's identity by its address, and a persistent programming system may provide a translation between different address spaces and object representations while preserving address-based object identity in both.

Most operating systems and processors in current use have a system model that is designed to support well-known address-oriented services like virtual memory and memory-mapped files. Because of this limited model, it is sometimes difficult to implement a new address-oriented algorithm on one of these systems. For example, Mach [10] provides some low-level address-oriented mechanisms with its "external pager" interface [16], but (as the name implies) clients of that interface must be structured like demand-paged virtual memory services.

We are developing a new operating system that is conducive to the implementation of new address-oriented algorithms. Specifically, we have derived a simple set of concepts we call *address management* that can describe a wide range of address-oriented algorithms. We intend to use this model of address management as the basis of our operating system. The model is described in Section 2.

The development of our model has been guided by the desire to support several useful algorithms, some of which have been hard to implement on existing operating systems. We present a brief survey of some of these algorithms in section 3, along with an interpretation of each in terms of address management. With this section as motivation, we proceed in section 4 to formalize our model of address management.

We have designed new memory management hardware to provide efficient support for our software abstractions. Conventional memory-management hardware is optimized to support virtual memory, which makes efficient implementation of more general operations difficult. The model described here can be translated quite directly into hardware. In section 5, we extend the formal model slightly and use it to develop a hardware architecture.

Part of the hardware design of section 5 has been implemented for the ARM 600™ processor, currently in development. Section 6 briefly describes that implementation.

2. Address management

In our model, a computer system contains many cooperating tasks. Some tasks provide address-oriented services such as garbage collected heaps, persistent object

ARM and ARM 600 are trademarks of Advanced RISC Machines, Ltd., Cambridge, England.

stores, virtual memory, or memory-mapped files to other tasks. These services are presented to their clients as special regions of memory that behave in particular ways.

2.1 Tasks and the single address space

Tasks are independent threads of control that share an address space. There is one address space for the entire system. This simplifies the model, and provides important benefits in the operating system.

Separate address spaces are useful for isolation of programs from one another, but that isolation makes it harder for programs to cooperate. For example, implementation of shared code libraries is more difficult with multiple spaces than without. Unwanted memory access can be prevented with permissions; separate spaces are not necessary for protection.

The size of the single address space might be a concern, but persistent programming languages alleviate address space limitations. A persistent object store with an address space larger than the processor's address space requires some mechanism for translating machine addresses to persistent addresses. This effectively renders the size of the processor's address space irrelevant, as long as it is large enough to accomodate a reasonable working set, because work is really taking place in the larger persistent address space. Support for persistent languages is an important goal of ours.

The most important advantage of the single address space model is the possibility of system-wide object identity defined by the virtual addresses of objects. Since no two objects can occupy the same address, object addresses can be used as unique identifiers or capabilities throughout the system.

2.2 Domains

The system address space is partitioned into *domains*. A domain is a possibly non-contiguous set of addresses. Domains do not overlap, so an address can be in only one domain. Each domain has an associated task, called its *manager*, that provides the service associated with the domain.

Domains are associated with *controls*, which affect the execution of memory access operations. The set of controls is open-ended. In this paper we will consider only two controls: virtual-to-physical address translation and access permissions.

2.3 Environments

Each task operates in a context called an *environment* that defines its relationship to the domains in the system. Environments may be shared among similar tasks.

2.4 Mappings

During instruction execution, various translations are performed on addresses. These translations are defined by *mappings*. The most important mappings are the virtual-to-physical address mapping and the address-to-permissions mapping.

2.5 Faults

A *fault* is an event generated by the execution of an instruction. The execution process will halt an instruction and *signal* a fault if the appropriate conditions are met. For example, a fault is signalled when a memory access is attempted that is disallowed by the address-to-permissions mapping. A signal is usually passed to the manager of some domain determined by the type of fault and the operands of the instruction. Depending on the situation, the instruction may be restarted after the fault is handled.

2.6 Barriers

A barrier is a restriction of read access, or a restriction of the flow of addresses or control flow between domains. Violation of the barrier causes a fault to be signalled, which in turn causes some action to be taken. There are three kinds of barriers:

A *read barrier* restricts reading from a location.

A *write barrier* restricts writing pointers into a domain when they point into another domain.

A *control-flow barrier* restricts program counter movement between domains.

Write barriers and control-flow barriers are called *cross-domain* restrictions, because they are relations between domains.

2.7 Address detection

A write barrier only applies when the datum being written is an address. In order to implement write barriers, the system must be able to distinguish pointer data from non-pointer data. This is a common requirement of address-oriented systems such as garbage collectors. The usual solution is to use tag bits in each datum to flag the pointers. This can be implemented in hardware or software; the specific details are not important to our discussion.

3. Typical address-oriented services

This section describes some address-oriented services that we are particularly interested in supporting. For each algorithm, we show how it can be implemented within our model of address management.

3.1 Virtual memory

A virtual memory service [7] allows a program to use mass storage as working memory by providing the illusion of a large physical memory. The service ensures that its clients will be able to allocate and use memory without bothering to manage physical memory. Physical memory is used as a cache for the backing store, which is kept on a mass storage device. By manipulating memory permissions, the available physical memory is invisibly supplied to the client where it is needed at the moment.

The service defines a region of address space that will behave as the virtual memory. Within this space, some pages will be "valid", meaning that they are mapped to addresses in physical memory; others will be "invalid", meaning that they have no associated physical memory, and are either unallocated or swapped out to disk. This is implemented by manipulation of the virtual-to-physical address mapping.

The service is notified when a client attempts to access an invalid page. The service chooses a set of pages that will be swapped in (presumably including the one the client wants). It changes the address-to-permission mapping for the set to "no access", maps physical memory into the pages, fills the memory with appropriate data (usually read from the storage device), and releases the permissions so the clients can access the new data. When physical memory becomes scarce, the service needs to pick some valid pages and swap them out, once again unmapping them and preventing client access.

3.2 Garbage collectors

In a garbage-collected system, garbage is storage that is no longer accessible by "legal" language operations. A garbage collector finds garbage and makes its storage available for use. Many garbage collectors do this by finding all storage objects that are *not* garbage and eliminating the rest. Non-garbage objects are found by recursively following all pointers contained in storage objects, starting at a set of root objects, which includes global variables and the processor state of all tasks. We will consider three algorithms for garbage collection: stop-and-copy [4], concurrent [1], and generation scavenging [13].

Stop-and-copy collector: Memory is divided into two regions, called *from-space* and *to-space*. During normal execution, objects reside in to-space; from-space is not used. When to-space is exhausted (or nearly so), processing halts. The identities of the spaces are exchanged (*flipped*): from-space becomes to-space, and vice versa. The root objects are copied into to-space. The objects in to-space are then scanned for pointers into from-space. When such a pointer is found, the referenced object in from-space is copied to the end of to-space, where it will eventually be scanned, and the pointer in the object being scanned is updated to refer to the referenced object's new location. A forwarding pointer is left in place of each copied object, so later references to the object can be updated.

This process has the effect of copying all the reachable objects into to-space, and updating all the pointers in the objects to refer to the locations of the copies. When all the reachable objects have been copied, any objects left behind in from-space are garbage, and from-space can be reclaimed. Processing resumes, and the algorithm repeats.

Concurrent collector: The concurrent collector is like the stop-and-copy collector, but it does not require the clients to stop running while the collection occurs. It enforces a requirement that the client is not allowed to read pointers to from-space, and thus cannot tell that the collection is incomplete.

The restriction on clients' reading addresses in from-space is a read barrier, which is implemented by manipulating the address-to-permission mapping. After the flip, all memory that could contain addresses in from-space is protected from client access by turning off the clients' read access, and the copying process begins. When a page in to-space has been scanned, all the pointers in the page have been translated into to-space pointers; the client can access it without seeing from-space addresses, so the protection on that page can be removed. If the client attempts to access a protected page, a fault is signalled and the collector is notified; it immediately scans and translates that page, unprotects it, and allows the client to proceed.

Generation-scavenging collector: Most objects become garbage very soon after being allocated, and older objects are less likely to become garbage. Generation-scavenging collectors exploit this information by concentrating their efforts on younger objects. Memory is divided into multiple regions called *generations*, which are ordered by age and can be independently garbage-collected (*scavenged*). Objects are created in the youngest generation. When an object has survived several scavenges, it is moved to the next-older generation. Older generations are scavenged less frequently than newer generations.

To scavenge generations independently, the collector must be able to determine which objects in a generation are referenced from other generations, since these objects must not be collected even if there are no references to them within their own generation. It is uncommon for an object to reference objects younger than itself, so references from older generations to newer generations can be recorded with low overhead (old objects that reference newer objects are called the *remembered set*). Because references from younger to newer objects are not remembered, it is necessary when scavenging a generation to scavenge all younger generations at the same time; this is not much additional effort.

The collector must detect addresses being written to a newer generation than their own. This is an address-flow barrier. One domain is assigned to each generation, and appropriate cross-domain restrictions are created. Writes of addresses to newer generations signal cross-domain faults to the collector, which records them for future use.

There are alternative implementations of remembered sets [17]. The collector can write-protect older generations and record pages that were touched, then later scan the pages looking for references to newer generations. This obviously adds overhead to the collection, since more objects than necessary must be scanned, but it may be more efficient depending on the available hardware. A software-only implementation is also possible; inline code can be generated before all writes that quickly marks a data structure with the location of the writes. Since writes are uncommon, this may be efficient.

3.3 Persistent object store

A persistent object store [2] is a repository for data objects that outlive the programs that created them. There are many widely varying definitions and implementations of persistent object stores. Because it uses several features of our model, we will concentrate on Paul Wilson's proposed scheme in [15], although similar techniques are used by ObjectStore [3] and Cricket [12].

This kind of store is similar to virtual memory in that objects are mapped into memory from a disk. However, because the store has its own object addressing scheme, addresses of objects in the store are not the same as memory addresses. For example, they may be larger than memory addresses, or store addresses may refer to whole objects rather than byte offsets. Objects in the store contain pointers in this format, yet clients access objects as though they were present in memory in the usual form, with pointers in the format of memory addresses. Before the client can access an object, its pointers must be translated.

When the store is opened, the root objects of the store are read into memory and translated to memory format. Virtual address space is allocated for the objects referenced by the root objects, but no physical memory is mapped into it, so the service is signalled whenever clients attempt to read or write to this space. Upon notification, the service reads the appropriate object into memory and translates it, allocating space for the objects referenced by the object as before, and the process repeats.

The store has a requirement like that of the concurrent garbage collector, only instead of from-space addresses, it is persistent pointers that clients are not allowed to see. The collector translates an object by following pointers from the object into from-space, copying referenced objects into to-space and updating the pointers. A store translates an object by allocating virtual addresses for the referenced objects and putting them into the object. Like the collector, the store manipulates the permission mapping of its domain to intercept reads and writes to untranslated pages.

The store is also similar to a virtual memory service in that it dynamically assigns physical memory to the parts of the store that are resident in memory. It manipulates the virtual-to-physical address mapping of its domain to assign physical memory where it is needed.

Like the generation scavenging garbage collector, the store must detect cross-domain address operations; in particular, it needs to know about pointers from persistent objects to transient objects, so it can make those transient objects persistent. This is an address-flow barrier like the intergenerational one, and it may be implemented in the same ways.

3.4 Transaction control

In a transaction-based system [9], multiple tasks share the same data. Simultaneous access to the data is controlled by making the tasks participate in *transactions*, which are sequences of operations that are combined into single atomic actions. Tasks acquire and release *locks* on data in the course of a transaction; by tracking the manipulation of the locks, the transaction system can enforce atomicity by aborting conflicting transactions.

Systems such as Cricket [12] provide page-based locks without requiring clients to explicitly acquire and release them, through manipulation of permission mappings for individual tasks. At the beginning of a transaction, a client task has no permissions on the transaction domain. If a client attempts to read from a page, the transaction manager assigns that client a read lock on that page, gives the client read access to the page, and allows the client to proceed. At this point, the client can read the page freely, but the service will still be signalled if the client tries to write to the page, or if another client accesses the page. The rest of the locking system is similarly implemented.

Transaction services have a requirement unique among the services described here: different clients of the service need different permissions for the service's domain. The set of locks held by a client determines the permissions that client has. Each task runs in a unique environment.

4. A formal model of address management

In section 2, we informally described our model. We now express it in more formal terms.

4.1 Definitions

V is the set of addresses generated during an address operation; this set is often called the "virtual address space" of the processor. *P* is the set of addresses used to access physical memory.

E is the set of environments.

$AT = \{R, W, IF\}$ is the set of access types (read, write, and instruction fetch).

Perm is the set of access permissions, which is the powerset of *AT* (that is, $\{\emptyset, \{R\}, \{W\}, \{IF\}, \{R, W\}, \{R, IF\}, \{W, IF\}, \{R, W, IF\}\}$.

D is the set of domains. A domain is a tuple $<S, AMap, PMap, \ldots>$, where $S \subset V$ is the set of addresses in the domain, $AMap: S \rightarrow P$ is a function mapping the domain into physical memory, and $PMap: E \rightarrow (S \rightarrow Perm)$ is a function providing environment-specific permissions for the domain. *AMap* and *PMap* are controls of the domain; there may be other controls that do not pertain to this discussion. Domains do not overlap: no address appears in more than one domain's *S* (although some addresses may not be in any *S*).

$DC: V \rightarrow D$ is a function mapping addresses to their domains (a "domain classifier"). $DC(v)$ is the domain $<S, Amap, Pmap, \ldots>$ such that $v \in S$. Domains do not necessarily cover the address space, so DC may not be defined for some $v \in V$.

Data is the set of data representations available in the processor.

$Ptr: Data \rightarrow \{true, false\}$ is a function that is *true* for all representations of pointers.

$Addr: Data \rightarrow V$ is the function that determines the address of a pointer represented by a datum.

$CD: D \times D \rightarrow Perm$ is the function that defines cross-domain access permissions.

4.2 Instruction execution

All operations in our model take place during the execution of an instruction. An instruction may be a *read*, *write*, or *non-memory* instruction. "Executing" the instruction means determining the physical target addresses and generating appropriate faults; we will ignore other aspects of execution. Instructions are executed in the context of an environment *e*.

The common feature of all instruction executions is a procedure we will call $Translate(v, t)$. Given a virtual address $v \in V$ and an access type $t \in AT$, it either determines the physical address or signals a fault.

$Translate(v, t)$:

1. Let $d = DC(v_{target})$ be the domain $<S, Amap, Pmap, \ldots>$. If $DC(v_{target})$ is undefined, signal a *domain translation fault*.
2. Let $eperm = Pmap(e)$. If $Pmap(e)$ is undefined, signal a *domain access fault*.
3. Let $p = Amap(v)$. If $Amap(v)$ is undefined, signal an *address translation fault*.
4. Let $perm = eperm(v)$. If $eperm(v)$ is undefined, signal a *permission translation fault*.
5. If $t \notin perm$, signal a *permission violation fault*.
6. Use *p* as the physical target address.

We also define a procedure $CheckDomains(v1, v2, t)$. Given two virtual addresses *v1* and *v2* and an access type *t*, it performs the cross-domain permission test. It assumes $DC(v_1)$ is known to be defined.

$CheckDomains(v1, v2, t)$:

1. Let $d_1 = DC(v_1)$.
2. Let $d_2 = DC(v_2)$. If $DC(v_2)$ is undefined, signal a *domain translation fault*.
3. Let $perm = CD(d_1, d_2)$. If $CD(d_1, d_2)$ is undefined, signal a *cross-domain translation fault*.
4. If $t \notin perm$, signal a *cross-domain permission fault*.

All instructions perform an instruction fetch operation, which has two target addresses: $v_{current}$, the address of the current instruction, and v_{next}, the address of the next instruction to be executed. v_{next} will usually be the location immediately after $v_{current}$, except in the case of branches. An instruction fetch is executed in two steps:

1. Determine the physical address of the next instruction using $Translate(v_{next}, IF)$.
2. $CheckDomains(v_{next}, v_{current})$.

Non-memory instructions perform only the instruction fetch operation. Other instructions process either one or two target addresses in addition to performing their instruction fetch.

A read instruction has a single target address, v_{source}, the address to be read. It determines the physical source address using $Translate(v_{source}, R)$.

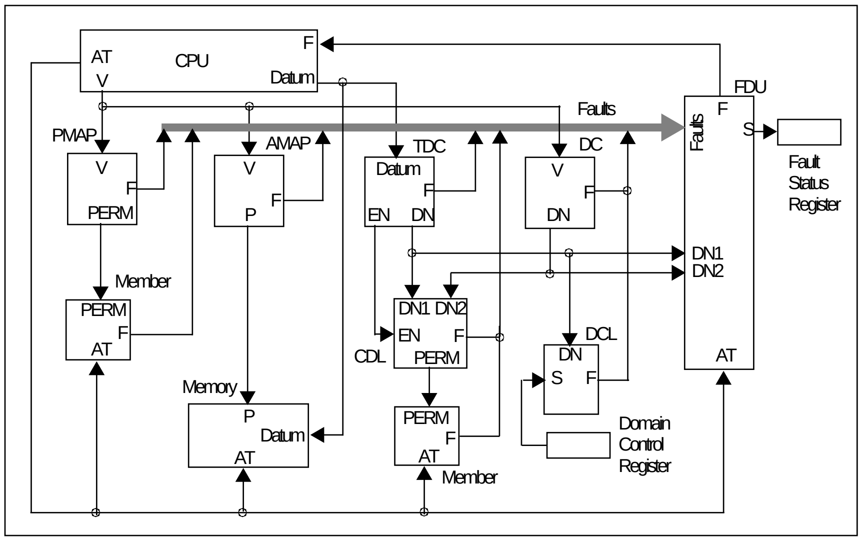
A write instruction has a target address v_{dest}, which is the address to be written into, and a *datum* which is to be written there. If they are writing a pointer, they have another target address $Addr(datum)$. The two-address procedure is as follows:

1. Determine the physical destination using $Translate(v_{dest}, W)$.
2. If $Ptr(datum)$ is *true*, $CheckDomains(v_{dest}, Addr(datum))$.

5. Hardware support

The address-management model can be implemented in hardware. In fact, implementations of a subset of it (namely the virtual-to-physical and address-to-permissions mappings) are common in standard memory management hardware. In this section, we explain how our broader model can be implemented.

Figure 1. Abstract hardware



5.1 Extended formal model

For better explication of the hardware implementation, we introduce a few new items into our model of address management.

DN is the set of *domain numbers*. A domain number is a small integer that represents a domain.

$Dnum : D \rightarrow DN$ is a function mapping domains to domain numbers. The operating system chooses appropriate assignments of domain numbers among the domains in use at any given time.

$DCR : DN \rightarrow \{true, false\}$ is a function that indicates the domain numbers whose domains' *Pmap*s are defined for the current environment. Let *e* be the current environment and *d* be the domain <*S*, *Amap*, *Pmap*, …> such that *Dnum*(*d*) = *dn*. Then *DCR*(*dn*) = true if *Pmap*(*e*) is defined.

Faults = {*domain translation, domain access, address translation, permission translation, permission violation, domain translation, cross domain permission*} is the set of faults.

A *fault status* is a tuple $<DN_1, DN_2, A, F>$, where DN_1 and DN_2 are domain numbers, $A \in AT$ is an access type, and $F \in Faults$ is a fault.

5.2 Hardware architecture

See figure 1 for a block diagram of the abstract hardware architecture, which is basically a data-flow representation of the model in section 4. This design is not suitable for direct hardware implementation, but it can be transformed into one that is.

Figure 2 shows the building blocks of the architecture. Each block represents some portion of the formal model.

- **DC** (domain classifier) accepts an address input *V* and produces two outputs: a fault *F*, and a domain number *DN*. If *DC*(*V*) is defined, *F* is set to *false* and *DN* is set to *Dnum*(*DC*(*V*)). Otherwise, *F* is set to *true* and *DN* is undefined.
- **TDC** (tagged domain classifier) accepts a *datum* as input and produces three outputs: a fault *F*, an enable *EN*, and a domain number *DN*. It combines a domain classifier and the *Ptr* and *Addr* functions.
  If *Ptr*(*datum*) is *true*, *EN* is set to *true*, and the domain classifier is used to find *Dnum*(*DC*(*Addr*(*datum*))). *F* and *DN* are the outputs of the domain classifier.
  If *Ptr*(*datum*) is *false*, *EN* is set to *false*, *F* is set to *false*, and *DN* is undefined.
- **AMAP** (address mapping) accepts an address input *V* and produces two outputs: a fault *F*, and a physical address *P*. It is the composite of all domains' *Amap*s. If *Amap*(*V*) is defined by some domain, *F* is set to *false* and *P* is set to *Amap*(*V*). Otherwise, *F* is set to *true* and *P* is undefined.

Figure 2. Building blocks

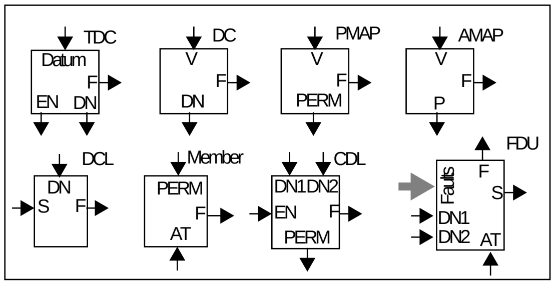
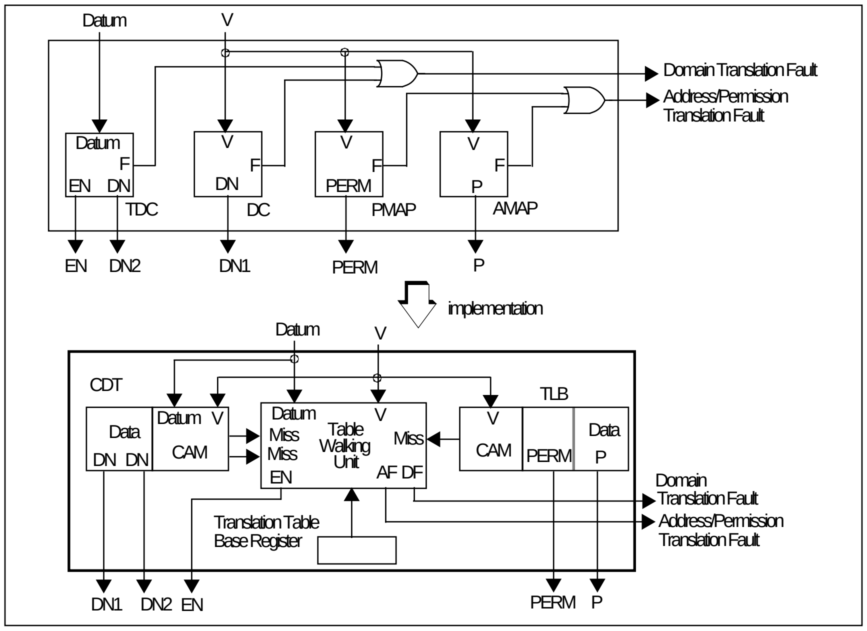


Figure 3. Mapping and domain classifier



- **PMAP** (permission mapping) accepts an address input *V* and produces two outputs: a fault *F*, and a set of permissions *Perm*. PMAP is similar to AMAP; it is the composite of all domains' *Pmap*(*e*), where e is the current environment.
- **DCL** (domain control logic) accepts a domain number *DN* and produces a fault output *F*, which is *true* or *false* depending on the contents of the domain control register input *S*. The register contains a bit for each possible domain number; *F* is determined by the bit corresponding to *DN*.
- **MEMBER** is an implementation of the relation $\not\in$. $F = AT \not\in Perm$.
- **CDL** (cross-domain logic) accepts two domain number inputs DN_1 and DN_2, and an enable input *EN*, and produces two outputs: a fault *F*, and a set of permissions *Perm*. If *EN* is false, *F* is set to *false* and *Perm* is set to *AT* (all permissions). If *EN* is true, then if $CD(DN_1, DN_2)$ is defined, *F* is set to *false* and *Perm* is set to $CD(DN_1, DN_2)$. Otherwise, *F* is set to *true* and *Perm* is undefined.
- **FDU** (fault dispatch unit) accepts one input for each member of *Faults*, as well as two domain numbers DN_1 and DN_2 and an access type *AT*. It has two outputs, a combined fault *F* and a fault status *S*. If any of the fault inputs are *true*, *F* is set to *true* and *S* is set to a representation of the fault status $<DN_1, DN_2, AT, fault>$, where *fault* is the highest-priority fault type whose corresponding input is *true*. If none of the fault inputs are *true*, *F* is set to *false* and *S* is undefined.

5.3 Implementation details

In order to produce a real hardware design, the abstract components in the previous section must be transformed into appropriate hardware structures.

The TDC, DC, PMAP, and AMAP modules can be implemented as a pair of content-addressable memories (CAMs) and shared table-walking logic (see Figure 3).

The **CDT** (cross-domain table) is a dual-ported CAM that accepts two addresses and delivers two domain numbers. The size of the CAM depends on the granularity of domain sizes.

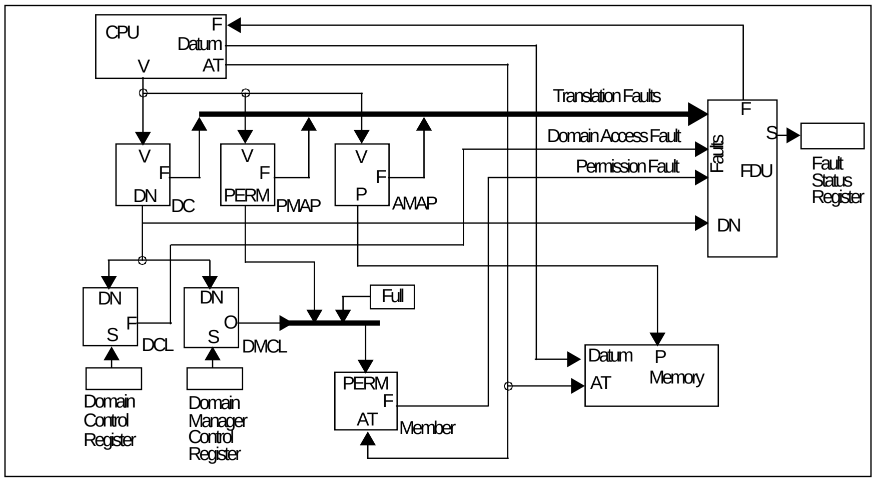
The **TLB** (translation lookaside buffer) is a single-ported CAM that accepts a virtual address and delivers a physical address and a set of permissions. This is a standard component of most memory-management units. The size of the CAM depends on the page size of the translation structure.

The **table-walking unit** is activated when either the CDT or TLB misses. It traverses a two-level data structure in memory to locate the needed information. The location of this structure is defined by the **translation table base register**. For a CDT miss, only the first level is used, because the domain information only resides in first-level entries. Both levels are always traversed for a TLB miss.

The table-walking unit also contains the logic (which we will not describe) that implements the *Ptr* and *Addr* functions.

Notice that because the PMAP and AMAP modules are implemented in a single CAM, the permission and address translation faults have been combined into a single output.

Figure 4. Simplified ARM 600 diagram



Similarly, there is only one domain translation fault output.

The cross-domain logic can be implemented in several ways. A CAM could be used to map one domain number to a vector of permissions on all other domain numbers, which would be demultiplexed by the other domain number. This approach becomes slower as the number of possible domain numbers increases, because the width of the data area of the CAM increases.

Another approach is to use a CAM indexed by both domain numbers that produces a single cross-domain permission. If the number of cross-domain operations expected in the system is small, the CAM will need relatively few entries, and because the data area is very narrow, the CAM will be fast.

The fully-general cross-domain model may not be necessary in practice. Often, an application is only concerned with pointers exported from or imported into its domain, or program counter entry to or exit from its domain. The other domain involved is not significant. To support only this functionality would simplify the cross-domain hardware to a set of four register/multiplexor pairs and a comparator. Two of the pairs would be associated with import and entry and indexed by the source domain number; the other two would be associated with export and exit and indexed by the target domain number. The bits in each register determine if a fault is generated. The comparator would be used to enable the other logic when the source and target domains are different.

The rest of the functionality is simple combinatoric logic.

- MEMBER is a simple boolean function.
- DCL is a multiplexor that selects a bit in *S* based on *DN*.
- FDU is a priority encoder and some routing logic to compose the value of the fault status register.

6. The ARM implementation

A subset of our hardware design has been implemented in the memory management unit of the ARM 600, a processor being developed by Advanced RISC Machines, Ltd. of Cambridge, England. In this section we describe the ARM 600 implementation, and how it can be extended in the future to implement the full address management model.

Figure 4 shows a simplified functional diagram of the ARM 600 MMU. It is identical to the general model of figure 1, with two major exceptions. There is no cross-domain logic, and there is support for *domain manager control*.

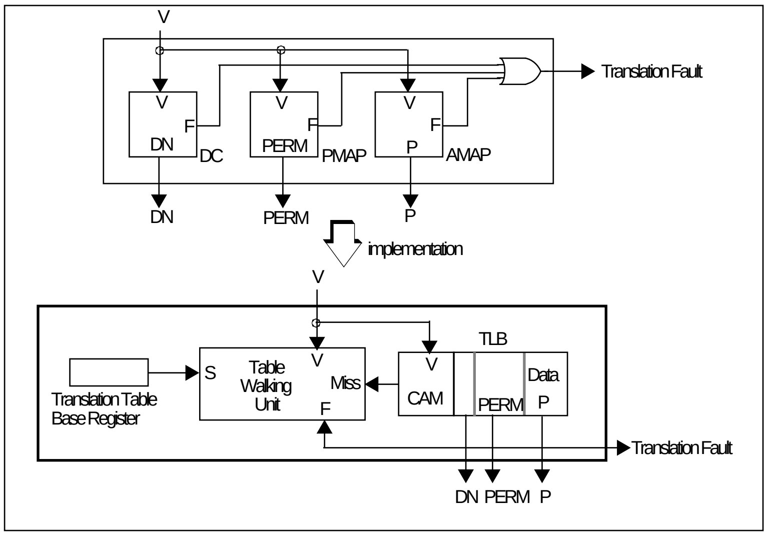
In general, the manager of a domain is not subject to the same permissions as its clients; instead, it has full permissions for its domain. To avoid having to switch the active permission mapping when switching to the manager’s environment (at fault time, for example), the hardware provides a fast way to turn on full permissions for a domain. The domain manager control register (DMCR) contains a bit for each domain number that indicates whether full permissions are in effect for that domain.

In the diagram, the DMCR is connected to a multiplexer (DMCL) that selects the bit in the DMCR corresponding to the current domain number. The bit controls a multiplexer that selects between the output of the PMAP and a constant encoding full permissions.

Figure 5 shows the implementation of the mapping portion of the ARM 600 MMU. It consists of a traditional TLB with the addition of the domain number, which is loaded from the first-level TLB entries.

Because we did not implement cross-domain logic in the ARM 600, we were concerned about the performance of alternative write-barrier implementations. To reduce the overhead of page scanning and the granularity of transac-

Figure 5. ARM 600 implementation details



tion locks, we allow the specification of access permissions on subpages. Each 4K page is divided into 1K subpages. Page-table entries contain a translation for the entire page, and a set of permissions for each subpage. This is similar to the approach taken in the RS/6000 [6].

Only one primary translation table is necessary. This is one of the major advantages of a single address space model with domain access control support. It is possible to switch between tasks without changing the TLB or translation table, because access to domains is controlled by the domain control register. When we designed the MMU, we decided to make the primary table large by traditional standards (16K vs. 4K) so that the secondary tables could be small (1K vs. 4K). This lowers the overhead for address-oriented services like transaction processing that require multiple permission mappings.

7. Conclusion and future directions

Some systems, such as MONADS [11] and POMP [5], have implemented advanced concepts such as persistent objects and garbage collection by making radical departures from traditional architectures. However, there are also software implementations that take advantage of "stock" hardware [1]. Such implementations are more economical since they don't need special hardware, but they are often complicated because stock hardware is designed to meet the traditional demands of virtual memory.

Our system model is close to tradition. Most of it can be implemented on common processors by changing only the MMU. The model enables new software directions while retaining the economy and compatibility of standard processor architectures.

In addition to completing work on an operating system based on our model of address management, we are interested in producing a complete implementation of our hardware model. We would like to get some experience with cross-domain barriers to determine the effectiveness of the cross-domain permission hardware. Another hardware direction is to examine other domain controls; for example, we have considered using domain controls to affect cache management.

8. Bibliography

[1] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices*, vol. 23(7), July 1988, pp. 11–20.

[2] M.P. Atkinson, P.J. Bailey, W.P. Cockshott, K.J. Chisholm, and R. Morrison. Progress with persistent programming. Technical Report PPR-8-81, Computer Science Department, University of Edinburgh, 1981.

[3] Thomas Atwood. Two approaches to adding persistence to C++. In *Implementing Persistent Object Bases: Principles and Practice*, Alan Dearle, Gail M. Shaw, Stanley B. Zdonik, eds., Morgan Kaufmann, San Mateo, California, 1990, pp. 369–383.

[4] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, vol. 21(4), April 1978, pp. 280–294.

[5] W.P. Cockshott. Design of POMP—a Persistent Object Management Processor. In *Third International Workshop on Persistent Object Systems* (Newcastle, Australia, 1989), J. Rosenberg and D.M. Koch, eds. Springer-Verlag, New York, 1989, pp. 367–376.

[6] Randy D. Groves and Richard Oehler. RISC System/6000 architecture. In *IBM RISC System/6000 Technology*, SA23-2619, International Business Machines Corporation, 1990.

[7] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. One-level storage system. In *Computer Structures: Principles and Examples*, D. P. Siewiorek, C. G. Bell, and A. Newell (editor), McGraw-Hill, New York, 1982, pp. 135–148. Originally in *IRE Transactions*, EC-11, vol. 2, April 1962, pp. 223–235.

[8] David A. Moon. Garbage collection in a large Lisp system. *ACM Symposium on Lisp and Functional Programming* (Austin, Texas, August 1984), pp. 235–246.

[9] J. Eliot B. Moss. *Nested Transactions: an Approach to Reliable Distributed Computing.* MIT Press, Cambridge, Massachusetts, 1985.

[10] Richard F. Rashid. Threads of a new system. *Unix Review*, vol. 4(8), August 1986, pp. 37–49.

[11] J. Rosenberg, D.M. Koch, and J.L. Keedy. A capability-based massive memory computer. In *Third International Workshop on Persistent Object Systems* (Newcastle, Australia, 1989). J. Rosenberg and D.M. Koch, eds. Springer-Verlag, New York, 1989, pp. 377–391.

[12] Eugene Shekita and Michael Zwilling. Cricket: a mapped, persistent object store. In *Implementing Persistent Object Bases: Principles and Practice*, Alan Dearle, Gail M. Shaw, Stanley B. Zdonik, eds., Morgan Kaufmann, San Mateo, California, 1990, pp. 89–102.

[13] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. *Proceedings of the ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference, ACM SIGPLAN Notices*, vol. 19(5), May 1984, pp. 157–167.

[14] Robert V. Welland. Managing the use of addresses in a computer system. Patent application, currently in preparation.

[15] Paul R. Wilson. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. Technical Report UIC-EECS-90-6, Electrical Engineering and Computer Science Dept., University of Illinois at Chicago, December 1990.

[16] M. Young, et al. The duality of memory and communication in the implementation of a multiprocessor operating system. In *Proceedings of the 11th Symposium on Operating System Principles*, November 1987.

[17] Benjamin G. Zorn. Comparative performance evaluation of garbage collection algorithms. Technical Report UCB/CSD 89/544, Computer Science Division (EECS), University of California, Berkeley, December 1989.