Xpriori Technical Paper

Digital Pattern Processing (DPP™) Memory Management

by Chris Brandin

Release 1.1

NOTE: In October 2003, Xpriori, LLC acquired NeoCore Holdings, LLC including all technology and patents. Any references to Neo, NeoCore or NeoCore Holdings, LLC technology or patents as such are now the property of Xpriori, LLC.

Xpriori, LLC
2864 S. Circle Dr.
Ste. 401
Colorado Springs, CO 80906
(719) 425-9840
www.xpriori.com

Version 1.1

Xpriori technology is protected by the following patents:
US Patent #5,742,611 (21 Apr 98)
US Patent #5,942,002 (8 Aug 99)
US Patent #6,157,617 (5 Dec 00)
US Patent #6,167,400 (26 Dec 00)
US Patent #6,324,636 (27 Nov 01)
US Patent #6,493,813 (10 Dec 02)
US Patent #6,792,428 (14 Sept 04)

Other U.S. and international patents pending.

The information in this white paper has been provided by Xpriori, LLC. To the best knowledge of Xpriori, it contains information concerning the current state of information processing technology. Xpriori, LLC disclaims any and all liabilities for and makes no warranties, expressed or implied, with respect to products described in this paper, including, without limitation, the implied warranties of merchantability and fitness for a particular purpose. No specific reliance should be made on the material provided herein without thorough investigation of the technology and its proposed application to specific circumstances. Product and technology information is subject to change without notice.

Introduction

This paper discusses a method employed to implement the Associative Memory Controller (AMC) function in a Digital Pattern Processor (DPP). Almost any numerical means to organize data can be used in a DPP. What is disclosed is a preferred method that has proved itself suitable for a wide variety of applications. Only a "minimum-case" implementation is discussed, in actual applications the memory management schemes can be more complex. This paper does not cover the processes involved in the creation of Icons (see the Appendix for references to papers about Icon generation). It is assumed that an Icon has already been created and discussions are limited to locating an association based on that Icon.

Memory Elements

The basic memory object in DPP is a "Quanta" and houses all data necessary to locate and manage an association. At a minimum Quantas contain the following elements (additional elements can be added to support other functions - such as behavioral sets, record locking, and multiple associations):

Forward Pointer
Confirmer
Association
Allocated Flag
Primary Flag

Finding Associations

The process to locate an association is as follows:

1. The Icon is split into two components – an Address and a Confirmer. The number of bits used for the address corresponds to the depth of the associative memory, the Confirmer is the remaining bits. For example, if a 64-bit Icon is used and the memory depth is 64K entries, then the Address portion of the Icon is 16 bits and the Confirmer is 48 bits.

2. The Quanta at the location pointed to by the Address portion of the Icon is read. If both the Primary and Allocated flags are set then processing continues, otherwise "not found" is returned.

3. The Confirmer from the Quanta is compared with the Confirmer portion of the Icon. If they match then the Association is returned, otherwise processing continues.
4. The Quanta pointed to by the Forward Pointer is read.

5. If the Primary flag is set then "not found" is returned, otherwise processing continues at step 3.

The Forward Pointer and Confirmer elements in a Quanta always add up to the same number of bits – the number of bits in the Icon. The Forward Pointer is the same size as the Address portion of the Icon and both Confirmers are the same size. This insures that Quantas are always the same size, irrespective of the depth of the associative memory. Basically this amounts to a singly linked list of entries that collide at an address where the collisions are resolved by comparing Confirmers as the list of entries is navigated.

Adding Associations

Memory use is deterministic; that is, no overflow area is necessary and the associative memory can be filled to 100% capacity. This is achieved by folding what would normally be the overflow area into the primary memory area. When an empty associative memory "Store" is allocated, all Quantas are linked into one big *doubly* linked "free" list (in this case the Association element is used as a reverse pointer – an unallocated entry obviously needs no association). A doubly linked list is necessary when a Quanta is unallocated because it must be removed from the free list when it is allocated. Allocated Quantas are singly linked (when more than one Quanta collides at a particular address or there are duplicates), while unallocated Quantas are doubly linked into one free list.

To understand how this works, we must examine the write process. When a new Quanta is added there are three possible circumstances to consider: the Quanta pointed to by the Address portion of the Icon is unallocated, allocated and primary, or allocated and not primary. The steps taken to add a new Quanta are as follows:

When the requested Quanta is unallocated:

1. Remove the Quanta from the free list.
2. Set the Allocated and Primary flags.
3. Save the Quanta Address in the Forward Pointer (so it points to itself).
4. Store the Confirmer and Association.

When the requested Quanta is allocated and primary:.

1. Get the next available unallocated Quanta from the free list.
2. Re-link the free list.
3. Set the Forward Pointer from the previous Quanta to the address of the new Quanta.
4. Set the Allocated flag in the new Quanta leaving the Primary flag cleared.
5. Set the new Quanta's Forward pointer to the previous Quanta's Address.
6. Store the Confirmer and Association.

When the requested Quanta is allocated and not primary:

1. Get the next available unallocated Quanta from the free list.
2. Re-link the free list.

3. Copy the previous Quanta to the new Quanta.

4. Navigate to the Quanta pointing to the previous Quanta and set its Forward Pointer to the new Quanta.

5. Set the previous Quanta's Allocated and Primary flags.

6. Save the previous Quanta Address in the previous Quanta's Forward Pointer (so it points to itself).

7. Store the Confirmer and Association into the previous Quanta.

In the last scenario, the Quanta that is not primary is moved and the new entry takes its place. In order to do this, the entire singly linked list of Quantas of which it is a member must be navigated in order to find the Quanta pointing to it. This Quanta "shuffling" occurs less often than one might suppose and therefore has little effect on performance. Also the number of Quantas in a singly linked list is generally low unless there are duplicates. In the case of long duplicate lists, there are a number of strategies that can be employed. Another data element (Reverse Pointer) can be added to the Quanta making all lists doubly linked. This solution increases memory use. There are other methods to deal with this issue that offer additional advantages; these are discussed in the paper entitled "Methods for Duplicate Entry Management in Digital Pattern Processors".

The write method described above assumes that no particular ordering of duplicates is required. For applications where ordering is desired, duplicate entries can be inserted at any location in a chain of duplicates. This is achieved by navigating through a list of duplicated until the desired insertion point is located and then writing the new Quanta as described in the second scenario ("When the requested Quanta is allocated and primary") above. In some applications (such as virtual Content Addressable Memories) duplicate Quantas are prohibited.

Additional Comments

Other functions, such as deleting and updating associations are implemented in ways that are obvious extensions to the functions described above, so they will not be discussed here.

The memory management scheme disclosed here results in nearly, but not quite, zero overhead associated with organizing the associative memory. Specifically, the total overhead amounts to two bits (the Allocated and Primary flags) per entry above the memory required to simply list the Icon/Association pairs. Usually, the flag bits are "stolen" from the Association field. There is a way to reduce this overhead to virtually zero. This can be done by establishing four free lists, each representing a quadrant of the total Store size. As a result, two bits can be removed from the Forward Pointer (because it now only has to point to ¼ of the total address space), and those bits can be used for the Allocated and Primary flags. The only drawback to this approach is that one quadrant may fill-up before the others, resulting in a "Store Full" condition before all entries have been used. The number of Quantas lost is always very low (well below 1%).

Conclusion

This memory management scheme was designed to offer the following characteristics:

- Speed – it takes an average of 1.5 Quanta RAM accesses to locate an association when a Store is 100% full. Because RAM is generally faster than hardware CAMs, performance is typically superior.

- Efficiency – In applications where logical collisions are impossible, the overhead necessary to organize associations is very low – in some cases virtually zero. In "symbolic" DPP operations, the overhead is essentially less than zero because storage requirements are related to Icon/Association pairs rather than Key/Association pairs (Icons can be much smaller than the Key data they represent). There are no "overflow" areas – memory use is deterministic. Although typically allocated on binary boundaries, Stores of any desired depth can be allocated (including odd sizes).

- Flexibility – This method allows Stores of any depth to be created with no effect on performance whatsoever. Stores are allocated in RAM at run time, so there are essentially no constraints on their number or type.

- Low Cost – RAM is used to contain data (instead of CAM) and memory use is very efficient resulting in the lowest possible cost.

Appendix

Related Papers

Brandin, Chris, "A Definition of Digital pattern Processing™"

Brandin, Chris, "Three-Dimensional Content Scanning Using DPP™"

Brandin, Chris, "Optimized Coding Methods for Icon Generation and Manipulation In DPP™"

Brandin, Chris, "Management of Duplicate Data Elements in DPP™ Virtual Associative Memories"

Brandin, Chris, "Behavioral Set and Field Descriptor Implementations in DPP™"

Direen, Harry and Phillips, Keith, "Finite Fields and Properties of the NeoCore Icon Generator, Associative Processing Unit, and Associative Memory Controller used in Digital Pattern Processing™"

# # #