Managing Large Address Spaces Effectively on the Butterfly

Peter Tinker

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF UTAH

UUCS-87-012

Managing Large Address Spaces Effectively on the Butterfly

Peter Tinker

Computer Science Department

University of Utah

Salt Lake City, Utah 84112

USA

Abstract

The BBN Butterfly™ Parallel Processor is a commercially-available multiprocessor which uses a memory management strategy based on a segmentation of the available memory. Using all of the memory of the machine efficiently is difficult because of the need to change the memory mapping dynamically. This difficulty discourages the use of the Butterfly for applications requiring a large address space.

This paper presents a scheme for using all of the Butterfly's memory with minimal impact through dynamic re-mapping of memory. Called *SAR-smashing*, it is appropriate for applications which require access to widely-scattered memory locations throughout a potentially large address space. The method was developed to support an OR-parallel logic programming system for the Butterfly which requires large stack data areas. Benchmark timing results for the memory access strategy are presented.

This material is based upon work supported by the National Science Foundation under Grant DCR-856000, and by an unrestricted gift from L.M. Ericsson Telefon AB, Stockholm, Sweden.

1. Introduction

The BBN Butterfly™ Parallel Processor is a commercially-available multiprocessor which uses a memory management strategy based on a segmentation of the available memory. Using all of the memory of the machine effectively is difficult because of the need to change the memory mapping dynamically. This difficulty discourages the use of the Butterfly for applications requiring a large address space.

This paper presents a scheme for using all of the Butterfly's memory with minimal impact through dynamic re-mapping of memory. Called *SAR-smashing*, it is appropriate for applications which require access to widely-scattered memory locations throughout a potentially large address space. The method was developed to support an OR-parallel logic programming system for the Butterfly which requires large stack data areas.

The paper begins with a brief description of the Butterfly hardware and an explanation of its memory management strategy, focusing on the way it maps virtual addresses supplied by the user to physical addresses capable of identifying any byte in the physical memory of the machine. Space considerations make this description incomplete, but hopefully not misleading. The limitations of this approach are made clear, and the proposed alternative approach is presented. Benchmark timing results are given for both approaches. The paper concludes with a short discussion of the use of the new approach in a logic programming system.

2. The BBN Butterfly Parallel Processor

The Butterfly is an MIMD, shared-memory multiprocessor composed of homogeneous processing elements connected by a non-blocking Omega network called the *Butterfly switch*. Each processing element, called a *processor node* or simple a *node*, contains an MC68020 microprocessor with an MC68881 floating-point co-processor, one or four megabytes of random-access memory, and a *processor node controller* (PNC). The PNC handles all memory requests by the 68020, using the switch to handle requests involving memory which is not local to the node. A Butterfly may comprise up to 256 such nodes, in increments of one node. More detail can be found in [3], [4], and [5].

3. Butterfly Memory Management

The Butterfly was originally designed to use the MC68000 processor, which supports only 24-bit addresses. This aspect of the original design persists in the current hardware; only 24-bit addresses can be used, even though the current 68020-based machine supports full 32-bit addresses. How, then, can the potential Gigabyte of physical memory on a fully-configured Butterfly be accessed? The solution adopted involves mapping 24-bit *virtual* addresses to 32-bit *physical addresses*. More detail on Butterfly memory management can be found in [1], [6], [2], and [8].

Using physical addresses, any byte of memory on the machine can be identified. A physical address has the form shown in Figure 3-1.

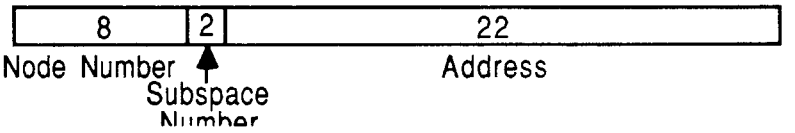

Figure 3-1: Butterfly Physical Address

The processor node number identifies the node on which the memory is located. The subspace number determines what mechanism the PNC uses to access the memory; for our purposes, only subspaces 2 and 3 are relevant. A subspace number of 2 indicates that the memory is local to the node which generated the address, and a subspace number of 3 indicates that the memory is not on that node. Local memory accesses do not need to use the switch and are therefore substantially faster than remote accesses (a 16-bit fetch from local memory to a register takes about 1.3 microseconds; the same fetch from remote memory takes about 6.3 microseconds[1]). A physical address for memory local to the node which generated the address has a subspace number of two: all other nodes which access this location use a physical address which is identical except that it has a subspace number of three. The 8-bit node number and the 22-bit offset combine to give a total address width of 30 bits, or a maximum of 4 Megabytes on each of 256 nodes.

[1]These figures are the result of timing 4000 consecutive in-line 16-bit fetch instructions to a machine register, with interrupts inhibited. Instruction-fetch time is included.

Physical addresses cannot be used directly on the Butterfly because the PNC, which handles all memory accesses, accepts only virtual addresses, which it maps into physical addresses. The 24-bit virtual address occupies a full 32-bit word and has the format shown in Figure 3-2.

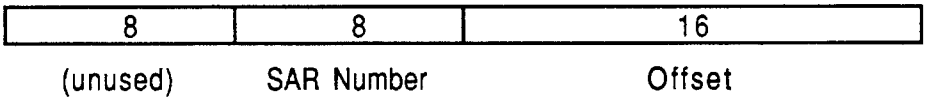

Figure 3-2: Butterfly Virtual Address

The most-significant 8 bits are unused, a legacy of the MC68000 heritage of the Butterfly. The least-significant 16 bits specify an offset of the memory from a base address. The base address is indicated indirectly by the *SAR number* field of the virtual address. This field specifies a *Segment Attribute Register* (SAR) to use in performing the virtual/physical address mapping. A SAR is a writable mapping register in the PNC which holds a *SAR value* of the form shown in Figure 3-3. To avoid confusion, note these three contexts in which the term SAR is used:

- A **SAR** is a physical register in the PNC. Some number of SARs are allocated to each Butterfly process.
- A **SAR value** is a 32-bit quantity of the form in Figure 3-3, which could potentially be stored in a SAR.
- A **SAR number** is a quantity in the range [0, 255] which specifies one of the SARs allocated to a process.

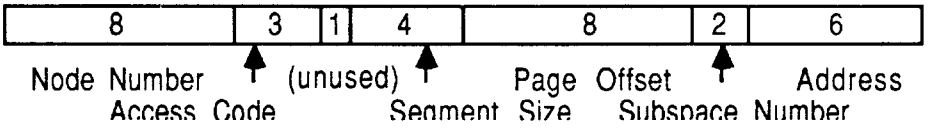

Figure 3-3: Butterfly Segment Attribute Register (SAR)

The node number indicates the node on which the memory is found. The access code and segment size indicate a protection mode and the size of the memory space, respectively, and are used to check for invalid accesses. The subspace number is the same as for physical addresses. The remaining fields specify an offset into the physical memory of the specified node and are used to determine the base address of the memory space indicated by the SAR.

Each SAR value corresponds to a *memory object*. These objects are manipulated by calls to the Butterfly's Chrysalis™ operating system and form the framework on which applications and

much of the operating system are built. An object is created by a call to the operating system function **Make_Obj**, which creates an associated *Object Attribute Block* (OAB) for the object. The OAB is located on the node containing the object and contains information about the current state of the object. In particular, the OAB contains the SAR value used when mapping virtual addresses to the object, and a reference count which is used to determine when an object can be removed from the system and its memory reclaimed, An object is specified to operating system functions by its *Object Identifier* (OID), which is unique system-wide. The OID of an object is returned to the user by **Make_Obj** when the object is first created. The maximum size of an object is 64 Kilobytes, corresponding to the width of the offset field in the virtual address.

The memory associated with an object can be accessed by calling an operating system function, **Map_Obj**, to retrieve the SAR value for the object, load it into a free SAR in the PNC, and return a virtual address corresponding to the first byte of the object's memory. The user can specify explicitly which SAR to use by passing its SAR number to the function, or can instruct the system to use any free SAR from those allocated to the user's process. The virtual address of any byte in the object can be computed by additions to the base address returned by **Map_Obj**. Any reference to such a virtual address is passed to the PNC by the 68020, where it is converted to its corresponding physical address and the memory access is performed.

There are 512 SARs in each PNC, of which up to 256 may be used by any single process. The total addressing capacity of a process is determined by the 256 SARs, each of which may indicate the start of a 64 Kilobyte object. Thus, at any time a process may only address 16 Megabytes of memory (256 SARs x 64 Kilobytes/SAR); on a fully-configured Butterfly with 4 Megabytes on each of 256 nodes, this accounts for only 1/64 of the available memory. Furthermore, the limit of 512 SARs per node makes it impossible to have several processes with large address spaces on each node.

If more than these 16 Megabytes are to be addressed, the values stored in the SARs must be changed to re-map the memory system. This can be accomplished using the operating system functions **Map_Obj** and **Unmap_Obj**. **Unmap_Obj** decrements the OAB reference count of the object and frees the SAR associated with the object by its **Make_Obj** call. As shown in Table

3-1[2], these functions are rather slow. Using them to re-map memory dynamically is clearly impractical.

Local Object:	827 microseconds
Remote Object:	1028 microseconds

Table 3-1: Memory Mapping Using Map_Obj and Unmap_Obj

4. SAR-Smashing: Fast Access to Butterfly Memory

With care, it is possible to make use of all of the Butterfly memory without undue concern for the number of SARs available or the time-consuming use of `Map_Obj` and `Unmap_Obj` to change the memory map. The scheme presented here, called *SAR-smashing*, has these features:

1. All of the available memory can be accessed by any process with little overhead for re-mapping.
2. As few as one SAR can be used to access all of the memory.
3. Changing the memory map can be accomplished in about 19 microseconds.

The price paid for these features is increased user responsibility for managing memory references.

To understand how SAR-smashing works, note that except for the subspace number, SAR values, are unique to each object, system-wide. It is the SAR value, not the OID, that determines the base address of the object's memory. It is the value in a particular SAR, too, that determines which object is accessed by a virtual address which uses that SAR. SAR-smashing simply changes the value which is stored in a SAR: doing so has the effect of mapping in the object represented by the SAR value, and virtual addresses which use that SAR now reference a different object. Instead of using virtual addresses with different SAR number fields (see Figure 3-2) that indicate different objects, the process can use virtual addresses that use the *same* SAR number field, and change the *value* in the SAR to change objects. SAR-smashing performs a "data context switch".

[2]Results in Table 3-1 are for 500 consecutive in-line Map_Obj and Unmap_Obj in alternation, using an explicit SAR number, with the OID and virtual address in machine registers, and with interrupts inhibited. The time for instruction fetches from local memory is included.

The following steps are taken to gain access to an object's memory:

1. Before any objects are created:
    a. A SAR is selected as the "smash SAR".
    b. A 64 Kilobyte object is created to serve as a table of SAR values. This object is mapped in normally by calling `Map_Obj`, and using a SAR other than the smash SAR. This table can hold up to 16K SAR values.
    c. A small object is created to hold the number of table entries (only two bytes are needed). This object is also mapped in normally. The number of table entries is initialized to zero.
2. When an object is created:
    a. The object is mapped normally using `Map_Obj`, specifying the smash SAR explicitly. `Map_Obj` returns the virtual address of the start of the object and, as a side-effect, places the SAR value for the object into the smash SAR.
    b. The SAR value for the object is extracted from the smash SAR. The subspace number field of the SAR value is set to three so that all PNCs which use the value later will generate remote physical addresses; the need for this is explained further below. Note that the SAR value could be extracted directly from the object's OAB without a `Map_Obj` call; we choose to use this more indirect method to take some advantage of the protection mechanisms used by `Map_Obj`.
    c. The SAR value determined in the previous step is placed in the table at the location specified by the counter's value. The entry counter is indivisibly read and incremented.
3. When the object is used:
    a. The SAR value for the object is retrieved from the SAR table and "smashed" (loaded) into the smash SAR.
    b. The virtual base address of the object is constructed. The base virtual address is the same for all objects: it has the smash SAR number in the SAR number field, and an offset of zero. The virtual address of the requested memory is computed using this base address.
    c. The requested memory request is made. The PNC will map the virtual address using the smash SAR, which will cause it to generate a physical address to the correct object.

Benchmarks indicate that loading the SAR value into the smash SAR can be accomplished in about 18.9 microseconds[3].

"Smashing" the SAR involves loading it with the SAR value without going through the usual mapping process used by `Map_Obj`. There is danger here for two reasons:

1. The object is never "unmapped" in the usual sense. If the object ever needs to be deleted, its SAR value must be loaded and the `Unmap_Obj` function called so the OAB reference count can be decremented properly.

---

[3]The benchmarks involved 100 executions of a loop with 5000 consecutive in-line SAR smashes, with the base address of the SAR table and the address of the smash SAR in machine registers, with interrupts inhibited. This time includes the time for instruction fetches (instructions are not in the cache) and for the calculation of the address of the location holding the SAR value.

2. Access to SARs is possible only when running in kernel mode, i.e., with special privileges. It is possible to corrupt the operating system while in kernel mode.

The SAR table need not be locked to prevent read/write conflicts when adding new objects' SARs. Since the table grows monotonically, the table entry counter acts like a stack pointer, ensuring that SAR values are always placed in free table locations. The Butterfly has provisions for the "test-and-set" operation needed to update the counter atomically.

Virtual addresses do not necessarily need to be calculated each time they are used; they can be maintained normally using address arithmetic. Note that the creation of virtual addresses does not involve an access to the indicated memory; it is calculated from the smash SAR number and the object offset only.

The SAR value loaded by `Map_Obj` from the object's OAB has a subspace number of two if the object is local to the node which maps it. For this SAR value to be used safely by any node when using SAR-smashing, the subspace number must be set to three. Using a subspace number of three means that access to the object by the PNC on the object's node takes the same amount of time as any other remote access, since the subspace number indicates that the switch must be used. To avoid this unnecessary delay, a SAR value which is known to refer to an object on the local node can have its subspace value reset to two before placing it in the smash SAR. If it is known that the object will *only* be used on its local node, the subspace value need not be set to three in the table, but can remain at two to maintain the efficiency of local accesses.

5. Using SAR-smashing for Logic Programming on the Butterfly

Modern logic programming compilation techniques use a stack-based abstract machine model such as the Warren Abstract Machine [9]. Typically, one stack (the so-called *local stack*) grows and shrinks by large amounts during program execution, while another (the *global stack*) generally grows with the execution. The global stack is typically much larger than the local stack. Neither data area is a stack in the strict sense of the term, since reference can be made to items which are deeply embedded in the area. In general, however, reference is most often made to items near the top of the area.

Logic programming using this execution model is clearly an application which can profit

from having a very large address space available to it. Furthermore, since reference can be made to items anywhere in the stacks, the memory must be available with minimal overhead for re-mapping. In our OR-parallel logic programming system, described in [7], we use SAR-smashing to minimize this problem.

Since we cannot use virtual addresses to refer to unique memory locations, we use *references* which are node-independent. References have the form shown in Figure 5-1.

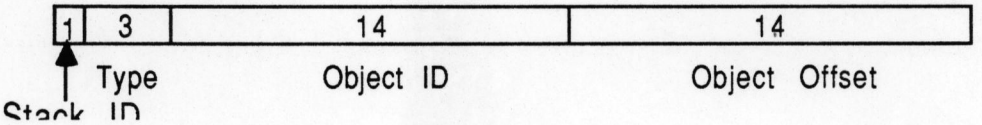

Figure 5-1: Reference Format

The most-significant bit is used to indicate on which type of stack the referenced item is located. The next three bits form a data type field. 14 bits are used to indicate the object in which the referenced item is located, yielding a potential maximum of 16 K objects. This field is used as an index into the SAR table. The least-significant 14 bits are used to locate the item in the referenced object. Each relevant item begins on a 32-bit boundary, so only 14 bits are required to locate the item in a 64 Kilobyte object. To compute the virtual address, the most-significant 18 bits are stripped away, the remaining 14 bits are left-shifted by two bits, and the result is OR'd with the smash SAR number, left-shifted 16 bits.

Such references are capable of locating any 32-bit item in a 1-Gigabyte physical memory. References can be passed from one node to another without change, and are used by extracting the object and offset fields, using SAR-smashing to perform the data context switch, and forming the appropriate virtual address. Typically, these steps do not all need to be performed, since several contiguous data items are generally used at the same time. Packing the references with data type information also saves many accesses to memory to determine the type of the referenced item. Table 5-1 summarizes some benchmark results of using these references[4]. The first figure indicates the time take to construct a virtual address from a given

[4]The results for creation of virtual addresses from references are from timings of 100 executions of a loop comprising 2500 consecutive in-line virtual address calculations, with both the references and the resulting virtual address in machine registers. The results for SAR-smashing from a reference are from timings of 100 executions of a loop comprising 500 consecutive in-line SAR smashes, with the references, the address of the smash SAR, and the base address of the SAR value table in machine registers. In both cases, interrupts were inhibited during the timing, and the reported times include instruction-fetch time.

reference; the second figure gives the time taken to extract the object ID from the reference and load the smash SAR with the correct SAR value from the table.

Creation of a virtual address:	6.4 microseconds
SAR Smashing:	23.5 microseconds

Table 5-1: Memory Operations Using References

6. Conclusion

SAR-smashing is a technique which enables a Butterfly user to make effective use of all of the memory on the machine. The cost for using SAR-smashing is an increased user responsibility for memory management, and a higher per-access cost for some accesses. It can be easily integrated with the standard Butterfly memory management system, so that access to often-used memory is as fast as the Butterfly will allow, while access to less-used memory encounters the SAR-smashing overhead. Benchmarks indicate that this method of dynamically changing the memory mapping process can proceed nearly two Orders of magnitude faster than using `Map_Obj` and `Unmap_Obj` to accomplish a similar re-mapping. Furthermore, the scarcity of SARs in the PNC is no longer a problem, since a single SAR is used to gain access to all objects.

Acknowledgements

I have gained much from discussions with Gary Lindstrom, which have fueled my search for ways to use the Butterfly effectively. I am also indebted to many BBN employees who have shared their intricate knowledge with me; I'm especially grateful to Michael Beeler, Will Crowther, Guy Fedorkow, Mindy Garber, Dave Mankins, and Bruce Moxon. Many thanks to Lal George for pointing out errors and inconsistencies in an earlier draft of the paper.

References

[1] BBN Laboratories, Inc.
Chrysalis Programmer's Manual
2.3.1 edition, 1986.
Contains many other documents.

[2] BBN Advanced Computers, Inc.
Tutorial for Programming in the C Language
1986.

[3] BBN Laboratories, Inc.
Overview
1986.
Also BBN Report No. 6148.

[4] BBN Laboratories, Inc.
Development of a Butterfly Multiprocessor Test Bed: Description of Butterfly Components.
Technical Report 5872, BBN Laboratories, Inc., March, 1985.
Quarterly Technical Report No. 1.

[5] BBN Laboratories, Inc.
Development of a Butterfly Multiprocessor Test Bed: The Butterfly Switch.
Technical Report 5874, BBN Laboratories, Inc., October, 1985.
Quarterly Technical Report No. 3.

[6] BBN Advanced Computers, Inc.
The Butterfly RAMFile System
1986.

[7] Peter Tinker and Gary Lindstrom.
A Performance-Oriented Design for OR-Parallel Logic Programming.
In *Proc. 4th Int. Conf. on Logic Programming.* May, 1987.
(to appear).

[8] BBN Advanced Computers, Inc.
The Uniform System Approach to Programming the Butterfly Parallel Processor
1986.

[9] David H.D. Warren.
An Abstract Prolog Instruction Set.
Technical Note 309, SRI International, October, 1983.