March 1988

Report No. STAN-CS-884197
Also numbered CSL-TR-88-354

The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation

by

D. R. Cheriton, A. Gupta, P. D. Boyle, and H. A. Goosen

Department of Computer Science

Stanford University
Stanford, California 94305

The VMP Multiprocessor: Initial Experience, Refinements and Performance Evaluation

David R. Cheriton, Anoop Gupta, Patrick D. Boyle, and Hendrik A. Goosen
Department of Computer Science, Stanford University, CA 94305

Abstract

VMP is an experimental multiprocessor being developed at Stanford University, suitable for high-performance workstations and server machines. Its primary novelty lies in the use of software management of the per-processor caches and the design decisions in the cache and bus that make this approach feasible. The design and some uniprocessor trace-driven simulations indicating its performance have been reported previously.

In this paper, we present our initial experience with the VMP design based on a running prototype as well as various refinements to the design. Performance evaluation is based both on measurement of actual execution as well as trace-driven simulation of multiprocessor executions from the Mach operating system.

1 Introduction

VMP is an experimental shared memory multiprocessor being built at Stanford University [12]. It follows the familiar model [6] of multiple processors connected by a shared bus to global memory with per-processor caches to reduce bus traffic. Its primary novelty lies in the use of software management of the per-processor caches and the design decisions in the cache and bus that make this approach feasible. Software control of the cache reduces the number of hardware data paths and simplifies the cache controller hardware, allowing a simpler, faster and more reliable multiprocessor workstation at a lower cost than conventional, hardware-intensive approaches. As described in this paper, software cache control also provides the flexibility of extended cache functions that support a number of advanced operating systems functions.

This project focuses on the problem of connecting multiple high-performance processors to a shared memory without significant performance degradation, rather than connecting a large number of processors of more modest capabilities [18] or not providing shared memory [24]. By high-performance, we mean the 10-20 MIPS microprocessors of modest cost available now and the 50-100 MIPS microprocessors of the near future. These fast multiprocessors require a fast cache for efficient execution and are able to execute the cache miss handling code quickly, making them suitable for this design.

The basic design and performance evaluation based on trace-driven simulations have been reported previously [12]. In brief, a VMP processor board contains a virtually addressed cache with a large (128-byte) cache block size [1] and a so-called *local* memory that stores the cache management code and data structures. On cache miss, the processor traps to a software miss-handling routine in the local memory, determines the physical address of the missing data and a cache slot to use (writing out the data if modified), initiates a block transfer of the data into a cache slot by the cache controller, and resumes execution when the block transfer completes. (The cache controller, system bus and the VMP system memory use *a* sequential access protocol to implement efficient block transfers.) A simple state machine called the *bus monitor* interrupts the processor to perform cache writebacks and invalidations to maintain cache consistency

[1]We use the term cache *block* to refer to the unit of transfer, addressing and invalidation in the cache. Some authors refer to this as the *cache* line.

according to the single-writer, multiple-readers protocol. Whether the bus monitor interrupts the processor for a given bus transaction is determined by the *action table,* an array with two bits per physical cache block frame. The two bits for a cache block frame indicate the action to be taken when a transaction for that frame is observed on the bus. Simulations suggested that cache block size in the 128-256 byte range provide cache miss rates under 0.2 percent for 256 kilobyte caches or larger, providing good performance for single processor execution. No data was available on consistency overhead at the time of this previous work.

In this paper, we present our initial experience with the VMP design, including the experience with designing and debugging the hardware, aspects of porting the V distributed system to the hardware and the cache performance of the system running this software. We also present a number of extensions to the design, both hardware and software, which we have identified since the original VMP paper. Finally, we present results from trace-driven simulations using memory reference traces of parallel applications in execution under the Mach operating system. We conclude that the design has significant performance and functionality advantages. However, additional research needs to be done in software techniques to reduce contention in order to realize VMP's full potential for very fine-grained parallel applications.

The next section describes the VMP cache management software and the functionality benefits of software cache control. Section 3 describes our experience in developing the prototype, porting the V distributed operating system and the actual cache behavior of the prototype. Section 4 presents the expected performance of the system, derived from trace-driven simulations. Section 5 describes a possible extension to handle simple forms of cache misses in hardware for processors with a significant amount of state which must be stored on a bus error. Section 6 compares our work to other relevant projects. We close with a summary of our results, identification of the key open issues, and our plans for the future.

2 VMP Cache Management Software

The VMP cache management software is central to the performance and functionality of VMP. It consists of several data structures and a set of routines to manage these data structures, all resident in the local memory of each processor. The cache *directory,* the main data structure, is a set of records, called Super *Cache Block Records (SCBR),* one per virtual memory page[2] represented in the cache. Each entry is efficiently accessible by virtual address and address space identifier, by physical address, and by set and bank numbers, as illustrated in Figure 1. Accesses by virtual or physical address use hashing with collisions handled by link fields in the SCBR. The SetBankMap contains a separate pointer for each slot in the cache.

The SCBR contains the virtual address for the virtual page it represents, its physical address in system bus memory, link fields for physical and virtual address space collisions, and N 32-bit fields, one per cache slot, where N is the number of cache slots per virtual memory page. These fields are packed in the same form as expected by the cache controller so there is minimal overhead in performing cache operations once the SCBR is located.

On a cache miss, the miss handler takes the following actions in the common case.

1. Get the SCBR for the faulting virtual address and address space identifier.

[2]The cache software can also be configured so that a SCBR instead represents a fixed portion of a virtual memory page. For example, an SCBR could represent 1 kikohyte of virtual space even though the virtual memory pages might be 4 kilobytes.

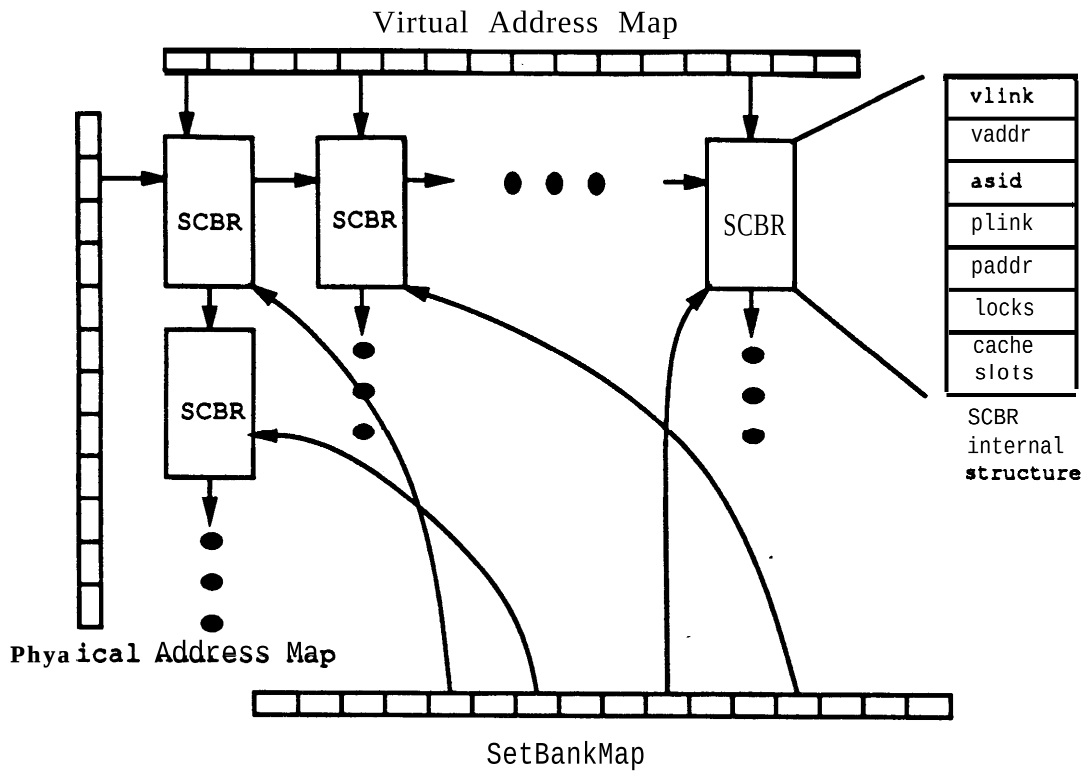

Figure 1: Cache Directory Data Structure

2. Check permissions.
3. Get an LRU slot for this address **from** the cache controller.
4. Write back the contents of the slot if **modified;** then invalidate.
5. Start the block move in of cache slot data.
6. Update SCBR and **SetBankMap** with new slot.
7. Restore state and continue execution.

The re-execution of the previously faulting memory reference is synchronized (and delayed as necessary) until the block transfer completes or **aborts.** If the block transfer is aborted, the faulting processor refaults when it attempts to resume. The cache miss handling *is **idempotent*** because the cache controller provides the same LRU slot each time we redo the cache miss and the data structure operations are idempotent. Thus, the cache management software handles redoing **a** cache miss exactly the same **as** if it was a first cache miss.

If the cache directory does contain an SCBR covering the faulting address, there are no **off**-board memory references other than the block transfer itself. Thus, the processor cannot have **a** fault while handling the cache **miss, and** as a result, when using (for example) the MIPS-X processor chip **[2],** it is not necessary to save the full processor state. Also, the simplicity of this case makes handling the miss entirely in hardware look feasible. (See Section 5.)

If there is no SCBR for the faulting address, the cache software invokes the operating system kernel to get the physical address of the virtual memory page being referenced and permissions on this page. It then allocates **and** initializes an SCBR and continues as described. It is possible that the virtual memory page is not in system memory, in which case the kernel suspends the process and handles the page fault. The cache management software performs the actual process switch on return from the kernel code in this case.

A cache management routine is provided to the operating system kernel to unmap a range of virtual addresses in a specified virtual address space. The unmapping is implemented as an exhaustive search of the cache directory for SCBRs in the specified range, with a few optimizations. First, a count of the SCBRs per address space is maintained and the search is terminated when the count goes to zero. Second, for a small number of pages to unmap, it looks up each SCBR individually using the virtually addressed hash table. Finally, the processor uses the "notify" bus transaction to signal other processors to perform the requested unmapping using a physical address corresponding (by convention) to the address space. Only processors with a' non-zero count for the address space AS set their action tables to be interrupted by notify transactions for address space AS. This procedure provides a simple mechanism for the kernel to invoke when it changes the virtual address space of a process, including when the address space is discarded on process termination. This scheme results in less bus activity than the approach we originally proposed [12].

The cache management routines are also invoked by bus monitor interrupts, indicating the need to write-out or invalidate cache slots to maintain consistency. These routines use the physical' address-to-SCBR mapping to locate the cache slot and its directory entry to update. If the FIFO of bus monitor interrupts overflows, an unlikely event with the current 512 entries in the FIFO, all the modified blocks in the cache are written back and the entire cache is invalidated. Full invalidation is required, rather than just shared block invalidation (as suggested previously in [12]), because the dropped interrupt might have been a notification of an unmapping operation, which was not used in the original design.

Software control of the cache makes several additional features straight forward, as described below:

2.1 Elimination of Page Tables

The large and sparse virtual address spaces required by many current and future applications has pushed some machine designs to use a large virtual memory page size (8 kilobytes or more) to minimize the size of the page tables. However, a large page size increases latency on a page fault, increases memory fragmentation and reduces the flexibility to use memory mapping techniques in the operating system kernel to efficiently copy data. A large page size also increases the lock and consistency contention overhead (as observed by Stonebraker [27] and Li [20] respectively) when the virtual memory page is used as the unit of locking and coherency.

The VMP processor cache and cache directory handle most virtual-to-physical address translations. Thus, a compact (machine-independent) operating system data structure is adequate for handling the few remaining *translation* misses without loss of performance, eliminating the need for conventional virtual memory page tables and allowing a fairly small (1 kilobyte) virtual memory page. In particular, explicit address translation is required only on a cache miss because the virtually addressed cache also acts as a translation lookaside buffer (TLB) on hits. On cache miss, the cache management software performs the address translation if any portion of the virtual page containing the miss address is in the cache because each SCBR covers an entire virtual memory page. Only otherwise does the cache management resort to accessing the operating system data structures. The cache management software does lazy reclamation of SCBRs so an SCBR with no valid slots is kept around until the virtual memory it represents is unmapped or until additional free SCBRs are needed, rather than being freed as soon as it has no valid slots.

The simualation results in Table 1 gives number of translation misses as a fraction of cache misses and as a fraction of all memory references when the cache block size is 128 bytes, and when the virtual page size is varied from 1 to 4 kilobytes. (A description of the three traces used for the data is provided in Section 4.) Actual translation miss measurements from single processor execution on VMP are given in Table 2.

Table 1: Translation Misses as Percentage of Memory References

Program	cache-misses (% all refs)	% cache-misses 1K	4K	% all refs 1K	4K
POPS	1.55	4.0	1.6	0.062	0.024
THOR	1.06	8.0	3.2	0.085	0.034
PEROUTE	0.14	17.0	6.8	0.024	0.010

The original cache software freed an SCBR as soon as it had no valid slots. However, this immediate reclamation produced much higher translation misses. For example, POPS with 1 kilobyte pages and immediate reclamation resulted in 60 percent of the cache misses being translation misses versus 4 percent with the lazy scheme.

The infrequent access indicated by the measurements of Table 1 makes it feasible to use far more compact data structures than conventional page tables.[3] As a particular realization of these benefits, the V kernel virtual memory system [9], now running on VMP, represents a virtual address space as a set of *bound regions*, ranges of addresses bound to portions of open files, as depicted in Figure 2. The open file descriptor in the kernel includes a hash table mapping

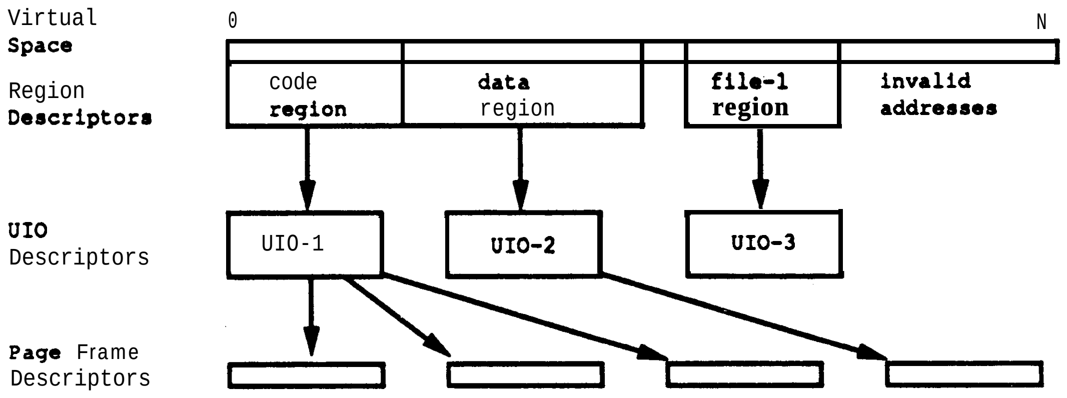

Figure 2: Compact Virtual Memory Data Structures

to page frames' containing data from that open file as well as parameters for accessing the file server. The file server (outside the kernel and usually running on another machine) maintains the mapping of open files to disk pages, removing that problem from the kernel. The space overhead is dominated by the 28-byte page frame descriptors for each 1 kilobyte page, giving a 2.8 percent overhead. For example, a fully mapped 10 megabyte address space requires 280 kilobytes of page frame descriptors. This space overhead would be reduced to 0.7 percent by going to a 4 kilobyte page size (because the page frame descriptor size is independent of the page size).

[3]This behavior also reduces bus traffic.

'This data structure corresponds to what has been called an inverted page *table* [8] except it applies to the cached file, not the address space.

The cost of an address translation using these data structures is estimated to be 10 microseconds per translation on average versus about 4 microseconds with a conventional page table. By these estimates, this technique effectively adds 4.8 nanoseconds to memory references on average, using Table 1 and assuming no full page faults. This is an acceptable overhead even with 75 nanosecond memory **access,** as on VMP. Moreover, if a translation miss results in a full page fault (requiring network or disk access to satisfy the fault) the performance penalty for the compact data structures is insignificant.

The 1 kilobyte page that is feasible with VMP significantly reduces the latency of a network page fault compared to incurred with a large page size. For example, paging over a 10 Mb network with 16 kilobyte pages, a page fault would require at least 15 milliseconds more than a 1 kilobyte page to complete because of network latency. Using 1 kilobyte pages does not preclude reading 16 kilobytes on a page fault in order to match larger, more efficient, network and file server block sizes.

The elimination of conventional page tables and the associated management software, including page table paging, is a major simplification of the operating system kernel virtual memory system. Reducing the operating system kernel size without loss of performance is a significant research goal, motivated by reliability, security and maintainability concerns.

2.2 Efficient Copy Support

Copying data incurs **a** significant overhead in operating system functions. Data is copied from one address space to another as part of interprocess communication operations. Data is also copied between kernel buffers and address spaces as part of file **and** device I/O and various virtual memory operations.

On a multi-processor machine with per-processor caches, a conventional processor copy operation incurs significant overhead on the processor and system bus overhead both directly as part of the operation as well as indirectly as a result of pollution of the cache. In particular, the copying processor (unnecessarily) traps the destination cache blocks into the cache (if not already present), and then overwrites these cache slots by processor move operations from another cache slot. In addition, both the source and destination blocks that are referenced as part of the copy may effectively (partially) flush the cache of its previous contents, leaving the cache occupied by the copied data even though it is unlikely to reference it subsequently.

Copy overhead is reduced in some operating systems by ***deferred copy***; the source pages are mapped read-only in place of the destination pages and a real copy only takes place if the data is subsequently modified, a **so-called** *copy-on-write* **fault.** Some measurements [15] indicate that the copy-on-write faults are relatively infrequent. However, deferred copy requires that the source and destination of the copy be page-aligned and that the data be a multiple of the page size. Also, deferred copy requires efficient remapping and changing of access on the memory regions being copied, which can be expensive with multiple processors and large page tables.

VMP provides efficient copy support in several ways. First, the relatively small page sizes allowed by VMP, as described in the previous section, make deferred copy by remapping feasible more often. Second, the cache management implements special routines for efficient deferred copy. In particular,

DeferredCopyMap(asid,vaddr,len,srcpages,destpages)

remaps the specified area of the address space to read-only on the srcpages and records the associated destpages **as** the destination in case a write access occurs. This routine invalidates the cache slots corresponding to the previous. mapping of this portion of this address space and modifies the SCBRs to indicate the new mapping and protection bit settings. Subsequent references to this region are read from the source pages (until modified) and written if modified to the destination pages. Thus, the copy operation is incorporated into the normal cache loading and writeback. Finally, this routine recognizes the special case in which the srcpages are designated as all zeroes, a common case for data area initialization. In this case, the cache slot is zeroed on demand by the cache management software and the bus transfer is eliminated. This mechanism reduces the frequency of a process trapping in cache slots that are subsequently overwritten.

Several aspects of this mechanism are noteworthy. First, the procedure invalidates the old mapping in all the processor caches, using the notify interprocess mechanism to all processors handling this address space. Second, the procedure only sets up mappings for those pages that are currently in the operating system page frame pool. Thus, some cache misses may result in page faults to get either the source or the destination page, or both. Third, the cache management is able to call the operating system to page in the source and destination pages as needed and retrieve the mapping information if discarded from the cache. Finally, the routine uses the normal inter-cache consistency mechanism to to maintain consistency of the copy with other processors as follows. When a processor faults on a cache block that is marked deferred copy, it loads the block from the source address and attempts to assert exclusive ownership on the block for the destination (physical) address. (A bitmask is maintained in the SCBR indicating the blocks that have already been copied from source to destination.) If another processor has already accessed this block, the attempt to assert ownership is aborted and the processor retries the transfer from the destination location (since the other processor will write its copy of the block out to that address). When a processor voluntarily writes back a modified cache block that is part of a deferred copy and updates the copy of the bitmask maintained in the page descriptor, it notifies all processors that contain an SCBR for this page of the update. Thus, every processor acquires copies from the right page and maintains its consistency with the other processors, once acquired, using the normal consistency mechanism.

This mechanism increases the performance of the copy, reduces the per processor bus traffic, and reduces the interactions with the operating system to implement virtual memory operations. The improvement is dependent on cache state and program access patterns. Interestingly, a common workstation scenario use of deferred copy, the initialization to zero of the address space, all block transfers are eliminated for the deferred copy if none of the modified blocks are forced out of the cache before the program terminates. This mechanism only adds a simple test for deferred copy to the common case of cache miss handling so the added overhead is insignificant. Additionally, the copy mechanism deals with all the complexities of cache consistency with no new hardware support.

Complementary to the above routine, the cache provides a physical page copy. It works by selecting a cache slot and repeatedly reading the next cache-block-size of the source data into the cache slot and writing it back to the next destination address, using the hardware block copier for each transfer. Before each block copy, it checks for the data already being present in the cache, in which case that slot is used directly, avoiding the copy in as well as ensuring a consistent copy. If a cache block happens to be exclusively owned by another cache, the block copy in is aborted by the bus monitor in the other processor and written out, using the normal consistency mechanism. The block copy is then retried by the copying processor. The block copy out also forces any copies of that data from any processor caches, including the local processor.[5]

[5]The block copy out is aborted and retried after write out and invalidation if another cache holds a private copy.

The benefits and uses of this copy support are the subject of future research. However, it is evident that these techniques reduce the overhead on the cache, bus and processor compared to a conventional copy. In essence, both this section and the previous section describe the benefits of moving some operating system functionality to the processor cache management. This migration of functionality relies on the software controlled caches of VMP; the complexity of the implementation is not a problem in software but infeasible in hardware.

2.3 Process-level Cache and Lock Control

The cache management software implements a cache flush and unlock operation, made available to the process level by a system trap, that allows a process to flush a portion of the cache corresponding to virtual addresses in its address space. The flush may specify any one of:

writeback - just writeback any modified data.

read-only - downgrade to read shared, writing it out if modified.

invalidate - remove from cache entirely.

Optionally, the flush operation may also specify a lock to release? Note that the flush-and-unlock operation only affects the cache out of which the process is currently executing. The provision of this operation allows the process level to heuristically influence the cache behavior as an optimization, using its knowledge of accesses to shared data. For example, if a process completes the update of a shared data structure and it expects the next access to be made -by a different process running on another processor, it can use the trap to flush the modified data from the cache (and unlock the data structure), thereby reducing the probability of cache contention when this subsequent access occurs.

The flush operation may be used explicitly by the programmer or it may be inserted by an optimizing compiler for a parallel programming language. A similar approach has been proposed by researchers at DEC[7] as an alternative form of software-controlled cache consistency. However, in our design, the compiler cache operations are only optimizations; the cache consistency is guaranteed by the low-level cache management operations. Consequently, the VMP design allows one to realize the performance benefits of advanced compiler technology and programmer knowledge of data sharing, while relying on a small amount of mechanism for correctness, namely the VMP hardware and cache management software. (The cache management software is less than 2 percent the size of our compiler.) Moreover, the VMP design allows the use of conventional programming languages augmented with parallel constructs, as opposed to the safe languages that are otherwise required.

The cache software also implements an operation to flush data within a specified range from all caches for DMA operations; this operation is only accessible within the kernel.

2.4 Implicit Locking Support

Implicit locking support for data base management systems has become of considerable interest recently [8, 27). By *implicit locking,* we mean automatic locking of data based on memory references. That is, a record is read-locked if read, and write-locked if written, without the programmer having to explicitly lock data.

"The operating system supports locks that do not go through the cache, as we proposed in our earlier paper [12].
[7]Private communication from Mike Powell.

Implicit locking can be implemented in the VMP cache management software with no additional hardware support. The first reference to a record results in a cache miss. The cache miss handler acquires the necessary lock. On read reference, the cache entry is set as read-only so a subsequent miss causes the write-lock to be acquired. Alternatively, the cache block can be acquired with a write lock and downgraded to a read lock later if the cache block is not modified. The locking on each cache block is summarized in the SCBR for each virtual memory page represented in the cache as well as in the kernel virtual memory data structures, similar to the design used in the 801 system [8]. The cache block size of VMP is close to that suggested for data base physical locking, making the mechanism a good match. A smaller cache block size would produce excessive locking overhead; larger blocks would be result in more conflicts between physical and logical locking behavior.

In contrast to this simple software scheme, Stonebraker [27] outlines hardware support that is required to support implicit locking using a conventional virtual memory design (without consideration of multiprocessor issues). The IBM RT and 801 workstations also incorporate significant hardware support for 128-byte locking. We achieve the goals of these other designs with no specialized hardware and, in comparison to the RT, far better performance, at least on non-miss cache references.

2.5 Debugger Data Breakpoints

The cache management software can also be used to support the debugger setting breakpoints on data references to particular locations. When such a breakpoint is set, the cache block is marked as breakpointed in the virtual memory page frame and removed from all processor caches (using the unmapping mechanism). When an address within a breakpointed cache block is referenced, the address is checked to determine if it matches a breakpointed address. If so, the process is placed under the control of the debugger. Otherwise, the protection on the cache slot is changed to allow access, the process is single-stepped through the reference and the protection on the cache slot is reset.

In this way, the debugger provides data reference breakpointing without special hardware at no performance overhead for references to non-breakpointed blocks (other than a check on a cache miss to determine if the cache block is breakpointed). The use of cache blocks rather than virtual memory pages provides a finer granularity for trapping and checking these references, reducing the performance penalty for running in this mode.

In general, the software implementation of cache management means that the software can effectively gain control, as needed, on every memory reference within blocks of addresses corresponding to the size of cache blocks. Our original motivation for this design was to simplify the hardware data paths, leading to a simpler, faster, less expensive and more reliable multiprocessor architecture. However, as we have described above, the cache management software can be extended to implement functionality that significantly improves the overall system design, performance and functionality.

3 Experience with the Prototype

A single processor VMP (wire-wrap) processor board is now running the V distributed operating system as a single node in our Ethernet-based cluster of DEC Microvax and SMI SUN workstations. At the time of writing, the first printed circuit layout board version of the the processor is running. We plan to build at least 15 processor boards and configure 3 5-processor nodes in 1988. This hardware will provide a basis for further experimentation and measurement with VMP running the V distributed system with a variety of parallel applications.

3.1 Hardware Development Experience

The prototype processor board contains 147 ICs, of which 37 are memories, 25 are bus interface logic, 48 are PALs and 37 are miscellaneous logic, including the processor (68020), floating point unit (68881) and 14 drivers for these two major ICs to connect to on-board busses. The components require approximately 3200 connections. Overall, the processor board is relatively simple and should be further simplified by the larger memory chips, larger PALS, and integrated bus interface logic expected in the future.

As further evidence of the simplicity of the design, the processor board required only 3 corrections (not counting wirewrapping mistakes) in order to pass the diagnostics. More impressive, there have been no errors in the board detected since that time, during which we have ported the V operating system to the machine and run various applications, as reported below. The printed circuit board version was designed and laid out using the DEC Western Research Laboratory CAD system, fitting on one quadruple-size VME board.

It is difficult to quantify the design and implementation time accurately because we were developing the hardware facilities for this work concurrently, there were many unrelated tasks that interfered with our progress and a lot of software effort was required and performed concurrently. In fact, the time to get the node useful for software development and porting of the operating system was dominated by the time to develop a PROM monitor that supported network boot loading and the writing of the diagnostics. However, the time to implement the hardware and software was less than 2 man-years. In addition, we believe that the relatively low part count for the board, the ease of debugging and the apparent absence of further problems speaks well for the design.

We are using a commercially available VME memory board that supports sequential memory access plus commercially available VME system controller and Ethernet boards.

3.2 Operating System Porting Experience

The V distributed operating system [10] was ported to the VMP node in approximately 7 man-months of effort although some of the basic device support was performed as part of developing the PROM monitor.

The cache management software was relatively easy to develop and debug. Considerable time was spent reorganizing the code to optimize the handling of simple cache misses. Once a few clerical errors in the software were fixed, the cache management has performed without problems. Interestingly, much of the cache software appeared to have been exercised at the point that kernel initialization was completed. As one (unexpected) example, setting breakpoints in the kernel code from the PROM monitor caused the cache management to perform "assert-ownership" operations on the referenced instruction cache block to gain write access as part of inserting the breakpoint. Even the consistency routines invoked by the bus monitor were exercised by the kernel virtual memory management software because the same physical memory was accessed by different virtual addresses.

A significant positive aspect observed during the port was the limited amount of machine-specific virtual memory code required for VMP. VMP requires 540 lines of machine-specific memory management code versus 1488 lines of code required for the DEC Microvax. The major simplification arises because the VMP implementation has no page tables, following the design described in Section 2.1.

The V kernel as ported also supports multiple processors. Unfortunately, we do not at this time have multiple processor boards with which to test the software. However, the V kernel currently runs on the &processor Firefly [28], an experimental DEC workstation, so we have some confidence in the basic adequacy of the software. (Also, the cache management software has been used in modified forms to exercise the various consistency interrupt mechanisms, with the one processor generating consistency interrupts for itself.) Further tuning and refinements to the software are planned. However, the system is running well enough to collect some measurements, as presented below.

3.3 Performance Measurements

Measurements were collected on the prototype single-processor node using on-board hardware counters to count memory references and the cache management software to keep statistics on cache miss behavior. As configured, the V kernel uses *a* 1 kilobyte virtual memory page size and *a* 128 byte cache block size. The cache is **4-way** set associative and 128 kilobytes in size. Table 2 presents measurements of the cache miss behavior as a percentage of memory references for several different applications, including the operating system memory references incurred during the execution of the application. The **"misc"** row lists measurements from running *a*

Table 2: Misses as Percentage of Memory References

Program	cache	instruction	data	translation
	misses	misses	misses	misses
misc	0.07	0.036	0.034	0.0043
LaTeX	0.05	0.024	0.026	0.0004
compiler	0.04	0.024	0.016	0.0020
multiprog	0.08	0.034	0.046	0.0029

collection of various **small** programs including a **file** listing, **file** copy, time query, user query and change directory, a total of 41 million memory references.[8] This row indicates the miss rate one could expect when executing a typical sequence of interactive commands on a workstation. The next applications, **LaTeXand** compiler, are large programs, *a* text formatter and a C compiler respectively, each over 700 kilobytes and typical of compute-intensive workstation applications. **LaTeXwas** measured formatting this paper, resulting in over 264 million memory references. Compiler is a **6-pass** C compiler (including the linker) compiling a modest size program (resulting in 63 million memory references). Multiprog is the compiler and **LaTeXrunning** concurrently, giving an indication of VMP behavior under multiprogramming.

We first observe that the cache miss rate is in fact substantially lower than predicted from our previously reported trace-driven simulation [12]. There are several factors that may explain this discrepancy. First, we conjecture that the traces used previously were too short to get beyond the cold start effects, given the large size of cache. Each program execution represented in Table 2 generated more than 60 times as many memory references as the instruction traces that we used previously. (This conjecture raises a concern in general about conclusions drawn

[8]The measurements were made with the 68020's instruction cache disabled so that the cache fields all references.

from the relatively short traces used in most cache simulations.) Second, the measurements were made running under the V kernel, as compared to our previous traces which were taken from a system running VMS. We conjecture that the smaller size and more modular structure of the V kernel reduced the penalty of operating system memory references. In particular, the "hot spot" code in the V kernel is concentrated in the interprocess communication and process switching, a very small amount of software. Finally, applications in our measurements were accessing files remotely from a file server (rather than executing a local file system) using the V network interprocess communication, so operating system execution was further reduced. However, this configuration is realistic and dominant in many environments that make extensive use of diskless workstations, including our computing environment.

All the translation miss rates are close to those derived from our traces in Table 1 for 1 kilobyte pages except for LATEXand the compiler (which are much better).

With the processor performance normalized so that processor performance with no cache misses is 100 percent,[9] as done previously in [12], this level of cache misses (averaged over our 4 measurements to 0.06 percent) gives an average processor performance of approximately 97 percent. We do not have the hardware or software operating at "full speed" at the time of writing, however, our previous elapsed time estimates for cache misses of 17.5 microseconds when replacing an unmodified cache block and 20.5 microseconds when replacing a modified cache block seem to be achievable.

In general, these measurements indicate the validity of our previous predictions of cache performance and show real VMP performance on typical workstations loads. This performance is essentially unaffected by running multiple applications simultaneously (especially when the cache is changed to .5 megabytes) because the cache is not flushed on a context switch. It remains to consider the performance of parallel (shared memory) applications on VMP.

4 Trace-Driven Multiprocessor Cache Simulations

The expected multiprocessor behavior of the VMP cache design was investigated using trace-driven simulations, pending the availability of a multiple processor configuration. In particular, we studied the effects of cache block size on the miss ratio of the cache and the traffic generated on the bus. The traffic generated is separated into two components: (i) that due to the intrinsic miss rate of the memory references and (ii) that due to the interference between multiple processors We also discuss the impact of the application structure on cache performance.

The traces used for our analysis are derived from a multiprocessor extension of the ATUM [4] address tracing scheme. The traces were collected from a Digital VAX-8350 multiprocessor consisting of 4 VAX CPUs connected to shared memory. Each trace is approximately 3.5 million references long and includes both user and operating system references [1]. The three parallel applications from which we use traces are (i) POPS [17], a parallel implementation of rule-based systems; (ii) THOR, a parallel logic simulator; and (iii) PEROUTE [23], a parallel router for standard cells. While the first two programs exploit parallelism at a fine granularity (the individual tasks consist of a few hundred instructions), the PEROUTE program exploits parallelism at a much coarser granularity (the individual tasks consist of tens of thousands of instructions). As we show later in this section, the smaller number of synchronization instructions executed by PEROUTE make a big difference in the amount of coherence traffic that is generated. We also

[9] $perf\ otmance = (1 + MissRatio * AverageMissCost * RefsPerInstr \bullet InstrExecutionRate)^{-1}$ From MacGregor [21]: RefsPerInstr=1.2 and $InstrExecutionRate = (7 clocks/instn * 60 nsecs/clock)^{-1} = 2.4$ MIPS

note that the programs (and thus the traces) are not optimized to run on the VMP architecture, for example, they are not structured to make use of the large cache block sizes, and consequently the traces are somewhat pessimistic in predicting the performance of VMP.

Table 3 gives the basic characteristics of the three traces used here. The columns, in order, give the names of the applications, the total number of references in the trace, the number of instruction fetch references, the number of data read references, the number of data write references, and the number of interlocked read-modify-write data references.

Table 3: Trace Characteristics (in thousands)

Program	total	ifetches	dreads	dwrites	ilocks
POPS	3142	1624	1257	261	19.7
THOR	3222	1456	1398	368	24.2
PEROUTE	3508	1833	1266	408	1.1

To explore the effect of VMP cache block size on performance, in Figures 3 - 5 we show the number of bus transactions generated (the sum of read transactions, write transactions, and assert-ownership **transactions**[10]) as a function of the cache block size. Note that simply counting bus transactions is a reasonable performance measurement only to the extent that the cost of a bus transaction is independent of the **transfer** unit. However, the actual cost of a bus transaction is B + CL, where B is the basic cost of starting and managing a bus transaction, C is some constant based on the bus width and speed, and L is the size of the data transfer. Nevertheless, taking the bus transaction cost as independent of cache block size is a reasonable approximation in VMP for two reasons. First, the CL factor can be made small by using a wide bus, fast serial access memory, and a fast serial bus protocol and block copier. In particular, the bus time can be halved by doubling the width of the system bus because essentially all bus transfers move blocks of data. Second, the factor B for **VMP** is quite large, as moat bus transactions occur **as** part of a software trap. This large value **for** B further **reduces** the **impact** of the differences in the CL factor. For example, assuming a **64-bit** wide bus that takes **100ns** for bus arbitration, **200ns** for first word transfer (read/write), **50ns** for each subsequent word transfer, and 4000ns to handle the trap plus startup **costs,** the elapsed time to the completion of transfer is 4650ns for a 64 byte cache block size, **5050ns** for a 128 byte block size, and 4250ns for assert transactions. For the above reasons, in the following discussion we only present data on number of bus transactions required **and** not on **absolute** time required.”

Figures **3, 4,** and 5 present the number of bus transactions required by the three applications as a fraction of the total number of memory references in the trace. The total number of transactions generated (***totalTUs***) is further split up into two components. The first component **is** the number of intrinsic transactions (***intrinsicTUs*****).** This is the number of bus transactions that would be generated in the **absence** of any coherence traffic. The second component is the number of coherence transactions (***coherenceTUs*****).** This is the number of transactions required solely to keep the caches coherent. The graphs also show the total number of cache misses that were observed for the traces. Note that the total number of bus transactions is larger than the total number of misses because it is quite possible for a miss to result in multiple bus transactions. For example, a miss can result in the write-back of dirty **data** and the read of

[10]**While both read and write data transactions require transfer of data over the bus, the assert-ownership transactions do not.**

[11]**Another reason for using bus transactions is that the reader can substitute his own** estimates **for how long individual bus transactions take as** a **function of cache block size, and thus evaluate** tradeoffs **corresponding to his cost model.**

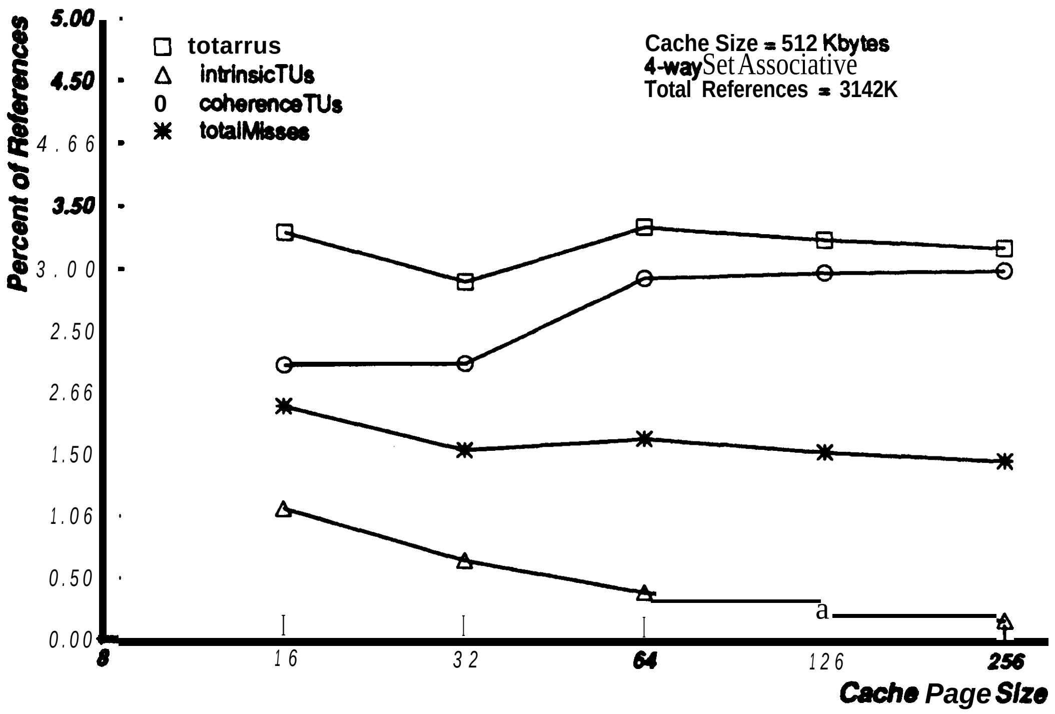

Figure 3: POPS: Bus transactions vs cache block size.

clean data.

Several observations can be made from the data presented in the graphs. First, the total number of TUs and total Misses is acceptably low for PEROUTE but rather high for POPS and THOR. PEROUTE has a lower rate of intrinsic TUs and, more significantly, a much lower rate of coherency TUs. The latter reflects, in part, the coarser grain of parallelism in PEROUTE. It is also a consequence of the low number of interlocked references in PEROUTE (see Table 3) as compared to the other applications. PEROUTE achieves this low number by allowing non-exclusive access to shared data structures at the cost of tolerating some error in the final solution. However, the large amount of coherency traffic relative to intrinsic traffic in all programs is to be expected, given the large size of the per-processor cache.

Second, increasing the cache block size does not decrease the total number of misses and TUs significantly except for PEROUTE despite the fact that the number of intrinsic bus transactions decreases significantly for **all** three **programs.** Thus, the number of ***coherency-induced*** misses and coherency TUs is increasing. The number of coherency-induced misses is roughly indicated by the difference between the total misses and the intrinsic TUs. Here, we use the intrinsic TU data points as an approximation for the number of ***intrinsic misses,*** misses in the absence of contention. Note that $intrinsicTUs = intrinsicMisses * (1+$ *probability-Of-A-Mali,Fed-* ***Replacement-Cache-Slot***). Note that the probability of a cache slot being modified increases with increasing block size. Thus, the number of intrinsic misses is less than the number of intrinsic TUs and decreases faster than the number of intrinsic TUs with increasing block size.

The increase in the number of coherency-induced misses with increasing block size in POPS and THOR roughly quantifies the effect of a larger cache block size increasing the likelihood that multiple unrelated shared data objects reside in the same cache block. When different processors perform writes to these coresident objects, coherence traffic is generated that would not arise if the objects were on separate cache blocks and, because this coherence traffic results in invalidations, additional misses result as well. Using a programming system that placed

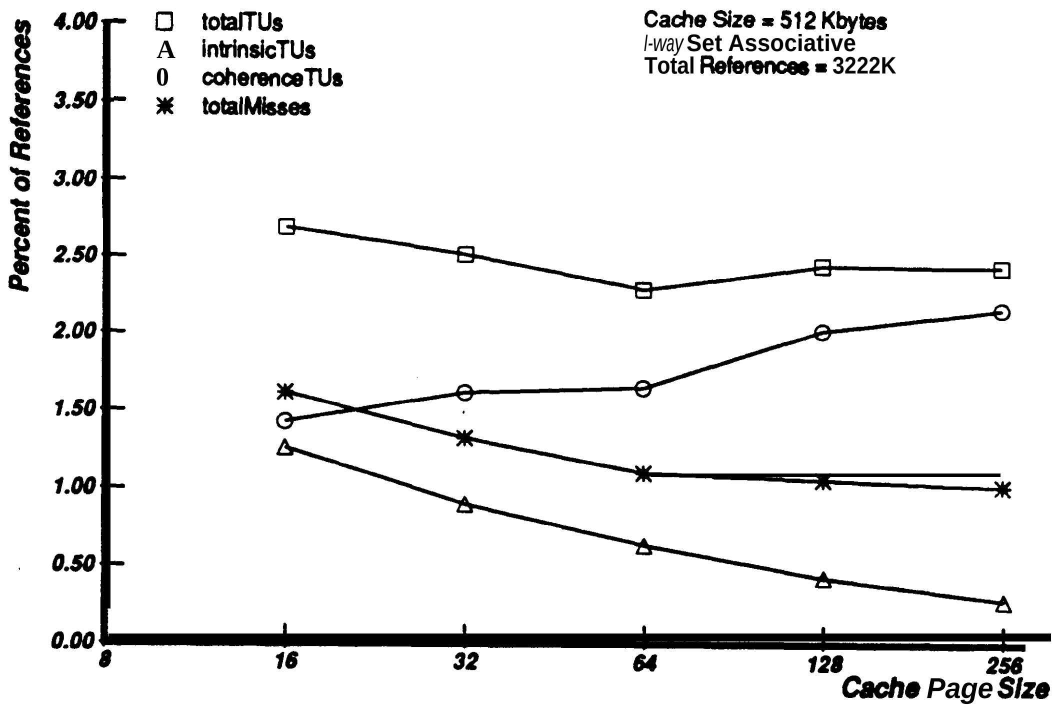

Figure 4: THOR: Bus transactions vs cache block size.

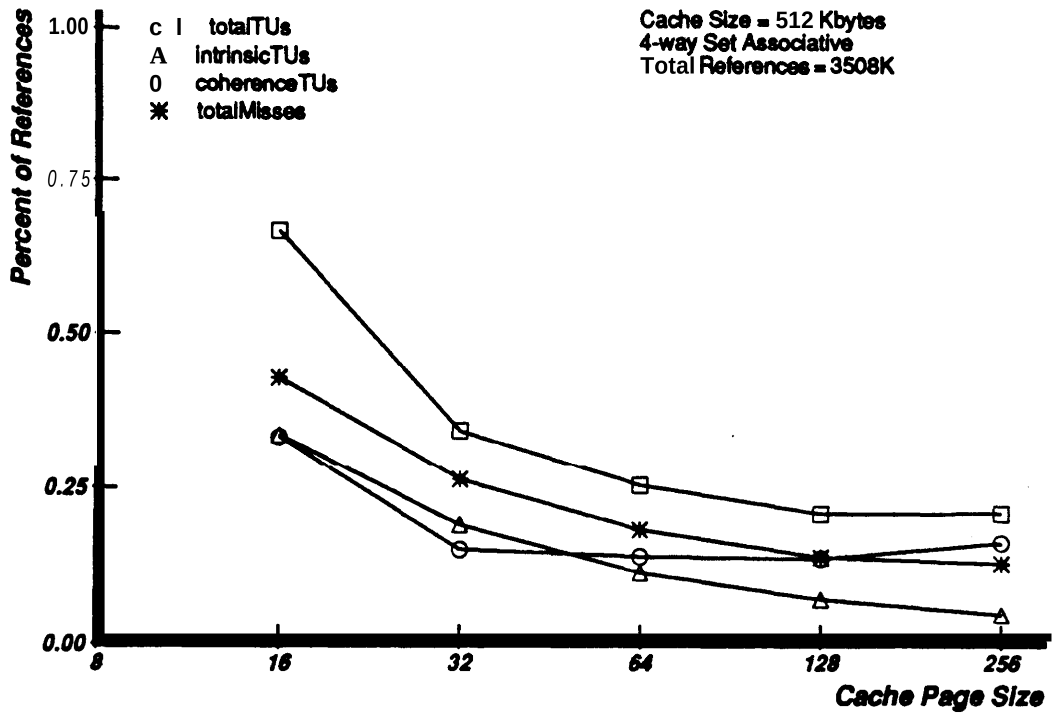

Figure 5: PEROUTE: Bus transactions vs cache block size.

unrelated shared objects on the different cache blocks would eliminate this effect. Thus, for example, in POPS, total Misses should be approximately 0.8 percent of references rather than 1.3 percent running with a 256 byte cache block size and intelligent cache block placement of shared data objects. (On the negative side, it may increase the number of intrinsic misses and TUs because the execution occasionally benefits from reading in or invalidating two or more distinct shared objects in one TU.) Thus, a programming system that implements intelligent placement of shared-data objects relative to cache blocks should be able to eliminate extra coherency traffic introduced by coresidency and possibly intentionally cause coresidency to reduce both misses and coherency traffic for data objects that are used together. Thus, a large block size should be better for data contention with programming system support providing that the size of shared data objects is close to cache block size.

Extra coherency traffic is also introduced by lock contention. The traces reflect the applications using the interlocked memory references to the cache to implement software locks, rather than using a kernel lock or other special mechanism. Because the locks are often accessed alternately by different processors, a read-modify-write reference frequently results in three bus transactions: read-private by the referencing processor, write-back by the cache block owner, and a reissued read-private by the referencing processor. If the lock is held by a process on another processor, it requires a similar 3 transactions to clear that lock or modify any data in the cache block containing the lock. To roughly quantify the effect of the locking, we ran a simulation of POPS (the trace with the highest level of contention) with interlocked read-modify-write references eliminated. This change reduced the total number of bus transactions by 40% even though these references are only 0.66% of all references, Eliminating lock access from the cache also reduces the frequency of **access** to small **shared data** objects. This reduces the intrinsic TU cost of placing shared data objects in different cache blocks. Our investigation of a hardware extension to VMP to support efficient locking is described in the next section.

Finally, examining the total number of bus transactions (**intrinsicTUs** + **coherenceTUs**) generated by the three **programs** as a function of the cache block size, we see that the optimal cache block size varies for minimizing the number of bus transactions. While the low point for POPS is at 32 bytes, the best size for THOR is 64 bytes, and 128 bytes for PEROUTE.[12] Although these numbers might seem to argue against the large cache block size of VMP, we note that for both POPS and THOR exhibit a miss rate that is quite high (roughly 1.5% of all references for POPS, relatively independent of the cache block size). This high level of traffic significantly taxes VMP, and supports the other thrust of our work, namely software techniques to reduce contention.

Overall, **we are encouraged** that PEROUTE appears feasible to run on a **4-processor** VMP without modification. Moreover, modifications to the other applications, including the locking and intelligent **shared** data object placement, appear to offer considerable performance improvements.

5 Hardware Refinements

Several refinements and extensions to the VMP hardware are being investigated, as described in the following section.

[12]Note that a flat profile for totalTUs in the graphs indicates that we should favor smaller cache block sizes, since there is at least some extra cost (although far from proportional) to going to larger cache block sizes.

5.1 Locking Support

Locks are single-bit data objects that exhibit no spatial locality of reference and require interlocked memory access. The large cache block size and software-implemented consistency makes memory-based locks inefficient to implement in VMP. Moreover, kernel-implemented locks are expensive in comparison to conventional memory-based locks because of the kernel trap overhead to effectively execute a single instruction, the test-and-set. As mentioned in the previous section, a VMP machine extended with **a** separate bus and special memory for locking would execute POPS with only 60% of the bus transactions indicated in Figure 3. Thus, efficient handling of locking does promise significant improvements in performance if the figure for **POPS** is indicative of other tightly coupled parallel applications. Overall, **separation** of memory access from lock memory allows the two different forms of memories to be implemented in ways optimal to each, similar to the separation of code memory from data memory in some machines.

We are proposing to extend VMP to provide ***a*** *lock* segment as part of every virtual address space for efficient memory-based lock access. The lock segment is supported by a lock cache and ***a*** lock, bus connecting the processors. The lock segment is a range of bits (addressed by byte addresses) which can be tested, cleared **and** atomically test-and-set. Each bit is associated with an ***address*** space identifier ***(asid)*** for protection. An attempt to access a lock with a different **asid** than currently being used by the processor results in a protection error.

Applications request lock allocation from the operating system, which interacts with the cache management software to perform this allocation. Efficient barriers can be realized by using a lock per process.

The **lock** segment is implemented by the cache hardware routing the fixed range of virtual ‘addresses in the lock segment to the lock cache, **a** special cache **memory on each** VMP processor board. An attempt to read a lock not in the lock cache returns **1 or** set. The lock cache then requests **a** copy of the current value of the lock from the lock bus. Thus, there is no **processor-**observed miss on access to the lock cache. A variety of options are being considered for the lock bus protocol.

Conceptually, we regard locking as **a** contention ***control*** mechanism **for memory. For example,** a critical region is locked to avoid contention in **access** to the **data structures associated** with that critical region (as well as to provide consistent **access).** The **proposed** locking support adds a small amount of hardware **and** provides a low-cost contention control **mechanism,** allowing VMP to be effectively used with finer grain parallelism and many more existing applications than would otherwise be feasible.

5.2 Memory-based Consistency Support

VMP currently **uses** a bus monitor and action table on each processor board for consistency, allowing it to **use standard** memory boards. Two disadvantages **of** this scheme are that: (1) it requires an action table on every processor board proportional in size to the size of physical memory and (2) it relies on bus broadcast, limiting the **scalability** of the system. We are investigating building a VMP memory board that includes a consistency directory that would replace the action table and bus monitor on the processor board.

The consistency directory consists of an entry per cache block frame of the memory module. Each entry consists of **a N+1** sized **bitmask,** where N is the maximum number of processors to support. One bit indicates whether the page is shared or exclusively owned. The other N bits indicate the processors that have copies of the cache block, if any. The memory module checks the consistency directory on each block transfer and generates interprocessor interrupts to holders of the copies as needed and updates the consistency directory. For example, a read-private bus transaction for a cache block that is exclusively owned by another processor is aborted and the owner processor is interrupted, causing it to write back the cache block and release ownership. The interprocessor consistency interrupts are handled in software the same way as in the current design.

The bitmask ensures that only those processors with copies are interrupted. The use of large cache line size makes it feasible to store this processor bitmask per directory entry while keeping the space overhead low. For example, with a limit of 15 processors and 128 byte cache block size, the space overhead is less than 2 percent. Overall, this modification to the design further simplifies the processor board without any performance penalty, but at the cost of building custom memory boards. It shares this disadvantage with other memory-directory based consistency schemes that have been described and studied in various forms by a number of researchers [7, 3, 5]. However, it appears necessary to build a memory board to provide the high-performance sequential memory access that VMP requires in any case.

5.3 Hardware Handling of Simple Cache Misses

The efficiency of VMP depends on the processor being able to handle a trap on cache miss with low latency. However, highly pipelined, multiple ALU high-performance processors may have a high latency to save state on a trap. For these processors, the basic VMP design can be extended with a modest addition in hardware complexity so that a common case, the *simple cache miss*, is handled in hardware without causing the processor to trap. A *simple cache miss* is one for which the cache directory knows the physical address of the cache block data and the selected cache slot is not modified (so no write-back is required). Hardware handling of simple cache misses appears to have a performance benefit without significant increase in hardware complexity, particularly if the processor incurs a significant cost in saving and restoring its state as part of trap handling. Hardware handling of simple cache misses requires some additional functionality in the cache controller and the addition of a simple cache directory interface (CDI) state machine. The CDI is a simple state machine that accesses and updates the cache directory data structures, otherwise maintained by software in the local memory of the processor board, as described in Section 2.

On a cachemiss, the cache controller stalls the processor (as is feasible with the MIPS-X processor), selects a cache slot to use, notifies the CDI to invalidate the SetBankMap entry for this slot, requests a virtual to physical address translation from the CDI, starts the block transfer and responds to the CDI with the selected cache set and bank number, allowing it to update the cache directory data structures. As soon as the block transfer completes, the cache controller allows the processor to continue. If the CDI is unable to supply the physical address, or the slot is modified, the cache controller signals a bus error to the processor and the processor traps to the cache miss handling routine to deal with the cache miss in software. If the bus transaction is aborted, the cache controller simply repeats the above sequence, possibly allowing some time for the write-back in the aborting processor to take place.[13]

In the case of a write access when the required cache slot is present but not exclusively owned, the operation may be handled without the CDI. A cache flag indicates whether the slot can be made writable (i.e. the user has write permission). If so, the cache controller performs an

[13]This scheme could also handle the case of the replacement cache slot containing modified data if the cache controller could determine the physical address to which the writeback should take place.

assert ownership bus transaction, modifies the cache flags and allows the processor to continue. If the bus transaction is aborted or the slot is not marked as writable, the processor traps to the cache management software.

Using this design, the bus monitor action table is updated to indicate the new cache slot data as part of the block move into the cache. However, to keep the hardware simple, the action table entry for the old slot's contents is not cleared. This entry is cleared by the bus monitor interrupt routines on demand if there is a consistency conflict with another processor accessing this data. That is, the bus monitor interrupt routines recognize that the slot is invalid and simply clear the entry. This contention occurs infrequently so there no significant performance penalty for this simplification to the hardware. Similarly, SCBRs are not freed when their last slot is invalidated. Instead, these records are garbage collected on allocation when the free list is empty.

The extension adds minimal complexity to the cache controller because all the actions required for simple cache misses are functions of the cache controller in the original design except for reading and writing the CDI. Moreover, the updating of the action table is performed as part of the bus transactions (which the cache controller already implements) so no modifications a-e required to the bus monitor or action tables. The modifications to the cache controller are estimated to entail additional states in the PALs plus a data connection to the CDI. Thus, the major additional complexity is the CDI, which could be realized as a single chip with data connections to the local memory and the cache controller.

The fact that non-simple cache misses and consistency interrupts are still handled by software has considerable benefits, both in simplifying the hardware and providing flexibility in cache management and interfacing to the operating system. As a rough illustration of this benefit, the cache management software is 1652 lines of C, compiling to 11,140 bytes of code and initialized data (for the VMP prototype using the MC 68020 processor.) Only 862 bytes of this code deals with the simple cache miss case. Thus, the major complexity of cache management is not concerned with the performance-critical case, and yet represents the bulk of the software.

Using this extended hardware support, the time to handle a simple cache miss is largely dependent on the time for the block transfer, which is a function of bus speed and cache block size. If we assume a 64-bit wide 20 megahertz bus and a cache controller and CDI capable of handling *a* simple cache miss (except for block transfer) in 500 ns, it appears that a simple cache miss (in the no abort case) would take roughly 2 microseconds versus 10 to 15 microseconds with a 10 MIPS processor handling a simple cache miss. Using results from memory traces of the previous section, we estimate that the simple cache miss case occurs for about 90 percent of the cache misses (83-95 percent across the three applications) assuming a 128 byte cache block size and no contention (or sequential execution). If we further, assume a cost of 50 microseconds on-average for a non-simple cache miss and a 0.4 percent miss rate and a 60 nanosecond cache access time without miss, the average memory reference cost is 87 nanoseconds with the hardware support versus 120 nanoseconds without, a 37 percent improvement." If the non-simple cache miss case is more expensive, such as would occur if caches generated virtual memory page faults, the benefit would be reduced. For example, if a page fault costing 8 milliseconds occurs every 100,000 instructions, it adds an **average** of 80 nanoseconds to every memory reference, reducing the speedup to less than 20 percent.

With parallel applications, assuming our memory reference traces are indicative, roughly 60 percent of the simple cache misses must wait for a cache block to be written back. Assuming the wait time is roughly 10 microseconds and the miss operation is executed twice in this

"The costs are calculated as weighed averages of the costs for each case.

case, the average miss cost is 14 microseconds with hardware support and 26 microseconds without it (assuming the same parameters as above) so the average memory reference cost is 116 nanoseconds with the hardware support versus 164 nanoseconds, a 41 percent improvement. However, our traces indicate cache miss rates ranging from 0.14 percent to 1.5 percent for these applications, giving an improvement range of 19 percent to 66 percent in the memory reference cost. The higher miss rates come from contention.

Overall, the extended hardware support has significant performance benefit. However, its benefits are reduced if there is any degree of demand paging in the system (raising the cost of the non-simple cache miss). Its benefit is also reduced by very low miss rates. We plan to further explore the merits of this extension and a similar extension to cache block invalidation mechanism as we gather more extensive memory reference measurements from VMP.

6 Related Work

VMP is a workstation multiprocessor architecture, in the general class of the Firefly [28], but focused on the next generation of high-performance microprocessors, as represented by the MIPS-X chip [2]. VMP represents a significant departure from the Firefly in that cache management is handled largely in software, rather than hardware. The VMP measurements also reflect a cache with much larger line size than the Firefly or those previously studied [16, 26, 25], providing further insight into cache design.

Lee et al. [19] studied cache design for highly pipelined scientific processors using interconnection networks to memory, and their results are an interesting contrast to ours. In particular, they conclude that maximal speedup is achieved with a very small cache block size. The difference appears to be due to the highly pipelined multiprocessors they consider as well as their focus on scientific applications in which references to large data arrays dominate. Their work indicates the need for further investigation of the performance of the VMP architecture for these applications. However, we also see the potential of structuring such applications to have reference patterns to these arrays which is suitable for VMP.

A number of advantages cited for VMP relate to the virtual memory and transaction management system we are developing for the V distributed system. VMP appears equally well suited to support the Mach virtual memory system [14] as well as the 801 transaction software [8], both of which reflect current directions in operating system design. The in-cache address translation mechanism is similar to that proposed for the Dragon [22], although the Dragon design is realized entirely in hardware. The work of Wood et al. [13] focuses on the performance benefit of the virtual memory page tables being resident in the cache; their design does not exploit information in the cache directory.

7 Concluding Remarks

The VMP design was developed and described almost 2 years ago. In that time, it has been implemented and the resulting machine is running the V distributed operating system as part of our workstation cluster. The design has stood up well to the test of implementation and performance evaluation. The processor board is simple, fast and inexpensive, resulting in a short development and debugging time. In particular, the software control handles the complexity of interconnecting the virtually addressed cache (optimized for processor performance) with the physically addressed system bus (optimized for efficient system memory access) with a minimal amount of processor board hardware.

The flexibility of software cache management admits a number of sophisticated features as part of the cache management, including in-cache address translation, efficient copy support, implicit locking for database transactions, process-level cache flush control and debugger data breakpoints. These extensions give the software-controlled cache approach significant functionality and performance benefits over conventional hardware-intensive approaches.

Measured performance of VMP for a single processor configuration is very good. Cache blocks are replaced sufficiently infrequently that the misses are amortized over many (cache-hit) memory references unless the program is rapidly referencing a very large number of pages. If a program references a very large number of pages, we expect the memory reference cost to be dominated by the cost of full page faults to the network or disk, in which case again the software overhead in handling cache misses is insignificant.

Our trace-driven simulations of a multiprocessor configuration indicate excellent performance for one of the three applications (PEROUTE), which exploits parallelism at a coarse granularity and has low contention overheads. For this application the appropriate cache block size is 128-256 bytes. For the other two applications (POPS and THOR), the contention is significantly higher, so much so that any architecture would incur significant contention overhead. For these two applications a cache block size of 32-64 bytes seems to be more appropriate.However, we are confident that a larger block size will be feasible when these applications are suitably restructured.

In a workstation environment, most of the parallel execution is expected to rise from executing multiple (sequential) applications in parallel. VMP's excellent performance in these cases, coupled with reasonable performance for parallel applications, and the potential for improving the structure of parallel applications to reduce contention makes VMP appear attractive for this environment.

We have described several hardware refinements and extensions we are exploring. Special memory-accessed locking support is being added, providing an efficient contention control and consistency enforcement mechanism with minimal additional hardware. The processor-based consistency mechanism is being migrated to the the memory modules, simplifying the processor boards, making them independent of the size of physical memory and reducing the dependence on system-wide broadcast. Finally, simple cache misses are to be handled in hardware, reducing the cache miss cost, especially for highly pipelined processors for which saving and restoring the state on a trap is *a* significant cost. This hardware support also makes it feasible to use *a* somewhat smaller cache block size and respond to cache misses faster, thereby reducing the cost of contention for tightly coupled parallel applications. These extensions appear to make the VMP architecture applicable to a wider range of high performance tightly-coupled parallel processing with *a* relatively modest increase in hardware cost and complexity.

The challenge of the VMP design is in the software. It remains to develop programming systems for VMP that intelligently place and align data objects relative to cache blocks to minimize contention and cache miss rates. These demands on software technology are significant but also a common theme in previous efforts to redefine some of the hardware/software boundaries. We are working on a parallel programming paradigm called *workform processing* [11] which addresses these issues – we are working on the design of a new language and a compiler for it. As a test of the workform model, we plan to restructure the POPS and THOR programs, and see what effect such restructuring has on the bus traffic generated by them.

In the hardware area, besides the refinements described in this paper, we are exploring an extension to a multiple-bus hierarchy using an inter-bus cache module that hierarchically extends the processor bitmask-based consistency scheme to over one hundred processors. We are also considering a new version of the processor board using the MIPS-X processor. This processor appears to be well-matched to the VMP design given that it is fast, can stall for a significant time for cache miss handling, and requires minimal state saving and restoring on fault into the cache management software.

8 Acknowledgements

This work was sponsored in part by DARPA under contract **N00039-84-C-0211,** the National Science Foundation Grant DCR-83-52048 and by Digital Equipment, and ATT Information Systems. Anoop Gupta is also supported by an award from Digital Equipment Corporation. We are grateful to the DEC Western Research Laboratory for the use of their CAD software which was used to design and layout the VMP processor board. Jeremy Dion has provided invaluable assistance in this **area.** We also wish to thank for the referees for their comments, which lead to some significant improvements in the paper.

References

[1] A. Agarwal and A. Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proc. of SIGMETRICS,* ACM, May 1988.

[2] A. Agarwal and M. Horowitz. *MIPS-X Internal and External Caches.* Technical Report, Computer Systems Laboratory, Stanford University, 1985.

[3] A. Agarwal, R. Simoni, J. Hennessy, and K. Horowitz. Scalable directory schemes for cache coherence. In *Proc. 15th Int. Symp. of Computer Architecture,* ACM, May 1988.

[4] A. Agarwal, R.L. Sites, and M. Horowitz. ATUM: A New Technique for Capturing Address Traces Using Microcode. In *Proc. 13th Int. Symp. of Computer Architecture,* June 1986.

[5] J. Archibald and J.L. Baer. An economical solution to the cache coherence problem. In *Proc. 12th Int. Symp.* on *Computer Architecture,* pages 355-362, ACM SIGARCH, June 1985. also SIGARCH Newsletter, Volume 13, Issue 3, 1985.

[6] C.G. Bell. Multis: A new class of multiprocessor computers. *Science,* 228:462–467, April 1985.

[7] M. Censier and P. Feautier. A new solution to coherence problems in multicache systems. *IEEE TC,* C-27(12):1112–1118, December 1978.

[8] A. Chang and M. Mergen. 801 Storage: Architecture and Programming. In *11th Symp. on Operating Systems Principles,* ACM, November 1987.

[9] D.R. Cheriton. *Unified Management of Memory and File Caching using the V Virtual Memory System.* Technical Report STAN-CS-88-1192, Computer Science Dept. Stanford University, 1988.

[10] D.R. Cheriton. The V distributed operating system. *Communications of the ACM,* 31(4), April 1988.

[11] D.R. Cheriton. Workform Processing: a model and language for parallel computation. Stanford University, Computer Science Technical Report, to appear 1986.

[12] D.R. Cheriton, G. Slavenburg, and P. Boyle. Software-controlled caches in the VMP multiprocessor. In *13th Int. Conf. on Computer Architectures,* ACM SIGARCH, IEEE Computer Society, June 1986.

[13] **D.A. Wood et al.** An in-cache address translation mechanism. In *Proc. 13th Annual Int. Symp. on Computer Architecture,* pages 358-365, ACM, June 1986.

[14] **M.** Young et al. The duality of **memory and communication** in the implementation of a multiprocessor operating system. *In 11th Symp. on Operating Systems Principles,* ACM, November 1987.

[15] R. Fitzgerald and R.F. Rashid. The integration of virtual memory management and interprocess communication in Accent. *ACM Trans. on Computer Sys.,* 4(2):147–177, May 1986.

[16] J.R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proc. Tenth International Symposium on Computer Architecture,* pages 124-131, June 1983.

[17] A. Gupta, C. Forgy, and R. Wedig. Parallel Algorithms and Architectures for Rule-Based Sytems. *In Proc. 13th Int. Symp. of Computer Architecture,* June 1986.

[18] W.D. Hillis. *The Connection Machine.* MIT Press, 1985.

[19] R.L. Lee, P.C. Yew, and D.H. Lawrie. Multiprocessor cache design considerations. In *14th Int. Conf. on Computer Architectures,* pages 253-262, ACM SIGARCH, IEEE Computer Society, June 1987.

[20] K. Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale, 1986. published as Yale technical report YaIe/DCS/RR-492.

[21] D. MacGregor and J. Robinstein. A Performance Analysis of MC68020-based Systems. *IEEE* Micro, 5(6):50–70, December 1985.

[22] L. Monier and P. Sindhu. The architecture of the dragon. *In Proc. Thirtieth IEEE Int. Conference,* pages 118-121, IEEE, Februrary 1985.

[23] Jonathan Rose. Locusroute: A parallel global router for standard cells. Submitted for publication, Stanford University, Computer Systems Laboratory, 1987.

[24] C.L. Seitz. The Cosmic Cube. *CACM,* 28(1):22–33, January 1985.

[25] A.J. Smith. Cache Evaluation and the Impact of Workload Choice. *In Proc. 12th Int. Symp. on Computer Architecture,* pages 64-73, ACM SIGARCH, June 1985. also SIGARCH Newsletter, Volume 13, Issue 3, 1985.

[26] A.J. Smith. Cache Memories. *Computing Surveys,* 14(3), September 1982.

[27] M. Stonebraker. Virtual memory transaction management. *Operating Systems Review,* 18(2):8–16, 1984.

[28] C. Thacker and L. Stewart. Firefly: A multiprocessor workstation. *In 2nd Int. Conference on Architectural Support for Progmmming Lunguuges and Opemting Systems,* pages 164-172, ACM, October 1987.