vNUMA: A Virtual Shared-Memory Multiprocessor

Matthew Chapman[*†]
Gernot Heiser[*†‡]
[*] *The University of New South Wales*
[†] *NICTA*
[‡] *Open Kernel Labs*
matthewc@cse.unsw.edu.au
http://ertos.nicta.com.au

First published in Proceedings of the 2009 USENIX Annual Technical Conference

Abstract

vNUMA, for *virtual NUMA*, is a virtual machine that presents a cluster as a virtual shared-memory multiprocessor. It is designed to make the computational power of clusters available to legacy applications and operating systems.

A characteristic aspect of vNUMA is that it incorporates distributed shared memory (DSM) inside the hypervisor, in contrast to the more traditional approach of providing it in middleware. We present the design of vNUMA, as well as an implementation on Itanium-based workstations. We discuss in detail the enhancements to standard protocols that were required or enabled when implementing DSM inside a hypervisor, and discuss some of the tradeoffs we encountered. We examine the scalability of vNUMA on a small cluster, and analyse some of the design choices.

1 Introduction

Shared-memory multiprocessor (SMM) systems provide a simple programming model compatible with a large base of existing applications and operating systems. They naturally lend themselves to providing a single system image (SSI) running a single operating-system (OS) instance with a single resource name space.

However, for many compute-intensive applications, a network of commodity workstations presents a more cost-effective platform. These systems deliver the same (theoretical) compute power with much less expensive hardware, and are easily extensible and re-configurable. Yet their computing power is much more difficult to harness. Most existing OSes were not designed for cluster environments, and applications designed for shared-memory systems need to be redesigned for clusters by using explicit communication over the network.

Previous attempts have been made to bridge the gap between the ease of programming and legacy support of SMM systems and the economies of cluster hardware. These include distributed shared memory (DSM) libraries such as Ivy [23] or Treadmarks [19], which provide a limited illusion of shared memory to applications, provided that the programmer uses the primitives supplied by the library. Other projects have attempted to retrofit support for cluster-wide process scheduling and migration into OSes [2, 27, 35]. However, these approaches require extensive and intrusive OS changes, which are difficult to keep up to date with the fast pace of OS development.

This paper explores a different approach: the use of virtualization to bridge the gap between SMM systems and workstation clusters. We present *vNUMA* ("virtual NUMA"), a virtual shared-memory multiprocessor built from a cluster of commodity workstations. A hypervisor runs on each node of the cluster and manages the physical resources. A single virtualized instance of an OS, such as Linux, is then started on the cluster. This OS and its applications executes on a virtual ccNUMA machine with many virtual CPUs. The virtualization layer transparently maps the virtual CPUs to real CPUs in the cluster, and provides DSM using software techniques. In this way, a single OS instance can be scaled "outside the box", utilizing the computing resources of more than one node. Users gain all of the advantages of such an SSI multiprocessor, such as a single view of resources and transparent process scheduling.

The core ideas of vNUMA have been presented in an earlier short paper [7]. Here we focus on the design and implementation issues that are critical to making vNUMA work. We address the problem of constructing a high-performance virtual NUMA system on commodity hardware by:

- an approach to write sharing which individually intercepts sparse write accesses, while falling back to a page-based write-invalidate protocol when appropriate,

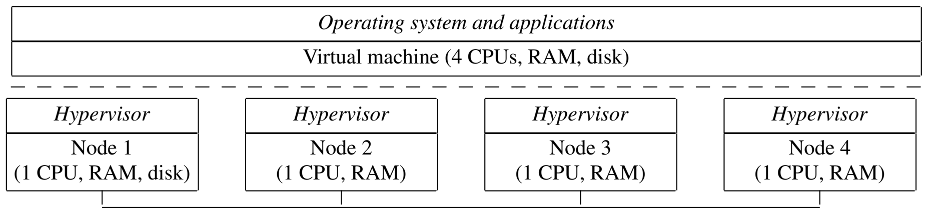


Figure 1: Example vNUMA system

- introducing the technique of write-broadcast with deterministic incremental merge for providing total store order, and
- demonstrating an efficient approach for avoidance of page thrashing.

In the next section we present an overview of the vNUMA hypervisor and its DSM system, which is designed for a small cluster of commodity workstations. In Section 3 we discuss a number of enhancements to established DSM protocols that improve their suitability for use inside a hypervisor. Section 4 takes a detailed look at implementation issues, including architecture-specific optimisations. Section 5 presents an evaluation of our vNUMA prototype. Related work is summarised in Section 6.

2 vNUMA Overview

2.1 Approach

In order to minimise overheads, vNUMA is designed as a Type-I hypervisor, executing on bare hardware with no host OS. Our prototype was built on Itanium workstations, which are frequently deployed in clusters for high-performance computing (HPC) use. While the vNUMA design is independent of a specific ISA, the implementation does use processor-specific optimisations.

The majority of previous software DSM systems have been designed as middleware running on top of an OS. In vNUMA, the DSM system is integrated with the hypervisor. There are two levels of memory address translation in a virtualized system. The guest OS maps applications' virtual addresses onto a *guest-physical* address space, which represents the physical memory of the virtual machine. Then, the hypervisor maps guest physical addresses to real physical addresses on a host computer. This lower layer, transparent to the guest OS, is where the vNUMA DSM system operates. It provides operating systems with the illusion of a single physical address space across multiple host computers, as indicated in Figure 1.

As a result, the shared address space in vNUMA comprises not just some subset of data memory that is known to be shared, but all of the memory of the virtual machine. Since our aim is to run unmodified application binaries (and, ideally, unmodified OSes), vNUMA must faithfully reproduce the hardware SMP programming model. Doing this efficiently presents challenges. On the other hand, vNUMA runs in the processor's privileged mode, which gives it access to certain techniques that may be difficult or prohibitively inefficient for a userspace DSM system. Examples include the efficient emulation of individual instructions, and the use of the performance-monitoring unit (PMU) to track the execution of specific instructions.

2.2 Basic DSM protocol

At the heart of the vNUMA DSM system is a simple single-writer/multiple-reader write-invalidate protocol based on the Ivy protocol [23]. For page location, vNUMA implements a fixed distributed manager scheme, whereby the global guest-physical address space is divided into equal-sized portions; each node acts as a *manager* for one of these portions.

vNUMA's transparency requirements imply that the concept of a manager node is unknown outside the hypervisor. However, efficiency is improved if the guest OS has a notion of locality. vNUMA uses the concept of NUMA node-local memory to ensure that the guest will favour locally-managed memory when making allocation decisions, and as such works best with a NUMA-aware guest OS. While for normal DSM systems the concept of the manager node is a complication required for efficiency, for the virtual NUMA system it is actually a good match.

vNUMA's DSM algorithm is based on the a version of the Ivy protocol which the Ivy authors describe as the "improved" protocol. The improvement keeps the *copyset* information (where copies of a page are held) with a changing page *owner* rather than the manager. This helps to minimise the number of messages required, and to avoid deadlock issues that are a problem with the basic protocol [13].

3 Enhancements to DSM Protocols

Latency of DSM operations is the crucial limiting factor for the performance of vNUMA. Whenever a fetch or invalidation message is sent, consistency requires that execution on the local processor must stall until the response is received. Here we discuss protocol improvements that are designed to minimise the number of stalls and messages required for DSM operation.

3.1 Double faults and ownership

In the original Ivy protocol, a page that has been fetched on a read fault would have to be re-fetched on a subsequent write fault in order to ensure consistency. A later optimisation avoided the double transfer with the help of version numbers [20]. We use an optimisation that seems to have been used in Mirage [11]: an owner can determine whether the page data needs to be sent simply by consulting the page's copyset information. This is because any intervening writes would have invalidated the faulting node's read copy and hence removed it from the copyset.

Another optimisation also goes back to Mirage but is simplified in vNUMA: as soon as the manager becomes a member of the copyset, ownership is automatically transferred to the manager (Mirage required extra messages for this).

3.2 Addressing sparse data accesses

Minimising the number of communication events in a distributed shared memory system depends critically on caching of remote data. Many commonly used data structures, such as linked lists and trees, tend to have poor spatial locality, and may result in a processor accessing many pages. If locally cached copies of these pages can be accessed, then overheads are small, but if each of the pages regularly requires a remote fetch, performance will suffer greatly.

In the absence of writes, pages eventually become read-shared, allowing each processor to access the cached copy of those pages without any communication. This is clearly desirable. Now consider that some processor occasionally writes a value to a certain page that is otherwise read-shared. In the Ivy protocol, first the writer must stall while all of the read copies are invalidated, then all of the active readers eventually stall and re-fetch the entire page data. Clearly it would be more efficient, for such sparse updates, to propagate the individual write to any readers.

3.2.1 Write detection

In any such protocol, writes must be detected and write update messages sent to other nodes. Write detection at sub-page granularity is a challenge to implement efficiently. *Page diffing*, as implemented in Munin [3] and many later systems, cannot be used by vNUMA, for several reasons.

Firstly, by the time that the diffing is performed, information has been lost about the size of the writes, which has implications for the outcome of conflicting writes. For example, assume that a 4-byte integer variable has an initial value of 0. Consider a case where processor P1 writes 1 to the variable, P2 writes -1, and then P3 issues a read. The Itanium architecture dictates that the outcome will be one of 0, -1 or 1 (depending on which of the writes have been seen at P3). However, the diff generated at P1 may contain as little as one byte, since in binary representation only one byte of the value has changed. The diff generated at P2 contains four bytes, since all four bytes of the binary representation have changed (-1 = 0xffffffff in hexadecimal). After both diffs are applied, the value at P1 may be 0xffffff01, which is not one of the valid outcomes. Diffing at a 32-bit granularity would solve this problem for 32-bit values, but there would still be problems with smaller and larger types. Systems that employ diffing, such as TreadMarks [19], rely on the programmer to avoid issuing conflicting writes within an interval, and to take care when using smaller types than the diff granularity. However, at the ISA level there is no such requirement; in fact the example above is completely legal if the programmer does not require a guarantee as to which change is applied first. This would present problems for legacy code on vNUMA.

Secondly, the standard diffing approach involves making the page freely writable on the first write access, in order to avoid further write faults. However, if a page is both readable and writeable, then atomic read-modify-write instructions such as compare-and-exchange will freely execute, thus destroying their semantics. User-level DSM systems that employ diffing schemes can avoid this issue by stating that the programmer must use the synchronisation constructs provided by the DSM system, and not rely on the behaviour of atomic instructions to shared memory. This is not practical for vNUMA.

An alternate approach, *software write detection*, as used in Midway [37], relies on compiler support. This would prevent transparent distribution of legacy applications, and is therefore also not suitable for vNUMA.

We therefore attempt to intercept writes individually, a technique we describe as *write trapping*. While this is prohibitively expensive for user-level DSM systems, the overhead can be kept much smaller in a thin hypervisor such as vNUMA. The current C language implementation results in an overhead of around 250 cycles per write, but this is largely due to compiler limitations; in theory under 100 cycles should be achievable.

Even so, writes are frequent operations and trapping *every* write in the system would be impractical; indeed the majority of pages in the system are not actively write-shared at all. vNUMA uses an adaptive scheme which changes a page's mode between this write-trapping (write-update) mode and the basic write-

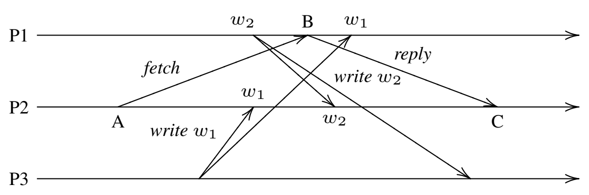


Figure 2: Timeline showing a possible ordering problem

invalidate mode, depending on the access pattern.

The adaptation scheme currently implemented is similar to the *read-write-broadcast* (RWB) protocol [31] developed for hardware cache coherence. The run-length of local writes to a page that are uninterrupted by writes received from other nodes is tracked with a counter. If the count exceeds a threshold, trapping of individual writes ceases and the page is transitioned to write-invalidate mode, in which we use the conventional Ivy-like write-invalidate protocol described earlier. This can reflect two types of access patterns — either one node is accessing the page exclusively, or one node is making a large number of updates to the page in a short time — and in both cases invalidation is likely to perform better. The decision is made individually by each node, so even if one node chooses to acquire the page exclusively, other infrequent writers continue to intercept writes to the page and report them back to the exclusive owner (providing there are no reads).

This scheme makes its decision purely on the basis of tracking write accesses. Its drawback is that it will not detect producer-consumer sharing with a single intermittent writer and multiple readers. This leads to periodic invalidation of the readers' copies and subsequent re-faulting, even though the write-update mode may be better in this case. An improved algorithm might be one similar to the *efficient distributed write protocol* (EDWP) [1], which tracks both read and write accesses, and prevents a transition to exclusive mode if more than one processor is accessing the page. However, this is considerably more complex (since sampling read accesses is required) and has not been implemented.

3.2.2 Write propagation

For pages in write-update mode, vNUMA broadcasts writes to all nodes. While this may seem inefficient, it has some advantages; it greatly reduces the complexity of the system and naturally results in total store order (TSO) consistency. Per-packet overheads are amortized by batching many writes into a single message (see Section 4.3). Certainly this design choice would limit scalability, but vNUMA is designed for optimal performance on a small cluster.

Each node generally applies any write updates that apply to pages that it has read copies of, and discards any irrelevant updates. However, care must be taken when applying write updates to a page that is being migrated. A node P2 receiving a page from P1 queues the updates it receives while the page is in flight. Then, it must apply the subset of queued writes that have not already been applied at P1. In other words, P2 must apply exactly those updates which were received at P1 *after* P1 sent the page to P2. An example is shown in Figure 2: write w_1 must be applied, while w_2 must be discarded.

Our algorithm for determining which writes to apply assumes that the network provides *causal order delivery*, which is a property of typical Ethernet switches (c.f. Section 4.5). We provide a brief description here, more details are available elsewhere [6].

We maintain at each node a counter of writes, and that counter value is included in a page-fetch reply message. As per Figure 2, A denotes the event of P2 sending a fetch message to P1, B the event of P1 receiving that message and immediately replying to P2, and C the event of P2 receiving the page. In the figure, the respective counter values are $N_A = 0$, $N_B = 1$, and $N_C = 2$. N_1 denotes the number of writes from P1 queued at P2 at event C ($N_1 = 1$ in the figure). The algorithm then becomes:

- discard the N_1 messages pending from P1;
- out of the remaining writes, apply the latest $N_C - N_B$ (and thus discard the earliest $N_B - N_A - N_1$ writes).

In the example, the first step will drop w_2 and the second step will apply w_1.

3.2.3 Deterministic incremental merge

The write-update algorithm as presented so far is insufficient to guarantee coherence in a strict sense. In the example shown in Figure 3, where nodes P1 and P2 simultaneously write to a location X, P1 could observe $X = 1$ followed by $X = 2$ while P2 observes $X = 2$ followed by $X = 1$, in violation of coherence. Two solutions to this problem exist in the literature [8]: a central sequencer or associating every write with a globally-unique sequence number. The central sequencer, while guaranteeing that all nodes converge on the same value, does not prevent intermediate values from being observed at a single node, in violation of the architecture's specification of memory coherence. It also presents a bottleneck.

A globally-unique sequence number can be implemented as a local sequence number — synchronised on communication — with the node number as a tie-breaker where no causality relationship exists [8, 21]. However, the conventional deterministic merging approach [8] would involve waiting to receive write mes-

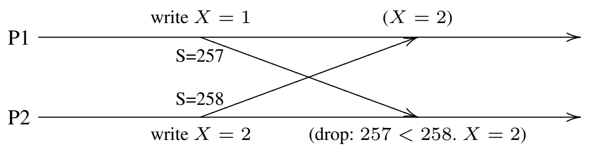


Figure 3: Coherence problem with write notices, and its resolution by deterministic merging according to sequence numbers.

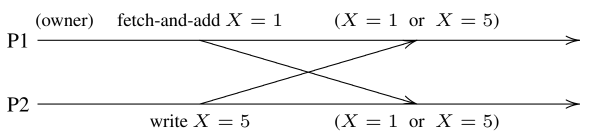


Figure 4: Simultaneous atomic operation and remote write. P1 is the owner of X and therefore has permission to execute atomic operations. According to the Itanium architecture, the correct result is either 5 or 6, depending on which operation appears first in the total order. Here, even with deterministic merging, $X = 1$ may occur.

sages from all nodes before deciding on a final value. As vNUMA only sends write messages as needed, a particular node may be quiet for a considerable time, which would necessitate regular empty write messages to ensure coherence.

Note, however, that coherence only requires total ordering on a per-location basis. Consider the case where $\{w_1, w_2, .., w_n\}$ are a set of writes to the same location, ordered by their global sequence number. From the point of view of program semantics, it is not essential to guarantee that all of $\{w_1..w_n\}$ are observed at any particular node, as long as the observed subset follows the correct ordering and culminates in the proper final value. In other words, observing $\{w_2, w_1, w_n\}$ is not allowed since w_1 must precede w_2, but observing $\{w_1, w_n\}$ or even just $\{w_n\}$ is allowable. Omitted intermediate values could correspond to the case where a processor was not fast enough to observe the intervening values.

We make use of this fact to implement a technique we call *incremental deterministic merging*. Each incoming write notice is applied immediately, but it is only applied to a certain location if its sequence number is greater than that of the last write to that location. Since every node receives all write notices, the value of that location always ultimately converges on the write with the maximum sequence number (w_n), with any intermediate values respecting the required ordering. Figure 3 shows how this resolves the original problem.

3.3 Atomic operations

The protocol described so far is sufficient for correctness, but highly inefficient for hosting an OS (such as Linux) that uses atomic instructions (`xchg`, `fetchadd` or `cmpxchg`) to implement kernel locks. Any of those operations results in a fall-back to write-invalidate mode, making kernel locks very expensive. We therefore introduce an extension to the protocol, which we call *write-update-plus* (WU+).

An important observation is that, in the Itanium architecture and other typical processor architectures, there is no requirement for ordering between an atomic read-and-write instruction and remote reads. A remote read can safely return either the value before or after the atomic operation. Thus, there is no need for invalidation of read-only copies when an atomic operation is encountered; the write phase of the operation can be propagated to readers via the write-update mechanism.

However, in order to guarantee atomicity of the read and write phases, only one processor at any time can be allowed to perform an atomic operation to a particular location. In the WU+ protocol, we enforce that only the owner of a page can execute atomic operations on that page. Any other node must first acquire ownership.

In addition, simultaneous atomic operations and remote writes can lead to incorrect results, as shown in Figure 4. The WU+ protocol therefore enforces a single writer for pages targeted by atomic operations. Thus, at any point, a page can be in one of three modes: *write-invalidate*, *write-update/multiple-writer*, or *write-update/single-writer*. The transition from multiple- to single-writer mode occurs when atomic operations to a page are intercepted; nodes are synchronously notified that they can no longer generate write updates to the page without acquiring ownership.

4 Implementation

The implementation of vNUMA is around 10,000 lines of code. Of this around 4,000 lines constitute a generic Itanium virtual machine monitor, the DSM system is around 3,000 lines, and the remainder deals with machine-specific initialisation and fault handling. In total the hypervisor code segment is about 450KiB (Itanium is notorious for low code density).

Besides generic protocol optimisations, we used a number of implementation techniques to optimise performance, which we discuss in this section. Some of these are processor-independent, others make use of particular Itanium features (but similar optimisations can be made for other ISAs).

4.1 Avoiding thrashing

A naïve DSM implementation suffers from a page thrashing problem, indicated in Figure 5. If two nodes simultaneously write to a page, the page may be transferred back and forth with no useful work done. A

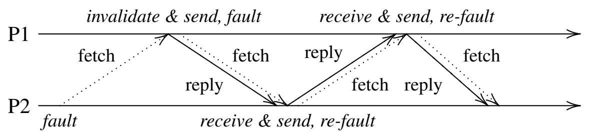


Figure 5: Timeline demonstrating the page thrashing problem. Solid lines indicate transfers of ownership.

frequently-used solution to this problem is to introduce an artificial delay to break the livelock. However, this is non-optimal by design, as there is no easy way to determine an appropriate delay, and the approach increases latency. Instead, we use an approach that guarantees that at least one instruction is executed before a page is transferred.

One way to implement this is by putting the machine into single-step mode after receipt of a page, and not processing any page requests until the trap that is caused by the execution of the next instruction is processed (at which time normal execution mode is resumed).

A cheaper alternative (implemented in vNUMA) is to consult the performance-monitor register that counts retired instructions to determine whether progress has been made since the last page transfer. (Note that checking the instruction pointer is not sufficient, as the code might be executing a tight loop, which could mask progress.) If lack of progress is detected, then one could fall back to the single-step approach. Instead we optimistically continue and re-check after a short delay. While this is similar to the timed-backoff scheme implemented in other DSM systems, vNUMA can use a very short delay to minimise latency, as the hypervisor can prevent preemption and thus ensure the opportunity for progress.

A complication of the chosen scheme is that one instruction may access several pages, up to four on the Itanium (an instruction page, a data page and two register-stack engine pages). This introduces the possibility of a circular wait condition, and thus deadlock.

We prevent deadlock by applying the anti-livelock algorithm only to pages accessed via explicit data references, and not instruction or register stack pages. Since the data reference is always logically the last reference made by an instruction — occuring after the instruction reference, and after any register stack accesses — instruction completion is guaranteed once the data page is obtained, and there is no possibility of deadlock. Indeed it is not necessary to apply the livelock prevention algorithm for instruction and register stack references, since instruction accesses are always reads, and the Itanium architecture specifies that register-stack pages should not be simultaneously accessed by multiple CPUs (or undefined processor behaviour could result). Even if a malicious application were to invoke this livelock case, it would not prevent the operating system from taking control and the process could be killed. Thus, this strategy prevents livelock in a well-behaved operating system while also preventing any possibility of deadlock.

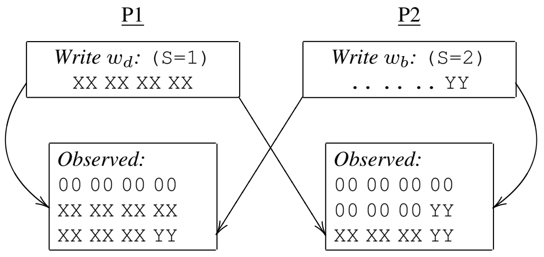


Figure 6: Combining writes of different sizes. On P2, write w_d appears to modify 3 bytes.

On some other architectures such as x86, this approach might still result in deadlock, since a single instruction may access several data pages. One possibility would be to release pages after a random period of time, even if no progress is made. In the worst case, this reintroduces the problems associated with backoff algorithms, but should perform better in the common case, while ensuring that a permanent deadlock does not occur.

4.2 Incremental merging

In Section 3.2.3 we somewhat vaguely referred to "locations" as the destinations of writes. Given that real architectures support writes of different sizes, we need to understand at which granularity conflict resolution must be applied. Figure 6 demonstrates that it must be applied at the byte, not the word level: the 4-byte write w_d at P1 with sequence number $S(w_d) = 1$ logically precedes the byte-sized write w_b at P2 with $S(w_b) = 2$. If the newer byte-sized write happens to be applied first at some node, then when the older 4-byte write is received, it must only appear to modify the top 3 bytes. This set of observed values is consistent with the Itanium memory consistency model [16].

This makes efficient implementation a challenge, as keeping separate sequence numbers for each byte of memory is clearly prohibitive. As the majority of updates do not conflict, tracking overhead must be minimised.

Fortunately, sequence-number information only needs to be kept for short periods. Once updates with a certain minimum sequence number are received from all nodes, all information related to lower sequence numbers can be discarded.

This observation enables an implementation of sequence numbers that is simple and has low overheads. We use a fixed-size buffer that stores information about

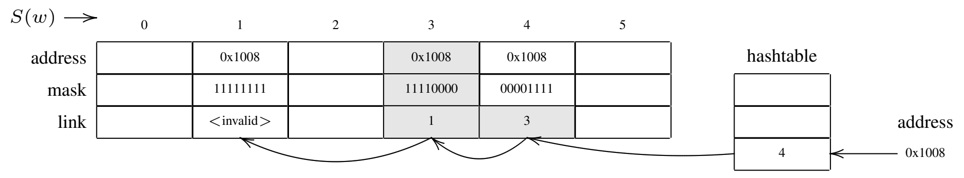


Figure 7: Data structure for coherence algorithm. The example shows an incoming write with sequence number 3, address 0x1008 and mask 11111111 (entire 8 bytes); the unshaded fields show the "before" state (but note that entry 4 is originally linked to entry 1). The hash chain is traversed as far back as sequence number 4; since that logically newer write wrote 00001111 (the lower four bytes), the mask is constrained to 11110000 (the top four bytes). The appropriate slot for the new write is then updated and linked in place.

a certain number of preceding writes (Figure 7). Each write is described by the address of the 64-bit machine word that it targets and a mask of bytes within that word (note that we assume that writes never cross a machine-word boundary). Writes are directly inserted into the buffer using the least significant bits of their sequence number as an index; assuming that sequence numbers are allocated in a unique and relatively dense fashion, this mapping is quite efficient. For fast lookup, writes are then indexed using a hash function of their target address; writes with the same hash value are linked together in a chain. This chain is always kept in reverse sequence number order.

The only operation on this data structure is adding a new write. While traversing the linked list to insert a write, all logically newer writes to the same address are encountered, which are used to constrain the mask of bytes to be written. Once a link field with an older sequence number is reached, traversal stops and the new write is inserted into the chain. The constrained mask is returned and used to determine the bytes in memory that are actually modified.

Since a chain is never traversed past the sequence number of a newly received write, the chains need never be garbage-collected. It is sufficient to make the buffer large enough so that it covers the window of sequence numbers that can be received from other nodes at any time. Since each node tracks the last sequence number received from each other node, a violation of this rule can be detected and a stall induced if necessary; however such stalls are clearly undesirable and can be eliminated by ensuring that each node does periodically send updates.

4.3 Write batching

Write update messages are small, and vNUMA batches many of them into a single Ethernet message in order to improve performance. Batching can make use of the processor's weak memory ordering model. The Itanium architecture uses *release consistency*: normal load and store instructions carry no ordering guarantees, but load instructions can optionally be given acquire semantics (guaranteeing that they become visible prior to subsequent accesses), while store instructions can optionally have release semantics (guaranteeing that they become visible after preceding accesses).

Acquire semantics require no special care, since the processor guarantees this behaviour on local operations, and because operations are never visible remotely before they are visible locally.

Release semantics require special care, however. Consider an access A that is followed by a write with release semantics W_{rel}. A must become visible on all nodes before W_{rel}. The processor interprets the release annotation and guarantees that A completes before W_{rel}. However, in the case that A is a write, local completion does not imply remote visibility — writes may be queued by vNUMA before being propagated to remote nodes. It is up to vNUMA to guarantee that A is observed before W_{rel}.

This is trivial if W_{rel} is to a write-update page: if A is to an exclusive page, it becomes visible immediately and thus necessarily before W_{rel}; if not, then the DSM system simply needs to ensure that the writes are sent in order. The interesting case is where W_{rel} is to an exclusive page and A is a queued write to a write-update page. In this case, the DSM system needs to ensure that W_{rel} is propagated before a read response to A.

The challenge is to detect when W_{rel} is to an exclusively-held page, as this cannot be made to trap without making all ordinary writes to the same page fault as well. Fortunately, the Itanium performance monitoring unit (PMU) provides a counter which can be configured to count releases. When a read request arrives for an exclusive page, the counter is checked to determine whether a release occurred on the last interval. If so, the write buffers are flushed before sending the read response.

As an additional optimisation, the write queue is eagerly flushed at the time that a write is intercepted, if a release has been seen (either on that instruction or in the previous interval) and if the network card transmit queue is empty. This expedites transmission of writes, since a release is usually used in the context of data that is intended to be observed by another processor. If the transmit queue is not empty, then the flush is scheduled to occur after a delay; this rate-limits the update packets and allows additional writes to accrue while previous update packets are being transmitted.

4.4 Memory fences

Itanium also provides a memory fence instruction, `mf`, that has both acquire and release semantics: loads and stores cannot pass it in either direction. The PMU counts `mf` as a release (as well as an acquire), so the above detection mechanism can be used to ensure that writes are ordered correctly across a fence. The one case that is problematic is the ordering between writes and subsequent reads. If a write is separated from a subsequent read by a fence, as in Figure 8, then the strict semantics of `mf` would require preventing the read from returning a cached copy before the write is visible everywhere. In practice this means that if both both X and Y are initially zero, at most one processor is allowed to read that value.

P1	P2
$Y = 1$	$X = 1$
mf	**mf**
read X	read Y

Figure 8: The memory fences prevent that both processors' reads return the initial values of the respective variables.

A strict implementation of the `mf` semantics would have severe performance implications in vNUMA. Instead, we decided to compromise our goal of full transparency, and require that `mf` operations are replaced by atomic operations (equivalent to a lock-based implementation of `mf`). Despite the assortment of synchronisation algorithms implemented in Linux, only one case was encountered in testing which required a full fence — the implementation of the `wait_on_bit_lock` function — and this was resolved via a simple modification.

4.5 Inter-node communication

vNUMA performance is highly sensitive to communication latency. This rules out hosting device drivers inside a guest OS as done in many modern virtual-machine monitors. Instead, vNUMA contains, inside the hypervisor, latency-optimised drivers for a number of Gigabit Ethernet chipsets.

We further minimise communication overhead by defining a very simple protocol at the Ethernet layer. We use the coalescing feature of Ethernet cards to separate the headers and payload into different buffers to enable zero-copy in the common case (in the special case where a local write occurs while a page is being sent, a shadow copy is created). Transfers of 4KiB pages either use a single 'jumbo' frame or are broken into four fragments. Fragmenting the packet is actually preferable to reduce latency, since the fragments can be pipelined through the network (this is also why four fragments are preferable to three, although above this the overheads outweigh the benefits).

vNUMA also makes extensive use of known properties of networking hardware, in order to avoid protocol overhead where possible. Specifically, vNUMA relies on the network to be (mostly) *reliable*, to provide *causally-ordered delivery*, and ideally to provide *sender-oblivious total-order broadcast*. The last requirement means that if P1 broadcasts m_1, and P2 broadcasts m_2, then either all other observers observe m_1 before m_2, or all other observers observe m_2 before m_1. "Sender-oblivious" means that P1 and P2 do not need to make any conclusions about the total order; this is an optimisation geared towards Ethernet, where a sender does not receive its own broadcast.

Causally-ordered delivery is guaranteed by the design of typical Ethernet switches. Reliability is not guaranteed, but packet loss is very rare. vNUMA is therefore optimised for lossless transmission. Timeouts and sequence numbers, combined with a knowledge that the number of messages in-flight is bounded, are used to deal with occasional packet loss.

Total-order broadcast usually holds in small switches but may be violated by a switch that contains several switch chips connected by a trunk, as a broadcast will be queued in a local port on one chip before forwarded over the trunk. It may also be violated when packets are lost. In this case, remote store atomicity may not hold in vNUMA. This could potentially be resolved with a more complex protocol for store atomicity, similar to our approach to coherence. We did not design such a protocol. In practice, this limitation is of little significance; many other processor architectures including x86 also do not guarantee store atomicity.

4.6 I/O

vNUMA contains support for three classes of virtual devices: network (Ethernet), disk (SCSI) and console.

The **network** is presented as a single virtual Ethernet card. As processes arbitrarily and transparently migrate between nodes, and TCP/IP connections are fixed to a certain IP address, transparency requires a single IP address for the cluster. Outgoing messages can be sent from any node, vNUMA simply substitutes the Ethernet address of the real local network card into outgoing packets. Incoming packets are all received by a single node. This has the advantage that the receiving part of the driver and network stack always runs on a single node, but the disadvantage that the actual consumer of the data may well be running on a different node.

The ideal approach for dealing with **disks** would be to connect them to a storage area network (SAN), so that they can be accessed from any of the nodes. This is done by Virtual Iron's VFe hypervisor [34], but is in conflict with vNUMA's objective of employing commodity hardware. Therefore, the vNUMA virtual machine provides a single virtual SCSI disk. The present implementation routes all disk I/O to the bootstrap node, which contains the physical disk(s). It would be possible to remove this bottleneck by striping or mirroring across available disks on other nodes.

The **console** is only supported for debugging, as users are expected to access the vNUMA system via the network. All console output is currently sent to the local console (which changes as processes migrate). Input can be accepted at any node.

4.7 Other implementation issues

vNUMA virtualizes inter-processor interrupts (IPIs) and global TLB-purge instructions in the obvious way, by routing them to the appropriate nodes.

In order to boot up a vNUMA system, all of the nodes in the cluster must be configured to boot the vNUMA hypervisor image in place of an operating system kernel. Then, one of the nodes is selected by the administrator to be the bootstrap node, by providing it with a guest kernel image and boot parameters; the other nodes need no special configuration.

Once the bootstrap node initialises, it uses a discovery protocol to find the other nodes and their resources, and provides them with information about the rest of the cluster. It then starts executing the guest kernel. As part of its normal boot process, the guest OS registers an SMP startup address and wakes the other nodes by sending IPIs. The other nodes start executing at the given address in the globally-shared guest-physical address space, thus faulting in the OS image on demand.

4.8 Limitations

Like the ubiquitous x86 architecture, Itanium was originally not trap-and-emulate virtualizable [24]. While this has now been mostly remedied by the VT-i extensions [17], a number of challenges [14] remain, particularly relating to the register stack engine and its interaction with the processor's complex translation modes. vNUMA utilizes some para-virtualization of the guest OS, and thus presently only supports Linux guests.

5 Evaluation

We evaluated vNUMA using three types of applications, which cover some of the most common use scenarios for large computer systems: computationally-intensive scientific workloads, software-build workloads, and database server workloads.

5.1 Test environment

Our test cluster consisted of eight HP rx2600 servers with 900MHz Itanium 2 processors, connected using a Gigabit Ethernet via an HP ProCurve 2708 switch. Since vNUMA does not yet support SMP within a node, only one CPU was used in each server.

The guest OS was Linux 2.6.16, using default configuration settings where possible, including a 16KiB page size. An exception are the Treadmarks measurements, which were performed with a 4KiB page size to provide a fair comparison of DSM performance (since vNUMA subdivides pages to 4KiB granularity internally).

Pre-virtualization [22] was used to automatically transform the Linux kernel for execution on vNUMA (our Itanium machines are not VT-i enabled). Three minor changes were made manually. Firstly, the Linux `wait_on_bit_lock` function was modified as described in Section 4.4. Secondly, the `clear_page` function was replaced with a hypervisor call to allow it to be implemented more optimally. Finally, the kernel linker script was modified to place the `.data.read_mostly` section on a separate page to ease read-sharing (the default setup co-allocates this section with one which contains locks).

Results presented are a median of the results from at least ten runs of a benchmark. The median was chosen as it naturally avoids counting outliers.

5.2 HPC benchmarks

HPC is a main application of compute clusters, and therefore a natural application domain for vNUMA. While many HPC applications use an explicit message-passing paradigm as supported by libraries such as MPI [26], a significant number rely on hardware-supported shared memory or DSM, and are therefore well-suited to execution on vNUMA. We used TreadMarks [19] as a DSM baseline. While TreadMarks may no longer represent the state of the art in DSM research, it is one of the few DSM systems that has been widely used in the scientific community.

TreadMarks is distributed with an assortment of benchmark applications, mostly from the Stanford SPLASH-2 suite [36] and the NAS Parallel Benchmarks from NASA [10]. To avoid biasing the evaluation against TreadMarks, we used the unmodified TreadMarks-optimised sources, and for vNUMA provided a stub library that maps TreadMarks APIs to `fork()` and shared memory. We also ran the benchmarks on one of our SMP servers on native Linux to show best-case scalability (although limited to the two

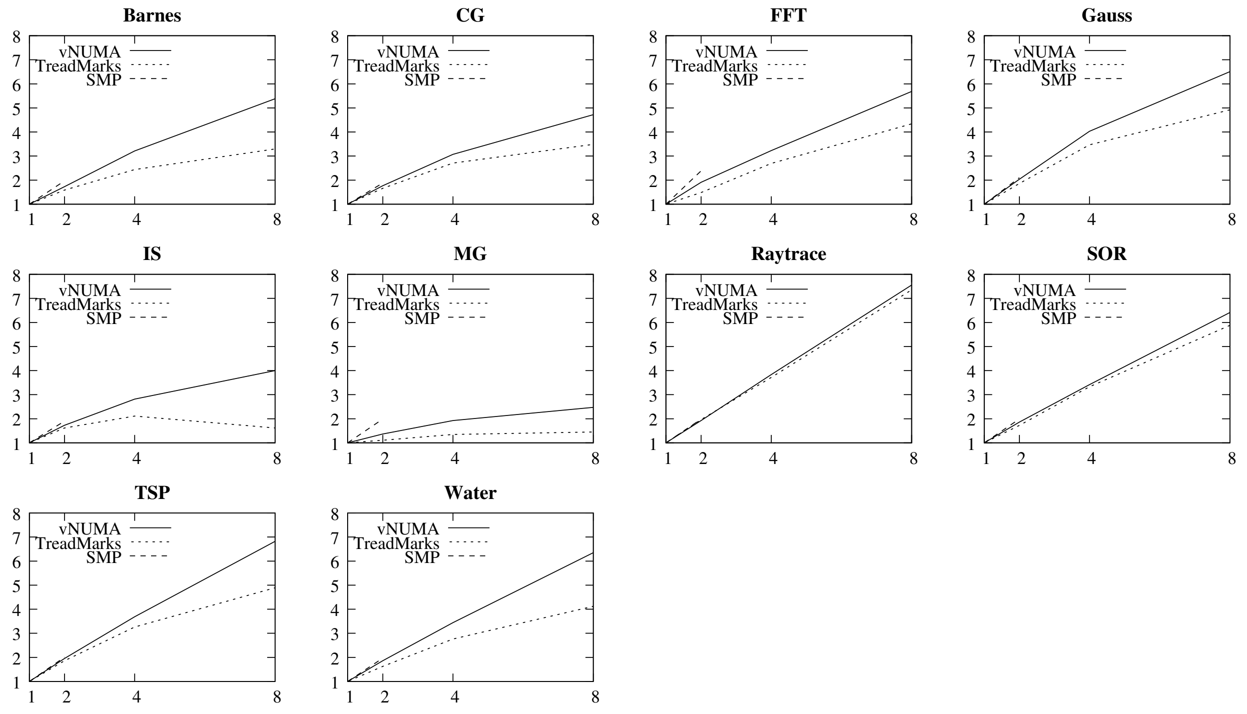


Figure 9: HPC benchmark performance summary. Horizontal axes represent number of nodes, vertical axes represent speed-up.

CPUs available).

Figure 9 shows an overview of results for each benchmark. While the ultimate limits of scalability are difficult to establish without a much larger cluster, vNUMA was designed for optimal performance on a small cluster. As the graphs show, vNUMA scalability is at least as good as TreadMarks on all benchmarks, and significantly better on **Barnes**, **Water**, **TSP** and **IS**. In absolute terms **MG** exhibits the poorest scalability, but it is a benchmark that poses challenges for all DSM systems, due to the highly irregular sizes of its three-dimensional arrays.

5.3 Compile benchmark

Large servers and clusters are frequently used for software builds. Figure 10 compares vNUMA's scalability with `distcc` [29] when compiling vNUMA. As compilation throughput tends to be significantly affected by disk performance, we eliminated this factor by building on a memory file system (RAM disk).

The figure shows that vNUMA scales almost exactly as well as `distcc`. The line labelled "Optimal" is an extrapolation of SMP results, based on an idealised model where the parallelisable portion of the workload (86 %) scales perfectly. On 4 nodes, the ideal speed-up is 2.8, while both vNUMA and `distcc` achieve 2.3. On 8 nodes, the ideal speed-up is 4.0, while both vNUMA and `distcc` achieve 3.1.

In the case of `distcc`, the overheads stem from the centralised pre-processing of source files (which creates a bottleneck on the first node), as well as the obvious overheads of transferring source files and results over the network. In the case of vNUMA, the largest overhead is naturally the DSM system. Of the 15 % overhead accountable to vNUMA in the four node case, DSM stalls comprise 7 %, the cost of intercepting writes is around 3 %, network interrupt processing around 2 % and other virtualization overheads also around 2 % (see also Section 5.4).

The majority of the DSM stalls originate from the guest kernel. This is because the compiler processes do not themselves communicate through shared memory. Their code pages are easily replicated throughout the cluster and their data pages become locally owned. However, inputs and outputs are read from and written to

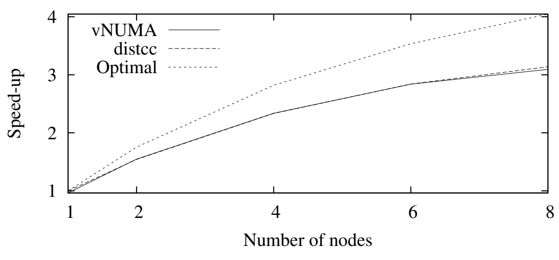
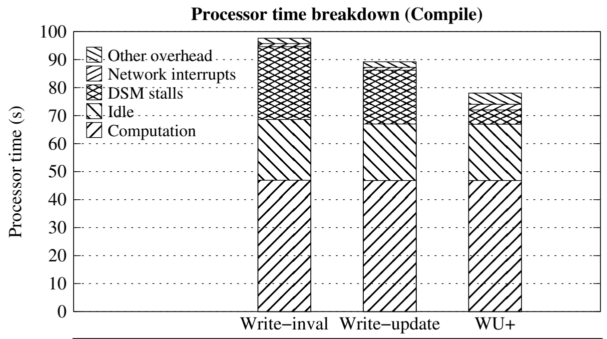


Figure 10: Compile benchmark performance summary



	Write−inval	Write−update	WU+
Processor time	97.6s	89.2s	78.0s
Speed-up	1.84	2.02	2.34
DSM stall time	26.0s (27%)	19.2s (22%)	5.1s (7%)
Stalls	420,000	285,000	164,000
Data fetches	276,000	187,000	28,000
Buffered writes	0	435,000	3,284,000
Write notices	0	47,000	496,000

Figure 11: Effect of protocol on compile benchmark

the file system, which shifts the burden of communication onto the kernel. In general, the compile benchmark can be considered representative of an application that consists of many processes which do not interact directly but interact through the filesystem.

Profiling the kernel overheads shows that the largest communication costs arise from maintaining the page cache (where cached file data is stored), and acquiring related locks. Similarly the file system directory entry cache (which caches filenames), and related locks, also feature as major contributors. Nonetheless, considering that the overall overhead is no greater than that of `distcc` — a solution specifically crafted for distributed compilation — this seems a small price to pay for the benefits of a single system image.

5.4 Effect of DSM protocol optimisations

To quantify the benefits of the chosen DSM protocols, we also executed the compile benchmark at three different levels of protocol optimisations: using the basic Ivy-like write-invalidate protocol, using our write-update protocol, and using our write-update-plus (WU+) protocol which intercepts atomic operations as well as ordinary writes. The results are summarised in Figure 11.

Performance is improved significantly by the more advanced protocols, with speed-up on four nodes increasing from 1.84 to 2.02 to 2.34. This is due to a sharp reduction in the number and latency of stalls. With the write-invalidate protocol, 420,000 synchronous stalls are incurred, totalling 26.0 seconds (an average of 62 μs/stall, which is dominated by the high latency of fetching page data that is required in 66% of cases). The write-update protocol reduces the number of synchronous stalls to 285,000, with a proportional decrease in stall time to 19.2s. However, the write-update-plus protocol has the most dramatic impact, reducing stall time to only 5.1s. While the total number of stalls is still 164,000, the majority of these are now ownership transfers, which involve minimum-length packets and therefore have low latency (17 μs in the common case). The number of stalls that must fetch data has decreased to only 28,000, which shows the effectiveness of this protocol in enhancing read-caching.

The price of this improved read-caching is that many more writes must be intercepted and propagated, which is reflected in higher overheads both for intercepting the writes (reflected in hypervisor overhead) and at the receivers of the write notices (reflected in interrupt overhead). Nonetheless there is still a significant net performance improvement.

5.5 Database benchmark

Databases present a third domain where high-end servers and clusters are used. We benchmarked PostgreSQL [30], one of the two most popular open source database servers used on Linux. The open-source nature was important to be able to understand performance problems. For the same reason — ease of understanding — simple synthetic benchmarks were employed instead of a complex hybrid workload such as TPC-C. Two tables were initialised with 10,000 rows each: one describing hypothetical users of a system, and the other representing posts made by those users on a bulletin board. A pool of client threads then performed continuous queries on these tables. The total number of queries completed in 30 seconds (after 5 seconds of warm-up) is recorded. This is similar in principle to benchmarks like TPC-C, but utilizes a smaller number of tables and a simpler mix of transactions.

Four different types of queries were used: **SELECT**, which retrieves a row from the users table by matching on the primary key; **SEARCH**, which retrieves a row from the users table by searching a column that is not indexed; **AGGREGATE**, which sums all entries in a certain column of the users table, and **COMPLEX**, which returns information about the five most prolific posters (this involves aggregating data in the posts table, and then performing a 'join' with the user table).

The results are summarised in Figure 12. vNUMA performs well for **COMPLEX**, which involves a base throughput of tens of queries a second. However, performance is degraded for the higher-throughput workloads, **SEARCH** and **AGGREGATE**, and most significantly so for **SELECT**, which involves little computation per query and can thus usually achieve thousands of queries a second on a single node. **SEARCH** and **AGGREGATE** barely manage to regain single-node perfor-

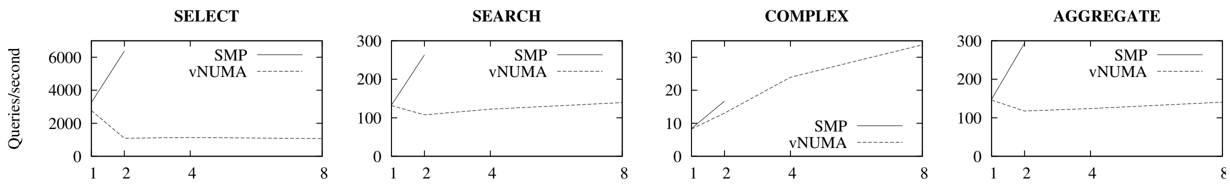


Figure 12: Database benchmark performance summary. Horizontal axes represent number of nodes.

mance on 8 nodes, while **SELECT** does not scale at all.

The cause of this throughput-limiting behaviour is simple: using multiple distributed nodes suddenly introduces the potential for much larger communication and synchronisation latencies. If one considers that each query involves at least a certain number of these high-latency events, then the maximum query throughput per node is inversely proportional to the number and cost of those events.

A breakdown of processor time usage for **SELECT** shows that only 14 % of available processor time is used for user-level computation, which explains why the four nodes cannot match the performance of a single node. Another 12 % is spent idle, which occurs when the PostgreSQL server processes are waiting to acquire locks. DSM stalls account for 57 % of processor time, with three-quarters of that being in userspace and specifically in the PostgreSQL server processes, and the other quarter in the Linux kernel. There is 9 % overhead for logging writes for the write-update protocol, and 2 % virtualization overhead (while **SELECT** normally experiences high virtualization overheads, the fact that it is only running 14 % of the time makes the virtualization overhead insignificant).

Further analysis, using performance counters, confirms that the major overheads are related to locking within PostgreSQL. The system uses multiple layers of locks: spinlocks, "lightweight" locks built on spinlocks, and heavyweight locks built on lightweight locks. Importantly, each heavyweight lock does not use its own lightweight lock, but there are a small number of contiguous lightweight locks which are used for protecting data about all of the heavyweight locks in the system. Thus, contention for this small number of lightweight locks can hamper the scalability of all heavyweight locks. In addition to this bottleneck, the multi-layer design substantially increases the potential overheads when lock contention occurs.

While this result is disappointing for vNUMA, it is not reasonable to extrapolate from PostgreSQL and assume that all database software will experience such severe locking problems. Since vNUMA can provide high levels of read replication and caching — and potentially a large amount of distributed RAM that may be faster than disk — designs that allow lock-free read accesses to data, such as via read-copy-update techniques [12,25], could theoretically provide very good performance. In this case, kernel performance would again become the ultimate challenge.

6 Related Work

Ivy [23] is the ancestor of most modern DSM systems. Ivy introduced the basic write-invalidate DSM protocol that forms an integral part of vNUMA's protocol. Mirage [11] moved the DSM system into the OS kernel, thus improving transparency. It also attempted to address the page thrashing problem, which was mentioned earlier in Section 4.1. Ivy and Mirage were followed by a large number of similar systems [28].

Munin [5] was the first system to leverage release consistency to allow multiple simultaneous writers. Aside from release consistency, other systems have also implemented entry consistency (Midway [4]), scope consistency (JIAJIA [9], Brazos [33]) and view-based consistency (VODCA [15]), which further relax the consistency model by associating specific objects with critical sections. However, all of these systems rely on the programmer to adhere to a particular memory synchronisation model, and thus they are not suitable for transparent execution of unmodified applications.

Recently there has also been much interest in virtualization, with systems such as Xen, VMware ESX Server and Microsoft Virtual Server making inroads in the enterprise. The majority of hypervisors are designed for the purposes of server consolidation, allowing multiple OS instances to be co-located on a single physical computer. vNUMA is, in a sense, the opposite, allowing multiple physical computers to host a single OS instance.

Since our initial work [7], three other systems have emerged which apply similar ideas to vNUMA: Virtual Iron's VFe hypervisor [34], ScaleMP's vSMP [32] and the University of Tokyo's Virtual Multiprocessor [18]. While these systems all combine virtualization with distributed shared memory, they are limited in scope and performance, and do not address many of the challenges that this work addresses. In particular, both VFe and the Tokyo system use simpler virtualization schemes and distributed shared memory protocols, resulting in severe performance limitations, especially in the case of Virtual

Multiprocessor. Virtual Iron attempted to address some of these performance issues by using high-end hardware, such as InfiniBand rather than Gigabit Ethernet. However, this greatly increases the cost of such a system, and limits the target market. Virtual Iron has since abandoned the product for commercial reasons, which largely seems to stem from its dependence on such high-end hardware. vNUMA, in contrast, demonstrates how novel techniques can achieve good performance on commodity hardware.

Little is known about vSMP, other than that it runs on x86-64 hardware and also relies on InfiniBand. The company claims scalability to 128 nodes, but only publishes benchmarks showing the performance of (single-threaded) SPEC benchmarks. No real comparison with vNUMA is possible with the information available.

7 Conclusions and Future Work

We have presented vNUMA, a system that uses virtualization to present a small cluster as a shared-memory multiprocessor, able to support legacy SMP/NUMA operating-system and multiprocessor applications. This approach provides a higher level of transparency than classical software DSM systems. Implementation in the hypervisor also has the advantage that many operations can be implemented more efficiently, and can make use of all the features of the underlying processor architecture. However, a faithful mirroring of the underlying ISA is required.

The different trade-offs resulted in protocols and implementation choices that are quite different from most existing DSM systems. Specifically, we developed a protocol utilizing broadcast of write-updates, which adaptively transitions between write-update/multiple-writer, write-update/single-writer and write-invalidate modes of operation. We also designed a deterministic incremental merge scheme that can provide true write coherence.

The evaluation showed that vNUMA scales significantly better than TreadMarks on HPC workloads, and equal to `distcc` on compiles. Database benchmarks showed the limitations of vNUMA for workloads which make extensive use of locks.

At the time this project was commenced (2002), Itanium was envisaged as the commodity system of the future, a 64-bit replacement of x86. This clearly has not happened, and as such, hardware supporting the present vNUMA implementation is not exactly considered "commodity", widespread deployment of Itanium systems in HPC environments notwithstanding. We are therefore investigating a port of vNUMA to AMD64 platforms. Some optimisations, such as those described in Section 4.3, will not apply there, but there is scope for other architecture-specific optimisations.

References

[1] James K. Archibald. A cache coherence approach for large multiprocessor systems. In *2nd Int. Conf. Supercomp.*, pages 337–345, 1988.

[2] Amnon Barak, Oren La'adan, and Amnon Shiloh. Scalable cluster computing with MOSIX for Linux. In *Proceedings of Linux Expo '99*, pages 95–100, 1999.

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *PPOPP*, pages 168–176. ACM, 1990.

[4] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, 1991.

[5] John B. Carter. Design of the Munin distributed shared memory system. *J. Parall. & Distr. Comput.*, 29:219–227, 1995.

[6] Matthew Chapman. *vNUMA: Virtual Shared-Memory Multiprocessors*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Mar 2009.

[7] Matthew Chapman and Gernot Heiser. Implementing transparent shared memory on clusters using virtual machines. In *2005 USENIX*, pages 383–386, Anaheim, CA, USA, Apr 2005.

[8] Xavier Defago, Andre Schiper, and Peter Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surveys*, 36:372–421, 2004.

[9] M. Rasit Eskicioglu, T. Anthony Marsland, Weiwu Hu, and Weisong Shi. Evaluation of JIAJIA software DSM system on high performance computer architectures. In *32nd HICSS*, 1999.

[10] D. Bailey et al. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, Mar 1994.

[11] Brett D. Fleisch and Gerald J. Popek. Mirage: A coherent distributed shared memory design. In *12th SOSP*, pages 211–223, 1989.

[12] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximising locality and concurrency in a shared memory multiprocessor operating system. In *3rd OSDI*, pages 87–100, New Orleans, LA, USA, Feb 1999.

[13] Ganesh Gopalakrishnan, Dilip Khandekar, Ravi Kuramkote, and Ratan Nalumasu. Case studies in symbolic model checking. Technical Report

UUCS-94-009, Dept of Computer Science, University of Utah, 1994.

[14] Charles Gray, Matthew Chapman, Peter Chubb, David Mosberger-Tang, and Gernot Heiser. Itanium — a system implementor's tale. In *2005 USENIX*, pages 264–278, Anaheim, CA, USA, Apr 2005.

[15] Zhiyi Huang, Wenguang Chen, Martin Purvis, and Weimin Zheng. VODCA: View-oriented, distributed, cluster-based approach to parallel computing. In *6th CCGrid*, 2001.

[16] Intel Corp. *A Formal Specification of Intel Itanium Processor Family Memory Ordering*, Oct 2002. `http://www.intel.com/design/itanium2/documentation.htm`.

[17] Intel Corp. *Itanium Architecture Software Developer's Manual*, Jan 2006. `http://www.intel.com/design/itanium2/documentation.htm`.

[18] Kenji Kaneda. Virtual machine monitor for providing a single system image. `http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/dvm/`.

[19] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *1994 Winter USENIX*, pages 115–131, 1994.

[20] R.E. Kessler and Miron Livny. An analysis of distributed shared memory algorithms. In *9th ICDCS*, pages 498–505, 1989.

[21] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21:558–565, 1978.

[22] Joshua LeVasseur, Volkmar Uhlig, Yaowei Yang, Matthew Chapman, Peter Chubb, Ben Leslie, and Gernot Heiser. Pre-virtualization: soft layering for virtual machines. In Y-C Chung and J Morris, editors, *13th IEEE Asia-Pacific Comp. Syst. Arch. Conf*, pages 1–9, Hsinchu, Taiwan, Aug 2008. IEEE Computer Society Press.

[23] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comp. Syst.*, 7:321–59, 1989.

[24] Daniel J. Magenheimer and Thomas W. Christian. vBlades: Optimised paravirtualisation for the Itanium processor family. In *3rd USENIX-VM*, pages 73–82, 2004.

[25] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *10th IASTED Int. Conf. Parall. & Distr. Comput. & Syst.*, Las Vegas, NV, USA, Oct 1998.

[26] Message Passing Interface Forum. MPI: A message-passing interface standard, Nov 2003.

[27] Christine Morin, Renaud Lottiaux, Geoffroy Vallée, Pascal Gallard, David Margery, Jean-Yves Berthou, and Isaac D. Scherson. Kerrighed and data parallelism: cluster computing on single system image operating systems. In *6th IEEE Int. Conf. Cluster Comput.*, pages 277–286, 2004.

[28] Bill Nitzberg and Virginia Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Comp.*, 24(8):52–60, Aug 1991.

[29] Martin Pool. distcc, a fast free distributed compiler. In *5th Linux.Conf.Au*, Jan 2004. `http://distcc.samba.org/`.

[30] PostgreSQL Global Development Group. PostgreSQL database software. `http://www.postgresql.org/`.

[31] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *11th ISCA*, pages 340–347, 1984.

[32] The Versatile SMP (vSMP) architecture and solutions based on vSMP Foundation. ScaleMP White Paper.

[33] Evan Speight and John K. Bennett. Brazos: A third generation DSM system. In *1st USENIX Windows NT WS*, pages 95–106, 1997.

[34] Alex Vasilevsky. Linux virtualization on Virtual Iron VFe. In *2005 Ottawa Linux Symp.*, Jul 2005.

[35] Bruce J. Walker. Open single system image (openSSI) Linux cluster project. `http://www.openssi.org/ssi-intro.pdf`, accessed on 30th September 2008.

[36] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd ISCA*, pages 24–36, 1995.

[37] Matthew J. Zekauskas, Wayne A. Sawdon, and Brian N. Bershad. Software write detection for a distributed shared memory. In *1st OSDI*, pages 87–100, 1994.