Estimating Memory Locality for Virtual Machines on NUMA Systems

by

Alexandre Milouchev

S.B., Massachusetts Institute of Technology (2011)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2013

Author .....
Department of Electrical Engineering and Computer Science
May 16, 2013

Certified by.....
Prof. Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Prof. Dennis M. Freeman
Chairman, Master of Engineering Thesis Committee

Estimating Memory Locality for Virtual Machines on NUMA Systems

by

Alexandre Milouchev

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The multicore revolution sparked another, similar movement towards scalable memory architectures. With most machines nowadays exhibiting non-uniform memory access (NUMA) properties, software and operating systems have seen the necessity to optimize their memory management to take full advantage of such architectures. Type 1 (native) hypervisors, in particular, are required to extract maximum performance from the underlying hardware, as they often run dozens of virtual machines (VMs) on a single system and provide clients with performance guarantees that must be met.

While VM memory demand is often satisfied by CPU caches, memory-intensive workloads may induce a higher rate of last-level cache misses, requiring more accesses to RAM. On today's typical NUMA systems, accessing local RAM is approximately 50% faster than remote RAM.

We discovered that current-generation processors from major manufacturers do not provide inexpensive ways to characterize the memory locality achieved by VMs and their constituents. Instead, we present in this thesis a series of techniques based on statistical sampling of memory that produce powerful estimates for NUMA locality and related metrics. Our estimates offer tremendous insight on inefficient placement of VMs and memory, and can be a solid basis for algorithms aiming at dynamic reorganization for improvements in locality, as well as NUMA-aware CPU scheduling algorithms.

Thesis Supervisor: Prof. Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

When I first spoke to Puneet Zaroo on the phone in the Fall of 2010 about a potential M.Eng. project at VMware under MIT's VI-A program, I didn't really know where it would take me. I felt I had general knowledge in computer science, but no experience whatsoever in operating systems. Puneet interviewed me and presented the project he had in mind and, to my surprise, offered me a position a few weeks later, which I accepted after he assured me of his belief that my qualifications would be strong enough to get the job done. Soon after graduation, I joined VMware and made my first baby steps reading the source code of the ESX kernel. My lack of knowledge was immediately apparent, which led to a period of getting up to speed that can be described perfectly with a popular MIT expression – "drinking from the fire hose." I would like to thank Puneet for his confidence in my abilities, and for putting up with the incessant flow of simple questions that I addressed his way while starting out.

I would like to thank both Puneet Zaroo and Vladimir Kiriansky for their deep involvement in my project, from its very inception through to the end of my first 6 months at VMware. I highly appreciated our weekly meetings, at first face to face and later over the phone, during which an entire hour was dedicated solely to my questions and to brainstorming about the project. These precious hours taught me how to methodically consider all aspects of a given problem; how to conduct good research; and how to tackle issues that appear to be showstoppers at first sight. I would also like to thank Alex Garthwaite, for taking me under his wing after I relocated from the VMware headquarters to the Cambridge, MA office for the Fall of 2011. His remarkable enthusiasm, in-depth knowledge about the ESX virtual machine monitor, and endless flow of ideas were of great help and daily motivation in an office with very few familiar faces.

Many thanks are due to Rajesh Venkatasubramanian, who accepted to be my mentor for my second internship at VMware in the summer of 2012. Rajesh gave a new direction to the project and was the ESX guru I needed every time I hit a roadblock.

For the last stretch of my M.Eng., I would like to address a big thank you to Rajesh Venkatasubramanian and Prof. Saman Amarasinghe, who was kind enough to work with me as my MIT thesis supervisor. Writing and submitting this thesis would not have been possible without their encouragements and reassurances, as well as their invaluable guidance and feedback.

I will not forget to thank Igor Braslavsky, my manager at VMware, who was always one call away for any issue that arose and made it possible for me to continue working at VMware even after the required internship for the VI-A program had been completed.

Last but not least, I would like to thank my close friends and family for their infinite support and positive energy in the many times that I was at my lowest. My warmest thanks go to my father, mother, and brother, who were there every step of the way and even proofread my thesis, although it was not at all in their area of expertise. This achievement is as much yours as it is mine.

Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

1.1 The NUMA Architecture

First invented in 1947 [4], the transistor quickly replaced the vacuum tube as the three-terminal device of choice for the construction of complex logic circuits. The invention of the integrated circuit followed thanks to two key innovations in 1958 and 1959, solving the problems encountered when assembling very large circuits made of individual transistors. The first programmable microprocessors were right around the corner of the next decade, with Intel's 4004 chip being commercialized in 1971. Since then, in accordance with Moore's Law [27], transistor densities have followed an exponential increase, with the number of transistors in a CPU doubling roughly every 18 months, allowing for more complex logic on a chip of the same size. This upward trend has been a reality for more than 50 years and is expected to continue at least for the next 10 years, if not much longer.

However, more transistors in a CPU do not readily translate to faster performance. Consequently, chip designers make use of a large arsenal of techniques in order to squeeze the most out of a single chip. One such technique is pipelining – separating the execution of an instruction into numerous stages, each taking a fraction of the total execution time. Among other positives, pipelined processors can take advantage of instruction level parallelism (ILP), which allows them to have multiple instructions executing different stages of the pipeline at the same time.

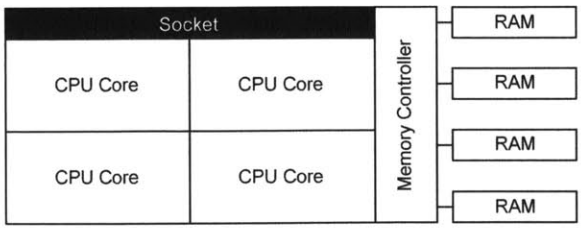


Figure 1-1: Simplified schematic of a quad-core SMP system.

To increase the number of instructions being executed in parallel, modern processors have very deep pipelines. In order to keep the single-instruction latency low, these processors need to have increasingly higher clock speeds, which has recently become a significant problem. A higher clock speed causes transistors to switch faster, which increases their power consumption and the amount of heat they generate. Transistors' power demand is worsened further by the fact that at the current scales of miniaturization, transistor gates, which control current flow, tend to be less capable of blocking the passing of electrons. Therefore, transistors waste power even when they are not switching [14]. It is for these reasons that manufacturers have turned to techniques other than increasing clock speed to increase performance. One natural way of maintaining the promise of Moore's Law is to create multi-core chips: CPUs that pack more transistors, but in separate cores, speeding up execution by having all cores execute different threads or processes [25, 28]. Such architecture are commonly referred to as SMP, or Symmetric Multiprocessing, as there are multiple identical processors connected to a single shared main memory. An example SMP architecure is shown in Figure 1-1.

The benefits of migrating to multi-core have been demonstrated extensively. A recent study by chip manufacturer Intel [26] investigated the trade-offs between overclocking a single-core processor and using the same core in a dual-core setup, but at a lower clock speed. Overclocking the core by 20% and using it as a single core CPU resulted in a 73% increase in its power requirement, but only a 13% performance gain. In contrast, using it in a dual-core setup, each core underclocked by 20% resulted in a mere 2% increase in the overall required power, but an admirable 73% increase in performance compared to using the single-core processor at its designated frequency.

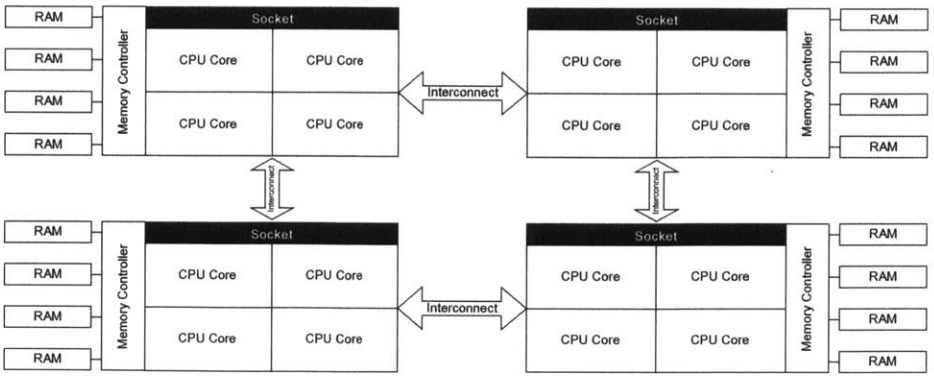


Figure 1-2: Simplified schematic of a generic 4-node NUMA system. An alternative design might include two additional interconnects forming an "x" in the middle, effectively ensuring every node is within 1 "hop" of every other node.

Multi-core processors have been prevalent for many years now, and are generally accepted as the most viable way of scaling CPU performance in the foreseeable future [3, 15]. However, processing power is only one of the factors to consider when evaluating system performance. Memory speed is another key factor, and it has alarmingly lagged behind CPU speeds for decades. This disparity has lead to the inclusion of complex cache hierarchies in today's hardware. Despite this, many workloads are memory-bound – that is, memory is the main bottleneck in their execution. As a result of high access latency, the CPU can become "starved for memory". In other words, no further instructions can be executed until data has been retrieved from memory in order to continue the computation.

While already concerning on single-core CPUs, this problem is only exacerbated on multi-core. Not one, but many cores can become stalled waiting for I/O at once due to access latency or the limited bandwidth available on the memory bus. This issue undermines the benefits of concurrent execution, and only worsens with the increase of the number of cores on a CPU. Therefore, a new, more scalable architecture is necessary to extract the full benefits of multi-core parallelism.

Non-Uniform Memory Access (NUMA) multiprocessors are designed with memory starvation in mind. Under the NUMA architecture, CPU cores are arranged in packages, or *nodes*, and each such package is connected to local RAM for quick access [22]. An example can be seen in Figure 1-2. As each processor uses a separate memory

bus, the theoretical promise of NUMA is performance scaling roughly proportional to the number of nodes, or memory banks.

Despite being distributed throughout the system, memory in NUMA is still presented to the programmer as a global, shared address space. Any memory location can be accessed by any CPU. This is achieved by linking the different nodes in the system with high-speed interconnects. As with regular SMP, memory performance is further improved by the use of a hierarchy of caches at each node. It is important to note that initial NUMA designs did not implement cache coherence across nodes, which meant processors were not guaranteed to retrieve the latest updated data in case the memory reference they were accessing was found in their local cache, but had already been modified on another node. Although easier to design and manufacture, this model was found to prohibitively increase the complexity of programming for such systems. As a result, nowadays NUMA machines are implied to be, in fact, ccNUMA (cache-coherent NUMA).

Under NUMA, memory references from a CPU's point of view can be divided into remote ones, which reside on other nodes, and local ones, which are stored in the CPU's local bank. When a CPU accesses memory, it first queries its local caches. If no level in the hierarchy contains the required data and the address is local, it will be retrieved from the local RAM. On the other hand, if it is remote, the CPU has to stall while memory is accessed over the high-speed interconnect. The non-uniform characteristics for NUMA systems are due to the increased latency penalty incurred when going over the interconnect, because of the greater physical distance covered. Typically, the fastest access times are achieved when hitting a local cache (from less than 1 to a few nanoseconds), followed by hitting a remote cache (dozens of nanoseconds), followed by an access to local memory (~60ns), and lastly an access to remote memory (~100ns). Approximate latencies for recent Intel processors are shown in Table 1.1.

Clearly, NUMA will favor some types of workloads over others. For example, workloads with small working sets that are mostly contained in caches should not experience slowdowns due to the distributed nature of the system. For memory-

Table 1.1: Approximate latencies for the individual data sources corresponding to an L2 cache miss on Intel Core i7 Processor and Intel Xeon 5500 processors [21].

Data Source	Latency
L3 CACHE hit, line unshared	~ 40 cycles
L3 CACHE hit, shared line in another core	~ 65 cycles
L3 CACHE hit, modified in another core	~ 75 cycles
remote L3 CACHE	~ 100-300 cycles
Local DRAM	~ 60 ns
Remote DRAM	~ 100 ns

intensive workloads, however, good performance can be achieved only if the data can be spread across the system, such that each processor can load data only (or predominantly) from its local bank, and thus avoids expensive remote accesses.

Unfortunately, due to the dynamics of CPU scheduling, load balancing, memory allocations, and several other factors, achieving sufficient *locality of accesses* in the general case is difficult. Different operating systems have taken different approaches to this problem. In this thesis, we will discuss the optimizations implemented in ESXi, VMware, Inc's type 1 hypervisor. We also explore means to further improve performance of virtual machines (VMs) on NUMA hardware. Specifically, we study new techniques for locality estimation and how this data can be used to improve performance. We show that statistical sampling of memory pages is a viable, efficient approach to estimating locality of virtual machines, whereas other potential methods do not provide the level of detail we need to implement further optimizations.

1.2 The Problem of Memory Locality

With NUMA, targeted allocation and migration of data and code in memory becomes essential. The number of remote accesses by each processor must be minimized, or else not only would any potential advantages of NUMA be negated, but performance might suffer even further than on a symmetric architecture, due to the high interconnect latency. This "locality" problem can be addressed in a variety of ways, none of which are mutually exclusive.

The first possibility is to expose the NUMA topology, and to offload the decision-

making to the programmer. This by itself can hardly be a solution, as it imposes unnecessary burden on application writers and will lead to slower development, less efficient software, or both. For example, here are some challenges the developers would face:

1. accounting for current and future NUMA topologies,
2. keeping track of data migration and locality,
3. predicting where remote accesses might happen,
4. duplicating memory in order to improve performance,
5. keeping duplicated memory consistent.

In addition, it will be difficult to change the existing mindset that assumes uniform access time to all of memory, which has been ingrained for the entire history of computing. Furthermore, modern operating systems (OS) provide isolation guarantees to running processes to shield them from improper interaction. Subsequently one application cannot have full knowledge of the state of the OS and all running programs, and is therefore limited in the optimizations it can aim to achieve.

A similar approach, but easier on the developers, would be to introduce libraries that provide useful abstractions, or compilers that automatically lay out memory in a NUMA-aware fashion. It is unclear, however, how well the compiler could optimize when presented only with the code and not the target topology – assuming that no single topology will dominate the others, and that users will not be expected to recompile applications to match the platform they are using. As for software libraries, they could provide limited benefits, but again would depend on the programmer making good use of the functionality provided. Not unlike compilers, it is hard to imagine that a general-purpose library could fine-tune NUMA performance for every possible program and topology.

A third approach would be to include optimizations at the operating system level. This is a particularly attractive option, as the OS controls every layer of execution

and has full knowledge of the topology it runs on, and the current state of the system. In this thesis, we concentrate exclusively on OS-level improvements.

In the future, it might be worthwhile to investigate coupling the aforementioned techniques for best results. Although the OS has full knowledge of the machine state and architecture, it can only indirectly monitor and make predictions on the most likely short- and long-term changes in the behavior of running applications. The applications could, through exposed libraries, inform the OS of the key features of their memory layouts and usage patterns, simplifying the job of the OS and allowing for maximal gains.

1.3 Our Approach

As mentioned above, this thesis will focus on OS-level optimizations. In traditional SMP (the most common form of UMA, or Uniform Memory Access architecture), all processors (or cores) share one memory bus, and therefore have uniform access time to all of memory. The main focus of modern operating systems' memory management modules is their paging policy: which pages to fetch into memory, which frame to load them into, and which pages to swap to disk in order to make room for new ones. The most attention is typically given to the algorithm for selection of pages to swap in/out, to reduce the occurrence of problems such as thrashing, where the same pages continuously get flushed to disk and accessed soon afterwards, bringing about a heavy performance hit.

With the advent of NUMA, new aspects need to be considered. For example, the importance of memory placement has risen dramatically, so which pages to fetch matters just as much as where in memory these pages are loaded. What is more, it is no longer enough to fetch a page and keep it in memory if it is accessed frequently. Often, processes will be scheduled to run on various nodes rather than stick to a single one, depending on the load distribution in the system; so, memory that was once local to a process may suddenly become remote. Therefore, dynamic detection of changes in locality and proactive migration of pages, as well as locality-aware scheduling, are

needed to keep performance high.

In order to make informed decisions on NUMA memory placement and CPU scheduling, it is essential to collect accurate data on current placement of memory and on access patterns, so as to predict the level of locality that can be achieved in the system, and identify which possible actions exhibit the lowest cost-benefit ratio. The main contribution of this thesis is the development of novel methods for locality profiling and analysis, with a proof-of-concept demonstration implemented on VMware ESX/ESXi.

The central technique presented consists of invalidating the mappings of sets of memory pages, forcing the CPUs to page-fault when accessing those pages. Upon faulting, the accesses are logged for later aggregation. We make extensive use of statistical sampling to get detailed information for each VM and its constituent virtual CPUs (vCPUs). With aggregate data in hand, we can enable quick response to changes in locality, precise fine-tuning of present and future data and code placement, detection of candidate pages and VMs for migration across the system, and many other NUMA optimizations. While we do not collect data for every active page in a VM, we focus on computing the general trend in memory usage. Trend data can be applied, for instance, to ration the allocations of VMs depending on their entitlement, in a NUMA-aware fashion.

1.4 Organization of this Thesis

The organization of this thesis is as follows. Chapter 2 introduces the current state-of-the art techniques employed in popular operating systems, including VMware ESX. We describe the various techniques in detail, compare the approaches, and present the shortcomings of the current implementations.

In Chapter 3, we explore the possible ways of estimating memory locality on various hardware platforms. We describe the currently available technology, such as performance counters, and how suited it is for the task. After a discussion of the alternatives, we dwell on the suitability and promise of statistical sampling as the

method of choice.

Chapter 4 exposes the implementation details of the ideas presented in Chapter 3. We start by evaluating the existing logic in ESX, then describe a series of augmentations that form the basis of this thesis. In particular, we describe techniques for detecting a virtual machine's per-node locality, and per-vCPU per-node locality. Per-vCPU estimation requires extensions which, incidentally, allow for direct detection of inter-vCPU sharing of data. This data can be used, for instance, as a hint to schedule vCPUs on nearby nodes, so their shared data resides in a lower-level cache.

Experimental results using a proof-of-concept implementation are presented in Chapter 5. An introductory section describes the experimental setup: the hardware configuration and the benchmark used. Section 5.2 covers 4 classes of experiments that were used to evaluate the correctness and applicability of the implementation. A discussion follows in section 5.3.

Chapter 6 summarizes the results of the thesis and suggests future work in the area. We present two specific applications of the new, more precise locality estimates. This includes an algorithm that was implemented as part of this thesis, but not thoroughly tested. The algorithm in question attempts to detect idle memory that unnecessarily clogs a node, forcing virtual machines to make remote allocations. Once detected, this idle memory can be evicted and scattered around the system to make space for active memory which will be accessed locally.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Current State-of-the-Art

In this chapter, we explore the NUMA optimizations implemented in various operating systems, and contemplate their potential shortcomings.

2.1 NUMA Optimizations on Different Operating Systems

2.1.1 VMware ESX/ESXi

ESXi server is in a unique position to implement NUMA optimizations, as it is not designed to run general-purpose software. As a type 1 hypervisor, ESXi runs virtual machines; therefore, rather than handling each process separately, it can group processes as pertaining to one and the same VM[1], and optimize the NUMA scheduling and placement of the entire VM and its memory.

VMware ESX/ESXi has boasted a NUMA-aware CPU scheduler at least since version 3.0. At boot, the kernel detects the NUMA topology it is running on, including number of sockets, number of cores on each node, and memory per node. After this initial detection, the scheduler uses that information to try to minimize remote accesses. Each VM is assigned a "home node" on startup, determined by the current

[1] A VM consists of a Virtual Machine Monitor (VMM), a helper process called the VMX, and a world for each one of its virtual CPUs (vCPUs).

state of the system so that CPU and memory loads remain balanced across nodes. A VM will preferentially allocate memory on its home node, and will also be scheduled to run on the home node as much as possible.

If a VM consists of more vCPUs than there are cores on a single node in the hardware, or if it has more memory than a single node can provide, then it is considered a Wide VM. In ESXi 3.5, Wide VMs were not assigned a home node and as such did not benefit from NUMA optimizations, suffering a performance hit. ESXi 4.1 improved on this by splitting Wide VMs into separate "NUMA clients," with each client being treated as a separate "small" VM.

However, clever initial placement is not enough for long-term, efficient NUMA scheduling. VMs powering off and new VMs getting started, as well as varying CPU and memory demand of the workloads inside the VMs may introduce load imbalance between NUMA nodes. To this end, ESXi periodically invokes a NUMA rebalancer, which performs two dynamic optimization. The first involves migrating VMs between NUMA nodes. A VM's home node gets reassigned, and its internal data structures get transferred to its new target node. This operation is expensive and might saturate the node interconnect, so it is only performed if it will be highly beneficial to performance and load balance in the long term.

The NUMA rebalancer also handles dynamic page migrations from remote locations to a VM's home node. This is regulated by continuously reassigning the VM's page migration rate, in proportion to the locality the VM is currently achieving. Pages are selected for migration using two concurrent methods: random selection and a linear scan, in order to avoid pathological cases. Immediately after a VM migrates, its page migration rate will be very high, as most of its pages will be left behind on its previous home node. Conversely, when high locality is achieved, the page migration rate will be low, to avoid scanning memory only to find pages that are already local to the VM's vCPUs.

In addition to dynamic migration of VMs and their memory, ESXi uses advanced sharing techniques to reduce VM's memory footprint and improve performance. ESXi capitalizes on the fact that many memory pages can be shared by different VMs. For

example, multiple VMs may be running the same application, or may have the same guest OS. In such cases, ESXi transparently serves the same copy of each shared page to the guests, avoiding unnecessary redundancy. On NUMA systems, the implementation has been overhauled to replicate copies of shared pages on each node, eliminating the need for VMs to access remote memory.

The final aspect to ESXi's handling of NUMA architectures was introduced in version 5.0. It allows users to expose the NUMA topology to a guest operating system using Virtual NUMA, or vNUMA. In this way, ESXi can leverage guest OS and application optimizations for NUMA systems. Enabling vNUMA also functions on non-NUMA systems, which is necessary for example when a virtual machine is migrated from a NUMA to a non-NUMA host system using VMware vMotion. In those cases, the underlying hardware memory access time is uniform for all CPUs, but the VM is fooled into thinking it runs on a NUMA topology.

2.1.2 Other Operating Systems

Most modern OSes are at least NUMA-aware, even if they do not actively optimize for it. With the increase in the popularity of NUMA, especially on server hardware, however, a stronger focus has been given to the problem of locality.

The Linux kernel has been NUMA-aware since version 2.6 [12]. Its default policy is to allocate memory locally, wherever a thread is currently running [7]. Threads are not bound to nodes by default, and as such are free to be rescheduled across NUMA domains. A NUMA API is available to applications through *libnuma* [2], with which they can specify application-specific policies. Administrators and users can also influence NUMA placement and behavior by using the *numactl* command-line utility, which allows processes to be bound to given *cpusets* [19]. If the *cpuset* of an application is subsequently modified, the process migrates to a CPU that is part of the new *cpuset*. A process' memory can be configured to also migrate upon changes in the allowed *cpuset*, but this behavior is disabled by default [20]. Lastly, several patches have been proposed to implement automatic page migration, but none have made it into a release version of the kernel so far [6].

Microsoft Windows provides similar enhancements to the ones already mentioned. It follows the traditional "initial placement" policy, and attempts to schedule processes on the same nodes as the memory they are accessing. New allocations are also satisfied locally. A NUMA API is available, which exposes the underlying topology to applications and allows them to set special policies and affinities at runtime [8].

Type 1 hypervisors are interesting to consider, given their typically more advanced NUMA support. In Microsoft Hyper-V Server, a popular virtualization platform, administrators are given the option of enabling or disabling "NUMA Spanning," thus specifying whether a VM is allowed to spill over more than one NUMA node. The latest version of the OS features an implementation of Virtual NUMA, enabling the guest operating systems' native NUMA optimizations [9]. No publicly available information could be found on adaptive VM and page migrations across nodes within a single NUMA system.

Another popular choice is the Xen Hypervisor, an open source alternative for server virtualization. The current version of Xen only supports pinning VMs to a node, not allowing its vCPUs to run on any other node than the one it is initially assigned to. In the next release, Xen will use the more flexible notion of "node affinity," which optimize for NUMA locality by mostly scheduling VMs and allocating their memory on their assigned nodes, while still allowing for them to be scheduled remotely as necessary for load balancing.

2.2 Shortcomings of Current Approaches

For the purposes of this thesis, we will discuss virtual machines from this point on, as opposed to individual processes. As discussed in section 2.1.1, it is insufficient to rely on static node bindings or predetermined node affinities to maintain high performance in the long term. Exposing a virtual NUMA topology is certainly a step forward within single VMs, and is a necessary requirement to getting the best performance out of wide VMs. However, VM interaction and overall system load balancing requires further optimizations, since VMs are, and should be, unaware of

and isolated from each other.

VMware ESXi's advanced dynamic migration mechanisms are the current state of the art in this respect. Even so, it is of interest to inquire how to improve the current ESXi implementation.

1. **VM migration**. In order to select VMs for migration, the NUMA scheduler needs to estimate their per-node memory access frequencies. Currently, ESXi uses total page allocations per NUMA node as a proxy for this. This metric is inaccurate at best. For example, consider the following scenario: vm_0 allocates 1GB of memory on $node_0$, and 200MB of memory on $node_1$. If vm_0's working set is a strict subset of the memory allocated on $node_1$, then vm_0 would highly benefit from being scheduled on that node. However, should rebalancing for a migration in the system, the total allocations metric would suggest vm_0 to be migrated to $node_0$, where most of its memory lies. This would lead to a majority of remote accesses and the corresponding decrease in performance, especially if the workload inside vm_0's guest OS is memory-bound. Therefore, a better memory locality estimation technique would improve edge cases of VM migration.

2. **Memory migration**. ESXi's memory migration works as follows: periodically, pages are picked, both at random and in a sequential scan. A page is migrated if it is remote, and ignored otherwise. This technique does not consider the relative "hotness" of pages, i.e. the current frequency at which they are being accessed. Consequently, unnecessary migrations can occur, since migrating an idle page does not benefit performance. In fact, a large number of such migrations can *hurt* performance, since the VM incurs the cost of copying each page and remapping it on its local node. Therefore, better targeting migrations at active pages can reduce overhead and lead to a quicker increase in performance.

As can be seen above, dynamic NUMA balancing could benefit from further improvements. In the next chapter, we examine ways to enhance ESXi's VM and memory migration mechanisms.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Estimating Memory Locality

In this chapter, we explore alternative methods to estimate memory locality on NUMA systems. Many of them hold substantial promise, but are not currently usable to fully enable us to meet our goals. We describe each in detail, and evaluate whether it might be improved in the future to fulfill our needs. Our own approach – using statistical sampling of memory – is discussed in section 3.2.

3.1 Alternatives Explored

3.1.1 Performance Counters

Prior research has suggested the use of hardware performance counters for locality estimation [13]. Modern processors include a Performance Monitoring Unit (PMU), which provides counters used to gather statistics on the operation of the processor and memory system. Different CPU manufacturers expose various micro-architectural events that can be recorded and counted by the PMU. On multi-core CPUs, each core usually provides separate counters, and there may also be counters available for socket-wide events related to the memory controller, caches, and other I/O.

Performance counters are of interest for two main reasons. First, they allow us to track all memory accesses, as opposed to statistical sampling of memory (described below) which only captures accesses to sampled pages. Second, if used correctly,

their impact on performance should be negligible – again, as opposed to our chosen alternative, which purposefully induces a page fault for each sampled access.

However, it was not clear prior to this writing whether current PMUs offer events applicable to NUMA locality profiling. We investigated the functionality provided on both Intel and AMD processors, and briefly present our findings below.

Intel

Processors based on the Intel Core microarchitecture support Precise Event-Based Sampling (PEBS) [11]. PEBS allows storing a subset of the architectural state when the counter for an event being tracked exceeds a certain threshold. This precise memory sampling focuses on loads rather than stores, since loads are typically responsible for long stalls in execution. PEBS can be enabled on last level cache misses, and thus can be used to track accesses to memory. The saved state includes the Instruction Pointer (IP), the values of the 16 general registers and, most notably, a virtual or linear address being accessed, reconstructed automatically from the instruction at IP and the saved register state.

However, because the state is saved after the trigger instruction executes, there are cases in which the virtual address cannot be reconstructed – namely, if the target of a load was the same register that initially held the virtual address. Because such an instruction is not uncommon, we cannot rely on this mechanism to provide us with the target virtual address reliably. Besides this issue, another major problem is that the PEBS buffer, in which the state is saved, can only be held at a guest linear address.

Recent Intel processors, such as the Intel Core i7, provide a "latency event" [21], which exhibits several enhancements. First, it can be configured to trigger only when the latency of a memory access exceeds a certain value. Second, since the event triggers before the load has finished, the virtual address can always be recovered. Lastly, latency events register the data source for each event in an additional register of the PEBS buffer.

Unfortunately, latency events also suffer severe drawbacks which render them

inapplicable for NUMA sampling. Like PEBS, they only operate with guest linear addresses. Also, the data source recorded for each event only differentiates between local and remote accesses, but does not provide fine-grain information on which node serviced the access. Finally, although both loads and stores are important when optimizing NUMA memory policies, these events only support load instructions.

AMD

Recent AMD processors, besides the regular per-core performance counters, also include four additional ones dedicated to counting *northbridge* events [18]. Northbridge counters can record, among other events, any data traffic that is handled by local and remote DRAM. Each memory controller in the system can count the number of read and write requests it fulfills [17], and as such, we can gather per-node access data. However, we only know that each access originated from the local socket, but we cannot tell which core executed the trigger instruction. This is not useful for our purposes, as there might be multiple VMs running on a socket, and it is important to know which one of them is accessing which data. It also precludes us from collecting information on a finer granularity than per VM (e.g. counting accesses from specific vCPUs), which is valuable for reasons examined later in this thesis.

Conclusion

In conclusion, latest-generation processors from two of the world's major manufacturers do not offer performance counters with functionality that can be used to gather per-node access information for each VM in the system. Therefore, an alternative approach that overcomes their various limitations needs to be found.

3.1.2 A/D Bits

Another mechanism for determining the location of a VM's hot pages would be to periodically examine the accessed/dirty bits of the VM's page tables. A/D bits are supported by both Intel's Extended Page Tables [11] and AMD's Nested Page Ta-

bles [16]. This approach has the advantage of providing a full picture, as opposed to requiring extrapolation from a randomly selected set of sample pages, as in 3.2.

However, there are several drawbacks. The first and most significant one is that if large pages are in use, all pages are likely to be either Accessed or Dirty, effectively masking accurate information on individual small pages' "hotness." Therefore, this approach would mandate the use of small page mappings only. This is undesirable, as large page mappings are key to reducing the overhead due to the increased cost of a page walk when nested page tables are enabled [5]. Second, no information is captured on which vCPU from within the VM initiated the access, so A/D bits can only provide information on a per-VM basis. This is not sufficient for wide VMs that spread over multiple nodes, as vCPUs on different nodes should be considered separately. Therefore, while we note that this approach may be valuable for identifying hot pages for migration, it is not sufficient for purposes of scheduling and NUMA client migrations.

3.2 Statistical Sampling of Memory

After initial investigation, we picked our approach of choice to be statistical sampling of memory. The idea is to randomly select a set of pages and invalidate their mappings in the TLB (Translation Lookaside Buffer) and the virtual MMU (Memory Management Unit). If any of these pages is subsequently accessed, it will lead to a page fault, which is processed by the VMM (Virtual Machine Monitor). In the page fault code path, the VMM can mark sampled pages as accessed, and at the end of a sampling period, the results can be extrapolated for a full picture of the spread of active pages across nodes. This gives us a good per-node estimate of memory activity, which can be used as the heuristic for VM migration.

For VMs that span more than one node, however, it is not sufficient to gather VM-wide per-node access pattern information. Rather, we are interested in per-vCPU per-node statistics, since memory local to one vCPU may be remote to another. For this reason, we propose to extend the sampling mechanism to provide per-vCPU data.

This can be done by invalidating pages' mappings not once, but many times over the sampling period. In this way, multiple accesses from different vCPUs can be detected to give us the necessary per-vCPU estimates.

Besides being useful for VM migration, the above estimates can also help in the case of memory migration. Namely, the selection of pages can be geared towards the most active remote nodes. While this method does not help individually detect all active pages, it could be useful on future hardware with dozens, if not hundreds of NUMA nodes, where identifying only the "hottest" nodes as targets will lead to an important difference in effectiveness.

Statistical sampling of memory successfully addresses the shortcomings of using performance counters or A/D bits for per-vCPU and per-node activity. In addition, such sampling has the advantage of being flexible in terms of performance and accuracy. We believe the performance impact of sampling to be minimal if using a sufficiently small sample set size, and infrequent enough invalidation of mappings. We found that small sample sizes (relative to a VM's memory size) were good enough for the purposes of NUMA scheduling and VM migration.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

Implementation

4.1 Current Sampling Technique in ESX

VMware ESX Server implements a proportional-share framework, which allows administrators to specify what portion of a shared resource a given client is entitled to using. In other words, in the case of memory, a VM with more shares will be allocated more physical pages than other VMs with smaller entitlement. As opposed to traditional proportional-share systems, however, ESX's algorithm is enhanced to consider the effects of idle memory on performance. In overcommitted scenarios – that is, more memory is requested by the clients than is available in the system – pages are reclaimed preferentially from clients whose allocation is not actively used.

It is precisely to detect such idle memory that ESX's current sampling technique was put into place[29]. The ESX kernel samples each VM independently. By default, four sample sets of pages are created at VM boot time. Each period, one of the sample sets is disposed of and populated with n pages selected uniformly at random. The state associated with these pages in the TLB and the virtual MMU is invalidated, such that upon the next access by the VM, a page fault will be triggered. The VMM records faults on sampled pages and computes, at the end of each sampling period and for each sample set, the fraction of sampled pages that was accessed.

A fast and a slow exponentially moving weighted average (EWMA) is kept for each sample set. The first helps establish an estimate that reacts quickly to changes

in activity; the latter remains stable over longer periods of time. Lastly, a special EWMA is computed for the youngest sample set (the set that was updated last). This value helps track intra-period abrupt changes in activity.

The final activity estimate for a VM is computed as the maximum of all averages for all samples sets. This effectively ensures the average will reflect any recent increases in memory consumption, but will slowly respond to a decline in activity, which is the desired behavior: a VM should be awarded memory as soon as its demand increases, but idle memory should be reclaimed slowly after activity lessens.

The period duration defaults to 1 minute, and the sample set size to 100 pages. All parameters can be fine-tuned by the user on a per-VM basis, in cases where the defaults might not lead to an acceptable performance-accuracy compromise.

4.2 Estimating Per-Node Memory Activity

It is precisely the mechanism described in section 4.1 that we extended in order to estimate per-NUMA node memory activity. This first enhancement was straightforward: the kernel is aware of the mappings of each VM physical page number (PPN) to a machine page number (MPN). It also knows which node an MPN belongs to. Therefore, at the end of each period, the sampling code in the VMM makes a system call into the kernel, which counts how many pages happened to be sampled from each node, and how many of those were accessed since being invalidated. The same slow and fast EWMAs as described in the previous section are computed for each node, and the maximum of all values is the final activity estimate.

4.3 Estimating Per-vCPU Memory Activity

As discussed in section 3.2, to maximize locality for wide VMs it is necessary to collect per-vCPU data, since different vCPUs may be homed on different nodes. Similarly, per-vCPU estimates may come in handy when running a VM on vNUMA, with a guest and applications that are not NUMA-optimized. In that case, it is up to the

hypervisor to move memory around to boost performance.

In order to gather per-vCPU statistics, we implemented periodic re-invalidation of the page sample sets. In this way, many accesses to a page (from the same or different vCPUs) can be recorded over one sample period. Each sample set is extended with a hash table, which maps page numbers to arrays containing per-vCPU counts. Upon a fault to a sampled page, the faulting vCPU needs to find the page in one of the sample set hash tables, then increment the count corresponding to its vCPU number (vCPUs are numbered sequentially), as well as mark the page as accessed if this is the very first sampled access, in order to maintain the previous functionality.

At the end of each period, the same statistics as for per-VM and per-node estimates are computed. Note that since this is temporal sampling as well as spatial sampling, the accuracy of the results may be questioned. It is entirely possible that many vCPUs may be accessing the same page, but the same one always ends up touching the page exactly after it is invalidated, leading to only accesses from that vCPU being recorded. However, we report in the next chapter that this estimate is, in fact, accurate enough for our purposes.

The default delay between page re-invalidations is 100ms, and is easily configurable through a user-exposed parameter. This allows for a maximum of 600 accesses per page to be recorded over a sample period, which is a relatively high resolution, suggesting a longer delay value (and therefore a lower performance impact) might be more suitable.

4.3.1 Estimation of Sharing Between vCPUs

Incidentally, the availability of per-vCPU access data for each sampled page allows us to get direct estimates of inter-vCPU sharing of pages. Two different parameters are important here: which vCPUs share data, and in what proportion. For example, simply detecting that two vCPUs accessed the same sampled page can be treated differently if both vCPUs touch it equally often, as opposed to 90% touches from one vCPU and only 10% from the other. It could also be the case that one produces data, and threads running on other vCPUs consume it.

As these cases require different optimizations, it is important to separate the types of accesses (reads vs writes), and the relative affinity of a page to a vCPU or set of vCPUs. For demonstration purposes, however, we show in the next chapter a unified metric that incorporate both number of shared pages and relative frequency of access into a single value. Pairs of vCPUs are assigned a "sharing weight," suggesting that the pairs with highest weights be co-placed to run on the same socket, or even the same core, so they benefit from sharing the last-level cache of the processor.

Chapter 5

Evaluation

In this chapter, we describe the architecture of the evaluation system and the benchmark program we used to characterize the performance of our new estimates. In section 5.3, we describe the setup and results of the experiments we carried out for each of the features we added to ESX Server.

5.1 Hardware

We used a single machine for all experiments: an HP ML350 G6 X5650 server. It is based on the Intel Nehalem microarchitecture, and boasts a two-socket NUMA configuration, with each socket incorporating an Intel Xeon X5650 processor (6 cores, 2.66 GHz, 12MB L3). Each CPU can use Hyper-Threading, allowing each constituent core to appear as two logical cores. The system also features 12GB of RAM in total, 3x2GB in each node. In this way, each of the two processors is directly linked to half of the memory in the system, via a local memory controller. The two nodes are connected using Intel QuickPath Interconnect technology [10].

The test system is setup to run an internal build of VMware ESX Server 5.0.

5.2 Benchmark

Our testing strategy was to implement a benchmark with known memory access patterns, which would run inside a VM in ESX. Observing the estimates produced for per-node and per-vCPU activity would reveal the extent of the sampling error, demonstrating whether our estimates were indeed viable. We also tested inter-vCPU sharing and page access frequency estimation.

For our evaluation, we designed a multi-purpose synthetic benchmark to test the features we introduced in the hypervisor. The benchmark, meant to run as a user-level process in a VM, was a memory *toucher* application. It starts by allocating a block of memory, M, and iterating over it once in order to make sure the VM actually allocates pages (on Linux, allocation happens on first touch, rather than immediately after a call such as `malloc`). Then the *toucher* can be configured to iterate repeatedly over a constant-sized chunk of memory, or chunks of variable sizes using a step size s to simulate workloads with dynamic working set sizes. The iterations can be sequential or random, to intensify or eliminate cache effects due to spatial locality. If a step size is defined, a step duration, i, should also be provided. The workload iterates through increasing and decreasing step sizes for i seconds per step, reaching its peak memory consumption N times before terminating.

The benchmark also includes the possibility of running multiple instances side by side, which may optionally share a single block of memory using a shared key, k, and Linux shared memory segments (*shmget*).

5.3 Experiments Run

All experiments involved a single 4-vCPU VM with 4GB of memory running on our test machine. The VM is configured with two virtual NUMA nodes and two vCPUs on each node, with the `numa.vcpu.maxPerVirtualNode` option set to "2." The system is further configured to run a maximum of 2 vCPUs per physical node, in an attempt to make sure the kernel does not migrate vCPUs to run on the same

Table 5.1: Per-Node Working Set Estimation Experimental Setup

	vCPU0	vCPU1	vCPU2	vCPU3
Active	Yes	No	No	No
vCPU pinned	Node 0	Node 0	Node 1	Node 1
Total allocation (M)	1536MB	-	-	-
Memory location	Node 0	-	-	-
Memory shared	No	-	-	-
Step size	768MB	-	-	-
Step duration	8 min	-	-	-
Iterations	3	-	-	-

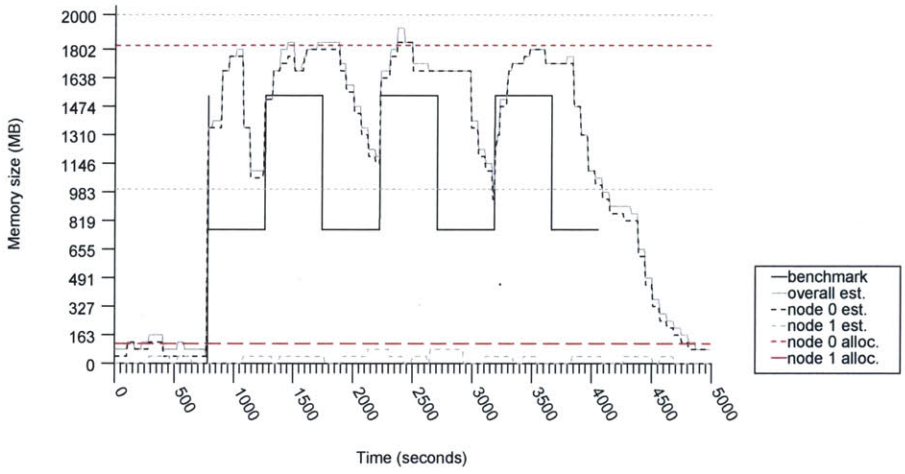


Figure 5-1: Results of per-NUMA node active memory estimation testing.

node, although they belong to different virtual nodes. This is achieved by setting the `numa.vcpu.maxPerMachineNode` option to "2." The VM runs Ubuntu 12.04 "Precise Pangolin" as the guest operating system.

5.3.1 Per-Node Working Set Estimation

To evaluate per-node active memory estimation, we setup an instance of the toucher benchmark as shown in table 5.1. A single instance was launched, constrained to run on vCPU0 and allocate memory only on vNUMA node 0 using the `numactl` command-line utility with options `--physcpubind=0` and `--membind=0`.

The results of the experimental run can be seen in Figure 5-1. As can be seen, node 1 is estimated to have nearly 0 active pages for most of the test period, while the

Table 5.2: Per-vCPU Estimation Experimental Setup

	vCPU0	vCPU1	vCPU2	vCPU3
Active	Yes	No	Yes	No
vCPU pinned	Node 0	Node 0	Node 1	Node 1
Total allocation (M)	1536MB	-	1536MB	-
Memory location	Node 1	-	Node 0	-
Memory shared	No	-	No	-
Step size	768MB	-	768MB	-
Step duration	8 min	-	8min	-
Iterations	3	-	3	-

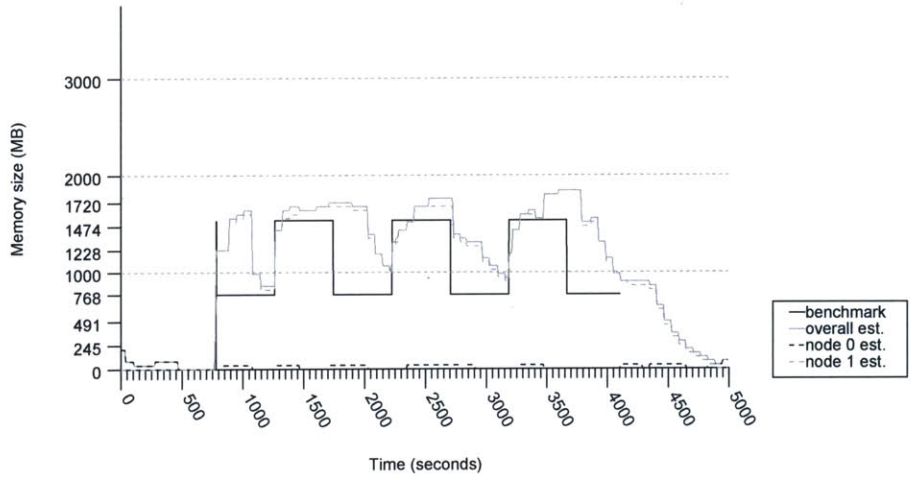


Figure 5-2: Results of per-vCPU, per-node active memory estimation testing. Plotted above are the overall and per-node activity estimates for vCPU0.

estimate for node 0 closely follows the overall VM's activity estimate, as is expected since the toucher is the only workload running in the VM.

5.3.2 Per-vCPU, Per-Node Working Set Estimation

For per-vCPU, per-node activity estimation, we set up our workload so that two vCPUs are running two touchers in parallel. Both are configured to alternate between iterating over 768MB and 1.5GB of memory, with vCPU0 touching memory pinned on node 1 and vCPU2 touching memory pinned on node 0, as displayed in table 5.2. vCPUs 1 and 2 are idle. Figures 5-2 and 5-3 show the outcome of the experiment. (Plots for idle vCPUs are omitted for the sake of brevity. The estimates for those vCPUs gravitate around 0.)

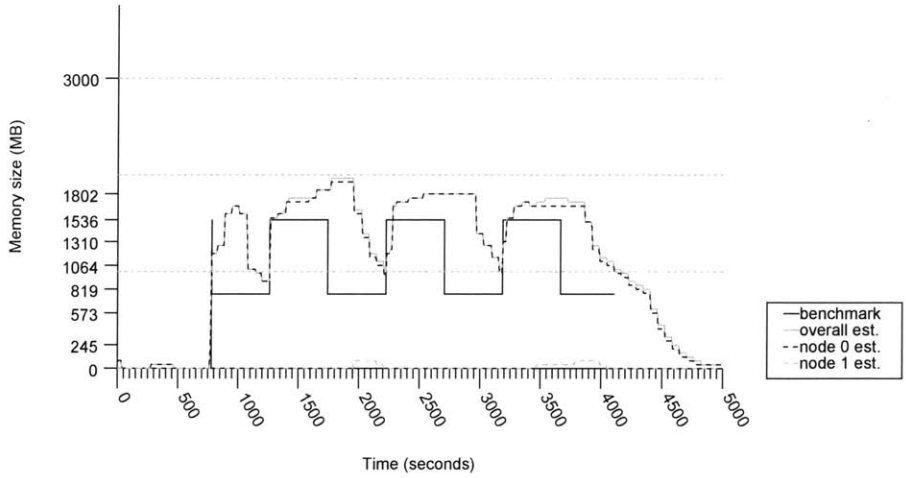


Figure 5-3: Results of per-vCPU, per-node active memory estimation testing. Plotted above are the overall and per-node activity estimates for vCPU2.

Table 5.3: Inter-vCPU Sharing Detection Experimental Setup

	vCPU0	vCPU1	vCPU2	vCPU3
Active	Yes	No	Yes	No
vCPU pinned	Node 0	Node 0	Node 1	Node 1
Total allocation (M)	1536MB	-	1536MB	-
Memory location	Node 0	-	Node 0	-
Memory shared	vCPU2	-	vCPU0	-
Step size	768MB	-	768MB	-
Step duration	8 min	-	8min	-
Iterations	3	-	3	-

For both active vCPUs, the overall activity estimate tracks very closely the running workload. Similarly, the per-node estimates for nodes that have no active memory remain close to zero, whereas active nodes match overall estimates in activity.

5.3.3 Inter-vCPU Sharing Detection

To evaluate inter-vCPU sharing detection, we configured two toucher processes to run as shown in Table 5.3. They are executed by vCPU0 and vCPU2, and both share a single memory chunk allocated by vCPU0 on node 0. Therefore, we expect to detect heavy sharing between vCPUs 0 and 2.

In order to demonstrate the effectiveness of our sharing detection mechanism, we designed a metric for sharing between each vCPU pair, which encompasses both

Algorithm 5.1: Computing a vCPU pair's sharing weight. Function A(bpn, $vCPU_i$) returns the number of times we recorded a touch on sample page bpn by vCPU $vCPU_i$.

input : lists of sampled page numbers (BPNs)
output: sharing weight of $vCPU_i$ and $vCPU_j$

```
1  overlap ⟵ 0;
2  pages ⟵ 0;
3  result ⟵ 0;
4  foreach bpn in BPNs do
5      if A(bpn, vCPU_i) > 0 or A(bpn, vCPU_j) > 0 then
6          overlap ⟵ overlap + [2 · A(bpn, vCPU_i) · A(bpn, vCPU_j)]/
7                              [A(bpn, vCPU_i) + A(bpn, vCPU_j)];
8          pages ⟵ pages + 1;
9  if pages > 0 then
10     result ⟵ overlap/pages;
11 return result
```

Table 5.4: Access Frequency Estimation Experimental Setup

	vCPU0	vCPU1	vCPU2	vCPU3
Active	Yes	Yes	Yes	Yes
vCPU pinned	Node 0	Node 0	Node 1	Node 1
Total allocation (M)	1536MB	1536MB	1536MB	1536MB
Memory location	Node 0	Node 1	Node 1	Node 0
Memory shared	vCPU3	vCPU2	vCPU1	vCPU0
Step size	768MB	768MB	768MB	768MB
Step duration	8 min	8min	8min	8min
Iterations	3	3	3	3

the number of shared pages and the access frequency to those pages by each vCPU. Our metric makes use of the harmonic mean, and is computed as described in Algorithm 5.1.

We plot our metric against time in Figure 5-4. The sharing weight of pair vCPU0-vCPU2 dominates all other pairs for the period of the benchmark run, thus successfully exposing inter-vCPU sharing with a direct estimate, without any guest participation.

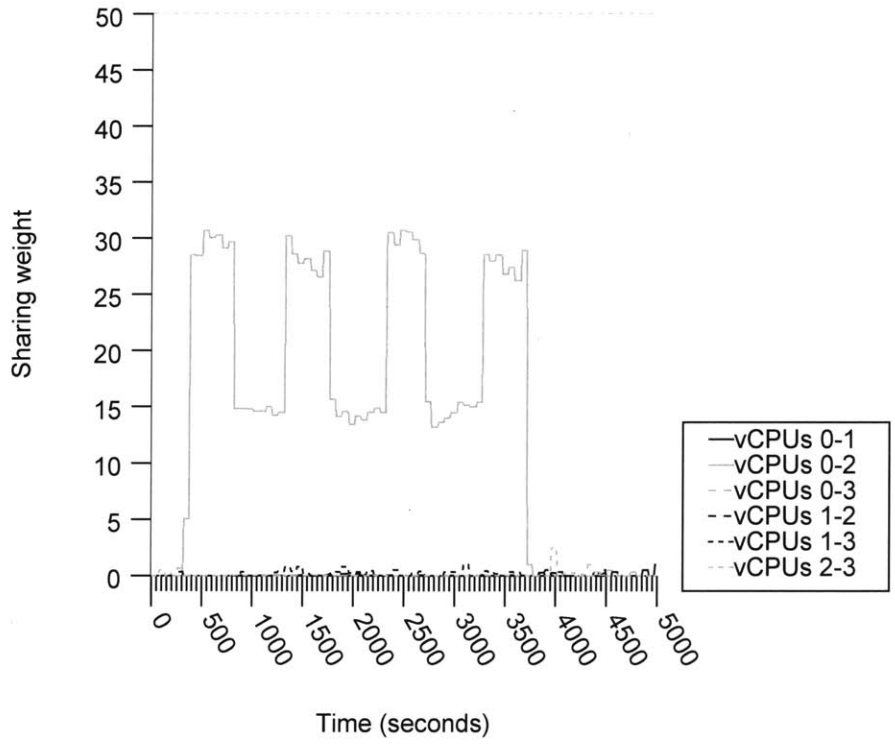


Figure 5-4: Results of inter-vCPU sharing estimation testing. The plot shows the sharing weight of the six possible vCPU pairs in a 4-vCPU configuration. vCPU0 and vCPU2 carry a notable higher weight, as they are executing a toucher workload iterating over a single shared chunk of memory.

5.3.4 Access Frequency Estimation

We configured our access frequency experiments as shown in Table 5.4. Four *toucher* workloads are launched, each pinned to a particular vCPU. The workloads running on vCPU0 and vCPU1 share a block of memory, as do the workloads running on vCPU2 and vCPU3. The first pair of vCPUs iterate over their memory as fast as possible, whereas the second pair execute 10 consecutive `asm("pause")` instructions after each access to 4 bytes of memory. This effectively lowers the access rate of the second pair significantly, relative to the first one. It is this relative difference that we look to observe in the output data.

The resulting access pattern, at the VM level, is plotted in Figure 5-5. Access rate plots are shown in Figure 5-6. Plot 5-6a highlights the difference in access frequency between node 0 and node 1 at the VM level. Plots 5-6b and 5-6c show the number of

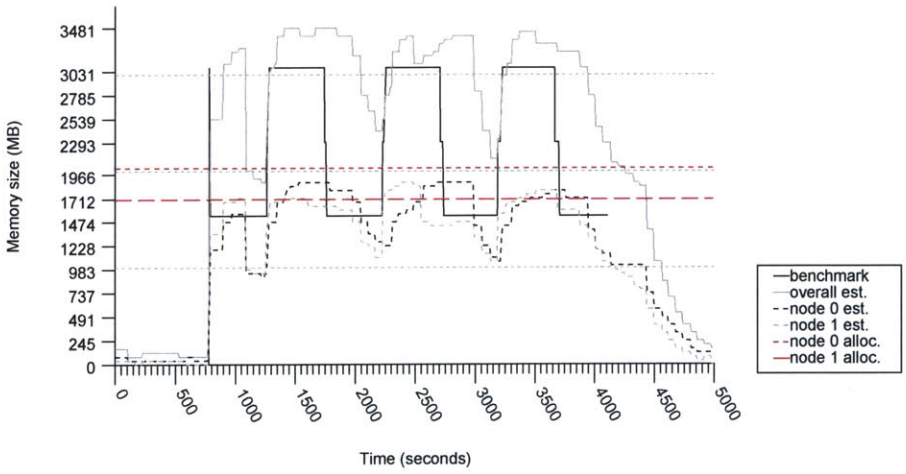
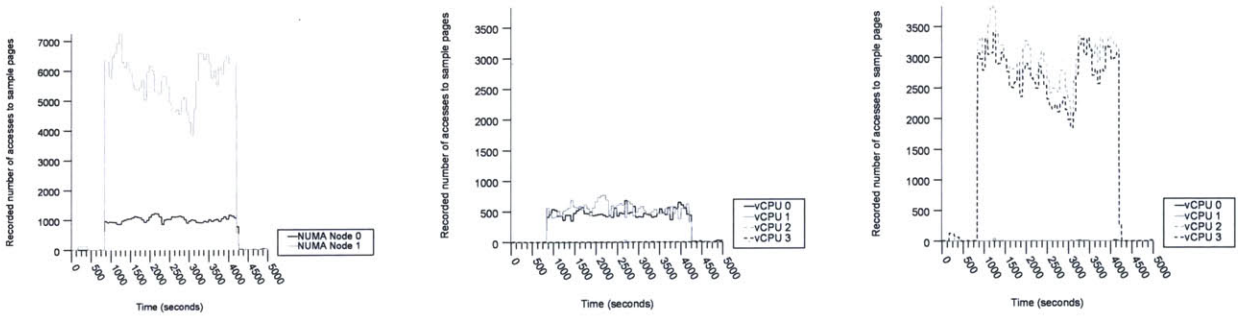


Figure 5-5: Access frequency detection testing: active page count estimates.



(a) Per-node access counts (b) Node 0 access counts per vCPU (c) Node 1 access counts per vCPU

Figure 5-6: Results of access frequency detection testing.

accesses recorded as originating from each vCPU to node 0 and node 1, respectively.

5.4 Discussion

The results for per-node active working set estimation show strong promise. Our estimates shoot up quickly with each activity increase, and slowly ramp down after activity has decreased or ceased. By design, the estimate overshoots, reaching a peak of 17% above the actual workload. However, our assumptions for the workload's consumption do not account for overhead memory such as, for example, pages used for the text segment. Therefore, we have reason to believe the estimates are in fact closer to the real working set size than the plots presented above would suggest.

Performance-wise, per-node estimation only adds some additional logic to the end-of-period sampling computations. No additional page faults are incurred, in contrast with per-vCPU sampling. As such, the added overhead for per-node estimation is negligible.

In section 5.3.2, we demonstrate we can successfully estimate the number of active pages on a per-vCPU, per-node basis. As discussed previously, this data will be highly valuable when evaluating wide VMs and VMs running over more than one NUMA node with vNUMA. It is worth noting that in Figure 5-2, the estimate does not soar significantly higher than the assumed workload consumption, which is not the case in Figure 5-3. We believe this is due to the use of temporal sampling; vCPU2 happened to access shared pages right after invalidation more often than vCPU0 did. This is not surprising, as vCPU2 is expected to have a higher access frequency to the shared block than vCPU0, simply because the memory is local to vCPU2's home node.

The performance impact of the periodic re-invalidation of page mappings for per-vCPU activity estimation was not studied in detail, as our implementation was destined as a proof-of-concept. Our initial results showed an unexpected slowdown of up to 10%, suggesting performance profiling of the implementation should be carried out to weed out any obvious flaws. A few hundred page invalidations every 100ms should not lead to a noticeable decrease in performance, though in the current implementation, all vCPUs are halted briefly for this step and additional system calls are required. On top of the additional invalidation operations, performance may also suffer a hit from the extra page faults induced. This can be mitigated by choosing a longer timeout between invalidations.

Our results in section 5.3.3 demonstrate we can successfully pick out a pair of vCPUs that share memory out of all possible vCPU pairs. Figure 5-4 shows our custom "sharing weight," which combines the number of shared pages and the average access rate to a page. It is of interest to note that our weight value is inversely proportional to the number of shared pages, and as such, the periods during which the two benchmark instances iterate over a smaller memory size, but cycle through it more frequently, have noticeably higher sharing weight. In other words, our metric

was designed to reward small working sets that are heavily shared. To illustrate why this is useful, consider the following scenario: vCPU0 shares memory with vCPU1 and vCPU2, and pages are shared in the same proportion on average. If the amount of memory shared between vCPU0 and vCPU1 is twice the amount shared between vCPU0 and vCPU2, we would prefer to co-place the first pair on the same socket rather than the second, since it is more likely that the pair's entire working set will fit in local memory.

As mentioned in Chapter 4, a unified sharing metric might not be ideal for fine-grain decision making on how to place memory. In a producer-consumer situation, the best course of action would be to co-place the collaborating vCPUs. In a read-only sharing scenario, it might be optimal to duplicate memory across nodes instead. However, as a first step, when a vCPU (or set of vCPUs) is considered for migration, the sharing weight can be a good indication of which node would be its optimal target, in case sharing has been detected.

Our final result in this chapter is access frequency estimation. Since we sample both spatially and temporally (i.e. we only record the first access to a sampled page every 100ms), our estimates are not very useful in absolute terms, as it is unclear how many accesses we are missing in-between page invalidations. However, they can be very valuable in relative terms. This can be seen by evaluating Figures 5-5 and 5-6a. In the former, we notice that the estimated number of active pages is close for both nodes over the duration of the workload. This is correct, because we set up our experiment so that each node has approximately 1.5GB of active memory. However, the latter figure shows us that the access rates to each node are indeed very different, with NUMA node 1 being accessed approximately 6 times more often, on average, than node 0. This reflects the other parameter used in our experiment, which imposed a lower access rate to memory on node 0. Taking into account our access rate estimates, we therefore realize that even our improved per-node memory activity estimates can be misleading, and that the access frequency to memory should be a very important factor in locality optimization decisions. In this particular case, if this spread of data and access patterns had occurred in a non-wide VM, it would

make sense to preferentially set the VM's home node to node 1. Reaching further in this analysis, it is possible to picture a scenario in which a node has more active pages than another, but the other is still preferred due to a higher average access frequency.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

Conclusions and Future Work

6.1 Summary of Results

The most important result of this thesis is the evidence of the applicability of statistical sampling of memory to various aspect of NUMA locality estimation. We have shown that this technique can deliver results where performance counters on modern processors, or other hardware-assisted approaches, would fall short. It is clear that dynamic reorganization of memory is necessary to achieve optimal performance on NUMA systems. In the case of a type 1 hypervisor such as VMware ESX, we need accurate data on memory access patterns per node and per vCPU process in order to make satisfactory migration decisions.

Our investigation in Chapter 3 concluded that, while other approaches might be more desirable, they are currently not realizable. Performance counters on both Intel and AMD do not provide us with the full information we require, and neither do A/D bits. Therefore, sampling memory and tracing page faults was deemed a reasonable alternative. We presented an overview of an implementation of sampling for locality estimation in Chapter 4.

Test data from our proof-of-concept implementation was presented in Chapter 5. Using a synthetic benchmark and the `numactl` Linux utility, we verified that our estimates were valid. Interesting results included the direct detection of inter-vCPU sharing of data, as well as observations of relative access frequency to pages by dif-

ferent vCPUs, and to different NUMA nodes.

6.2 Future Work

While the work in this thesis provided promising results, they are only a hint towards a long series of future works that can have a significant impact on memory performance on NUMA systems. As a starting point, the existing algorithms for dynamic memory and VM migrations should be modified to use active estimates instead of static allocation data. Besides such relatively straightforward changes, however, there is also entirely new functionality that can be developed. In this section, we first present an idea that could potentially lead to performance improvements in a specific corner case, which is not currently handled by the hypervisor. We then discuss the idea of characterizing workloads according to the memory usage and sharing patterns they exhibit.

6.2.1 Idle Memory Eviction

The corner case we suggest as worthy of investigation is one where a VM with idle pages is hogging its home node's available local memory, potentially forcing other VMs to allocate remotely. Without system-wide memory pressure, the kernel does not feel the need to reclaim memory from the VM, either by ballooning it or by other means, such as compressing it or swapping to disk. And without per-node data on active memory, it is not aware that the node is overcommitted with idle memory.

For instance, consider the following scenario. VMs $vm1$, $vm2$, and $vm3$ are started, in that order. By round-robin assignment of home nodes, on a two-node machine, $vm1$ and $vm3$ end up on the same node. If $vm2$ remains idle, and both $vm1$ and $vm3$ run a CPU-intensive workload, we expect one of them (say $vm3$) to be migrated for load balancing reasons and start sharing a home with $vm2$. If, however, $vm2$ ran a memory-hungry workload before this migration and allocated most of the memory on its home node, $vm3$ will not be able to migrate its memory along to its new destination. As such, it will end up accessing mostly remote memory from its

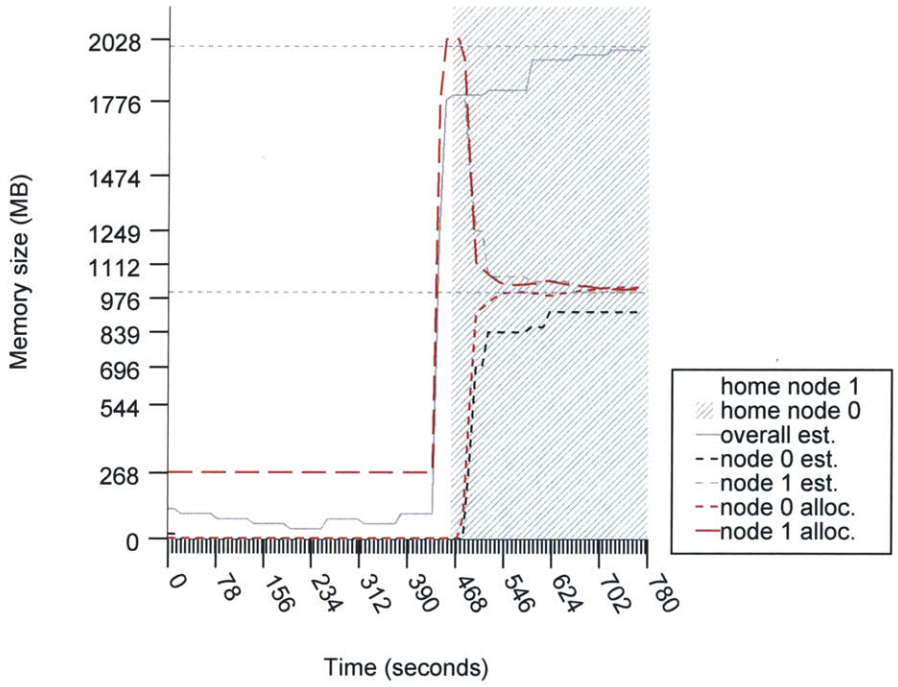


Figure 6-1: Idle memory eviction experiment. In this test run, one of the VMs in the system ($vm3$) migrates from its original home node, node 1, to node 0. Because the latter has most of its memory allocated to $vm2$, $vm3$ manages to relocate only half of its memory, ending up with 50% remote memory accesses.

previous home node, hampering performance.

A test run of this scenario is shown on Figure 6-1. We used the same test machine as described in Chapter 5. The experiment was set up with 5GB used by $vm2$ on node 1 (out of 6GB available), and 2GB used by each $vm1$ and $vm3$. The figure shows the usage and migration patterns of the latter, which is running a workload repeatedly iterating over a constant-sized block of data. As expected, a migration occurs and memory is migrated, but half of the VM's memory is left behind as it does not fit on the destination node. Ideally, in a situation like this, idle memory would be either reclaimed or migrated away to "make room" for $vm3$'s active memory. Migration is the more appealing solution, as reclaiming memory from $vm2$ might lead to decreased performance down the road, when $vm2$ ramps up its activity.

With the techniques from this thesis, we can detect the local memory pressure and take action. An algorithm could consider each NUMA client in turn, find ones exhibiting locality below a set threshold, and then look for idle clients residing on the

same home node that could be migrated away to free up valuable local memory. The freed memory would be used by the pressured clients to bring in pages in order to decrease the number of remote accesses.

Armed with per-node locality data, the algorithm can compute a prediction of what the long-term impact of the migration will be. We present below a set of equations to compute such a prediction for the overall performance delta in terms of CPU cycles gained. We first define a series of convenience functions, all pertaining to a single VM except $L(m, n)$, which is a property of the system:

$$
\begin{aligned}
O(n) &= \text{occupancy on node } n, \\
O_t &= \text{total occupancy across all nodes} \\
A(n) &= \text{number of active pages on node } n, \\
R(n) &= \text{last level cache misses per page per second, for pages on node } n^1, \\
M(m,n) &= \text{pages migrated from node } m \text{ to node } n, \\
L(m,n) &= \text{latency when accessing node } n \text{ from node } m, \\
D &= \text{average duration between VM migrations.}
\end{aligned}
$$

Let $vm1$ be the VM that has low locality, and $vm2$ the VM that is considered as a candidate to be migrated from node *src* to node *dst*, where *src* is also $vm1$'s home node. In order to compute the delta for $vm1$, we first need to estimate the number of pages that will be freed on its home node. We compute that number by assuming that once $vm2$ migrates, it will either fill up all available memory on node *dst* and still have remote memory, or it will migrate all of its remote memory and there will still be free space left on *dst*. In case not all remote memory can be migrated, we assume pages are chosen uniformly at random and migrated if they are remote, so the number of pages migrated from the source node is directly proportional to the fraction of $vm2$'s memory located on node *src*.

[1] If the per-node cache miss rate is not known, the average cache miss rate can be used as a proxy.

$$freed = \begin{cases} O_{vm_2}(src) & O_{t,vm_2} - O_{vm_2}(dst) < Free(dst) \\ Free(dst) \cdot \frac{O_{vm_2}(src)}{O_{t,vm_2} - O_{vm_2}(dst)} & \text{otherwise} \end{cases}$$

where $Free(n)$ is the number of unallocated pages on node n.

Next, we need to estimate the number of pages of $vm1$ that were previously remote and will be migrated to its home node (assuming $vm1$ takes up the freed memory in its entirety).

$$M(*,\ src) = min(freed,\ O_t - O(src))$$

Using the previous assumptions that pages are chosen for migration uniformly at random across all remote nodes, the number of pages migrated from node n is

$$M(n,\ src) = M(*,\ src) \cdot \frac{O(n)}{O_t - O(src)}.$$

The number of active pages migrated from each node n can then be estimated as

$$M_{active}(n,\ src) = M(n,\ src) \cdot \frac{A(n)}{O(n)}.$$

At last, we can estimate the number of CPU cycles $vm1$ gains from this migration:

$$\mathbb{E}[cycles] = D \cdot \sum_{n\ \in\ nodes} M_{active}(n,\ src) \cdot R(n) \cdot [L(src,\ n) - L(src,\ src)].$$

Similarly to $vm1$, we can estimate the change in performance for $vm2$. Because of the VM migration, we need to consider not only any migrated memory, but also the modified distance (and thus latency) to all nodes in the system:

$$\mathbb{E}[cycles] = D \cdot \sum_{n\ \in\ nodes} R(n) \cdot [A_{before}(n) \cdot L(src,\ n) - A_{after}(n) \cdot L(dst,\ n)].$$

An algorithm can use these estimates for a number of migration candidate VMs, and for a number of destination for each VM. The best delta will be achieved by migrating a VM with large amounts of idle memory on the source node, especially if it will not suffer a significant performance impact from the migration (e.g. if the VM itself is more generally idle). The algorithm can run independently of any performance-critical code paths, and therefore checking all VMs repeatedly will not hurt overall system performance. If a VM is found to have good locality and is not a good candidate for a memory eviction migration, it can be flagged as such and not be considered again until a later time (a few minutes would be reasonable as VM migrations should not be frequent events) to further reduce the impact of this algorithm.

In a broader context, the equations presented provide a framework for any algorithm dealing with VM migrations to estimate the outcome of a single migration, or two or more VMs swapping home nodes. Further research is necessary to determine whether this case, and other similar cases, are encountered frequently enough and provide sufficient performance gain to warrant inclusion into a release version of a commercial hypervisor, such as VMware ESX.

6.2.2 Workload Sharing Patterns Analysis

Another application of the estimation techniques described in this thesis would be the analysis and categorization of various workloads of interest with respect to their NUMA performance and data usage patterns. Recent research [23] has highlighted the importance of matching access patterns and data placement for applications running on NUMA systems. The inverse is also valuable – investigating why third party applications perform the way they do, and what the operating system and the administrator can do to maximize performance.

In the case of VMware ESX, one area in particular in which virtualization is increasingly relevant is High Performance Computing (HPC) [24]. Performance evaluations of existing HPC benchmarks on ESX show some run at close to or faster than native speed, whereas others suffer a higher performance cost when virtualized [1].

We suggest directly inspecting these workloads' data layouts, access rates, vCPU utilization, and possible inter-thread data sharing, in order to characterize where any major bottlenecks may lie.

With deeper insight into these bottlenecks, the hypervisor's CPU scheduling, memory allocation and reclamation, and any NUMA migration algorithms can be tweaked to perform better under specific workloads, which may possibly lead to general improvements in performance. VMware and other vendors can also provide further recommendations for workload-specific tweaks, helping customers make the most out of their system.

6.3 Conclusion

The research in this thesis has been fruitful. We have demonstrated successful NUMA locality estimation techniques based on memory sampling that are likely to bear fruit in the future, although they could possibly be phased out eventually due to increased hardware support. As discussed in the previous section, a lot remains to be learned on where these techniques might become most relevant. We look forward to the future and hope that this research will help ease the transition to systems with increasingly complex memory architectures, highlighting the importance of the role of software and the operating system in the management of memory.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Q. Ali, V. Kiriansky, J. Simons, and P. Zaroo. Performance evaluation of hpc benchmarks on vmwares esxi server. *5th Workshop on System-level Virtualization for High Performance Computing (HPCVirt 2011)*, Aug 2011.

[2] Joseph Antony, Pete P. Janes, and Alistair P. Rendell. Exploring thread and memory placement on numa architectures: solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *Proceedings of the 13th international conference on High Performance Computing*, HiPC'06, pages 338–352, Berlin, Heidelberg, 2006. Springer-Verlag.

[3] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[4] J. Bardeen and W. H. Brattain. The transistor, a semi-conductor triode [14]. *Physical Review*, 74(2):230–231, 1948. Cited By (since 1996): 155.

[5] N. Bhatia. Performance evaluation of intel ept hardware assist. Retrieved from `http://www.vmware.com/`.

[6] S. Blagodurov and A. Fedorova. User-level scheduling on numa multicore systems under linux. *Linux Symposium*, 2011. Cited By (since 2011): 7.

[7] Hewlett-Packard Company. Linux numa support for hp proliant [white paper]. Mar 2012. Retrieved from `http://bizsupport1.austin.hp.com/`.

[8] Microsoft Corp. Numa support (windows). [Online] Available: `http://msdn.microsoft.com/en-us/library/windows/desktop/aa363804(v=vs.85).aspx`.

[9] Microsoft Corp. Server virtualization, windows server 2012 [white paper]. 2012. Retrieved from `http://download.microsoft.com/`.

[10] Intel Corporation. An introduction to the intel quickpath interconnect [white paper]. Jan 2009. Retrieved from `http://www.intel.com`.

[11] Intel Corporation. Intel 64 and ia-32 architectures software developers manual, volume 3: System programming guide. Mar 2013.

[12] M. Dobson, P. Gaughen, and M. Hohnbaum. Linux support for numa hardware. In *Proceedings of the Linux Symposium*, July 2003.

[13] S. Eranian. What can performance counters do for memory subsystem analysis? *ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC08)*, 2008.

[14] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[15] M.D. Hill and M.R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[16] Advanced Micro Devices Inc. Amd-v nested paging [white paper]. Jul 2008. Retrieved from `http://developer.amd.com/`.

[17] Advanced Micro Devices Inc. Basic performance measurements for amd athlon 64, amd opteron and amd phenom processors. Sep 2008. Retrieved from `http://developer.amd.com/`.

[18] Advanced Micro Devices Inc. Amd64 architecture programmers manual volume 2: System programming. Sep 2012. Retrieved from `http://support.amd.com/`.

[19] A. Kleen. A numa api for linux [white paper]. Apr 2005. Retrieved from `http://developer.amd.com/`.

[20] C. Lameter. Local and remote memory: Memory in a linux/numa system. *Linux Symposium*, Jun 2006. Cited By (since 2006): 12.

[21] D. Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. 2009.

[22] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. Lohman. Numa-aware algorithms: the case of data shufing. *6th Biennial Conference on Innovative Data Systems Research (CIDR13)*, Jan 2013.

[23] Z. Majo and T. R. Gross. Matching memory access patterns and data placement for numa systems. In *Proceedings - International Symposium on Code Generation and Optimization, CGO 2012*, pages 230–241, 2012.

[24] M.F. Mergen, V. Uhlig, O. Krieger, and J. Xenidis. Virtualization for high-performance computing. *SIGOPS Oper. Syst. Rev*, 2006.

[25] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. Case for a single-chip multiprocessor. *Computer architecture news*, 24(Special Issu):2–11, 1996. Cited By (since 1996): 105.

[26] R.M. Ramanathan. Intel multi-core processors, making the move to quad-core and beyond [white paper]. Sep 2006. Retrieved from `http://www.cse.ohio-state.edu/`.

[27] Robert R. Schaller. Moore's law: past, present, and future. *IEEE Spectrum*, 34(6):52–55, 57, 1997. Cited By (since 1996): 138.

[28] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, 1997.

[29] Carl A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.