Optimizing Virtual Machine Scheduling in NUMA Multicore Systems

Jia Rao[1], Kun Wang[2], Xiaobo Zhou[1], Cheng-Zhong Xu[2]
[1]Dept. of Computer Science
University of Colorado, Colorado Springs
{jrao, xzhou}@uccs.edu
[2]Dept. of Electrical and Computer Engineering
Wayne State University
{kwang, czxu}@wayne.edu

Abstract

An increasing number of new multicore systems use the Non-Uniform Memory Access architecture due to its scalable memory performance. However, the complex interplay among data locality, contention on shared on-chip memory resources, and cross-node data sharing overhead, makes the delivery of an optimal and predictable program performance difficult. Virtualization further complicates the scheduling problem. Due to abstract and inaccurate mappings from virtual hardware to machine hardware, program and system-level optimizations are often not effective within virtual machines.

We find that the penalty to access the "uncore" memory subsystem is an effective metric to predict program performance in NUMA multicore systems. Based on this metric, we add NUMA awareness to the virtual machine scheduling. We propose a Bias Random vCPU Migration (BRM) algorithm that dynamically migrates vCPUs to minimize the system-wide uncore penalty. We have implemented the scheme in the Xen virtual machine monitor. Experiment results on a two-way Intel NUMA multicore system with various workloads show that BRM is able to improve application performance by up to 31.7% compared with the default Xen credit scheduler. Moreover, BRM achieves predictable performance with, on average, no more than 2% runtime variations.

1. Introduction

Multicore systems have become the fundamental platforms for many real-world systems, including scientific computing clusters, modern data centers, and cloud computing infrastructures. While enjoying the advantage of simultaneous thread execution, programmers have to deal with the problems multicore systems give rise to. Sub-optimal and unpredictable program performance due to shared on-chip resources remains top concern as it seriously compromises the efficiency, fairness, and Quality-of-Service (QoS) that the platform is capable to provide [41]. There are existing work focusing on hardware techniques [32] and program transformations [28, 39, 40] to mitigate the problem. Thread scheduling, a more flexible approach, has been also studied to avoid the destructive use of shared resources [7, 8, 11, 14, 30] or to use them constructively [5, 35, 38].

Non-Uniform Memory Access (NUMA) design, the emerging architecture in new multicore processors, further complicates the problem by adding one more factor to be considered:

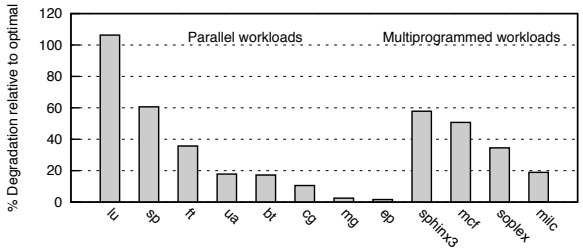

Figure 1: Worst case runtime degradations of parallel and multiprogrammed workloads in a NUMA system.

memory locality. The complex interplay among these factors makes the determination of the optimal thread-to-core assignment difficult. We ran the NAS parallel benchmarks and the SPEC CPU2006 benchmarks on an Intel NUMA machine (see Section 2.2 for the hardware configuration) with different thread-to-core mappings. Each parallel benchmark was configured with four threads and each multiprogrammed workload ran four identical copies of the corresponding CPU2006 benchmark. The memory of the benchmarks was allocated on node 0 of the two memory nodes.

Figure 1 shows the worst-case performance of these workloads relative to their best-case performance (termed as optimal) that is obtained by enumerating CPU bindings. We make two observations. First, not all the benchmarks are sensitive to the scheduling on NUMA systems (e.g., `ep` and `mg`), but when they are, the degradation can be significant (e.g., a 106% degradation for `lu`). Second, although differ in nature, both parallel and multiprogrammed workloads may suffer performance degradations on NUMA systems. Previous work used last-level cache miss rate [7, 8, 14] and inter-cache access rate [35] to quantify the contention and sharing on shared resources, respectively. However, no existing work has combined these two factors together to predict the optimal schedule for unknown workloads [42]. Given the factor of memory locality, performance optimization on NUMA multicore systems becomes even harder. This motivated us to develop a unified and automated approach to optimizing the NUMA performance for an arbitrary program.

Virtualization poses additional challenges to performance optimization on NUMA systems. First, programs running inside a virtual machine (VM) have little or inaccurate knowledge about the underlying NUMA topology, preventing program and guest operating system (OS) level optimizations from working effectively. Second, user-level schedulers often

apply strict CPU affinities to the virtual CPUs (vCPUs) [42]. This creates some isolated resource islands on the physical machine, making it difficult to enforce fairness and priorities among VMs [13]. These challenges call for an approach that integrates NUMA awareness into the VMM-level scheduling. As such, optimizations can be made transparent to users and do not affect the in place VM management.

In this work, we propose a NUMA-aware vCPU scheduling scheme based on a novel hardware-based metric for performance optimization in virtualized environments. We conduct a comprehensive measurement of a carefully-designed micro-benchmark and real-world workloads using hardware performance monitoring units (PMU). Measurement results show that the penalty to access the "uncore" memory subsystem is a good runtime index of program performance. The metric is synthesized from a number of hardware events, including last-level cache (LLC) accesses, memory accesses, and inter-socket communications. Based on this observation, we design a Bias Random vCPU Migration (BRM) algorithm that automatically optimizes vCPU-to-core assignment by minimizing system-wide uncore penalty.

We have implemented BRM in Xen and made a few changes to the Linux kernel running inside a VM to support BRM, and performed extensive experiments with our micro-benchmark and real workloads. Experiment results show that, compared with Xen's default credit scheduler, BRM improves program performance by up to 31.7% and performs closely to an offline determined optimal strategy. Moreover, on average, BRM achieves a performance variation of no more than 2% for all workloads, which is a significant reduction from the credit scheduler.

The rest of the paper is organized as follows. Section 2 presents the background of the NUMA architecture, gives motivating examples, and discusses challenges in a virtualized environment. Section 3 introduces the metric of uncore penalty. Section 4 and Section 5 describe the design and implementation of BRM, respectively. Section 6 provides the experiment results. Section 7 discusses related work and Section 8 presents our conclusions and discusses future work.

2. Background and Motivation

In this section, we first describe the NUMA multicore architecture and discuss its implications on program performance. Then, we further show that virtualization introduces difficulties in attaining optimal and predictable program performance in such systems.

2.1. The NUMA Multicore Architecture

As the number of cores per chip[1] increases, traditional multicore systems with off-chip shared memory controller (i.e., Uniform Memory Access) are likely bottlenecked by the memory bandwidth. To alleviate memory bandwidth competition, new processors integrate a memory controller on-chip. In such multicore multiprocessor systems, a processor can access physical memory either via its own memory controller or via other processors' memory controllers connected by the cross-chip interconnect. Accessing local memory gives higher throughput and lower latency compared with accessing remote memory. This Non-Uniform Memory Access (NUMA) architecture effectively removes the central shared memory controller and significantly increases per core memory bandwidth.

[1] We use chip, socket, and processor interchangeably to indicate an integrated circuit die.

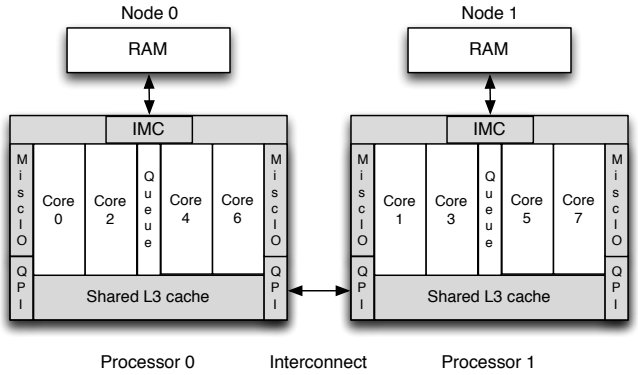

Figure 2: The architecture of a two-socket Intel Nehalem-based NUMA machine.

Figure 2 shows a schematic view of a NUMA multicore system based on dual Intel Nehalem processors. Although other NUMA multicore processors (e.g., AMD Opteron) may differ in the configuration of on-chip caches and the cross-chip interconnect technology, they bear much similarity in the architectural design. Therefore, most our discussions are also applicable to them. As shown in Figure 2, each socket has four cores, which share a last-level cache (L3 cache), an integrated memory controller (IMC), and an Intel® QuickPath Interconnect (QPI). The cores including their private L1 and L2 caches are called the "core" memory subsystem. The shared L3 cache, the IMC, and the QPI together form the "uncore" memory subsystem (the shaded area). Cache line requests from the cores are queued in the Global Queue (GQ) and are serviced by the uncore. Physical memory is divided into two memory nodes, each of which is directly connected to a processor.

2.2. Complications in Achieving Optimal Performance

For a parallel or multiprogrammed workload, its optimal thread-to-core assignment on a NUMA multicore system depends on the workload's memory intensity and access pattern. To avoid contention on shared resources, threads with high memory footprints may be better distributed onto different chips. In contrast, communicating threads should be grouped onto the same chip as cross-chip communication is expensive. Asymmetric memory performance further complicates the scheduling problem and requires that thread computation matches its data as much as possible. When all the three factors come into play, it is the dominant factor that determines the best schedule. In the next, we show that the dominant factor switches as the characteristics of the workload changes.

Inspired by [10], we design a micro-benchmark that exercises different components of a NUMA multicore system. The program can run with a single or multiple threads. Figure 3

```
thread_main(...)
{
    struct payload_t{
        struct payload_t *next;
        long pad[NPAD];
    };
    struct payload_t *ptr_private;
    /* Bind data to thread or to a specific node */
    set_mempolicy(...);
    ptr_private = create_random_list(...);
    for (n = 0; n < LOOP; n++){
        /* access private data */
        for (i = 0; i < WORKING_SET_SIZE; i++){
            for (j = 0; j < NPAD; j++)
                tmp = ptr_private->pad[j];
            ptr_private = ptr_private->next;
        }
        /* access shared data */
        for (i = 0; i < SHARING_SIZE; i++)
            __sync_add_and_fetch(th->ptr_share+i, 1);
    }
}
```

Figure 3: The main function of a thread in the micro-benchmark.

	Intel Xeon E5620
Number of cores	4 cores (2 sockets)
Clock frequency	2.40 GHz
L1 cache	32KB ICache, 32KB DCache
L2 cache	256KB unified
L3 cache	12MB unified, inclusive, shared by 4 cores
IMC	32 GB/s bandwidth, 2 memory nodes, each with 8GB
QPI	5.86 GT/s, 2 links

Table 1: Configuration of the Intel NUMA machine.

shows the main function of a thread. The data accessed by a thread is divided into two parts. One part is the private data that can only be used by one thread. It forms the primary working set of the thread. The private data is allocated inside the thread's own stack and created as a randomly-linked list. We set the payload of each entry in the list to 64 bytes (i.e., `NPAD`=7) matching the size of a cache line in Nehalem. We control each thread's working set size by altering the length of the linked list. As all the entries are randomly linked, hardware prefetching of adjacent cache lines can hardly help. It is the capacity of the LLC and the locality of memory that determine the performance.

The second part is a data area shared by all threads. It is allocated in the main thread and is set to be multiple of the long integer type (i.e., 8 byte, a word in Linux X86_64). We configure the threads to sequentially access the shared space and modify the value of each word by adding 1 to it. This effectively exercises the cache coherence mechanism. To guarantee the correctness under concurrent operations, we use the atomic add instruction `__sync_add_and_fetch` to serialize writes to the same memory location. By controlling `SHARING_SIZE`, we determine the amount of inter-thread traffic generated by the cache coherence protocol. We also specify the memory allocation policy (using `set_mempolicy`) for each thread to study how data locality affects the performance.

We measure the performance of the benchmark in terms of thread runtime. In the case of multiple threads, performance is measured as the average of individual threads' execution times. We ran the benchmark on a Dell PowerEdge T410 machine with two quad-core Intel Xeon E5620 processors. Table 1 lists the details of the hardware. If not otherwise stated, a thread accesses its private and shared data for 1000 times and 4 threads are used for the multithreaded version of the benchmark. Figure 4 shows how individual factors affect program performance. Figure 5 shows that the combination of these factors makes the determination of the optimal schedule difficult. For each test, we draw the performance of two scheduling schemes with the second scheme normalized to the first one.

2.2.1. Studying Individual Factors . To study the factor of data locality, we used a single thread with its data allocated on node 0 (i.e., the memory domain attached to processor 0). In the scenario of *Local*, we set the affinity of the thread to node 0 as well, guaranteeing all local memory access. *Remote* refers to the run that the thread was pinned to node 1 and accessed its memory remotely. In Figure 4(a), we can see that on-chip LLC can hide the penalty to access remote memory when the working set size is smaller than the LLC capacity (i.e., 12MB) and accessing memory remotely will not hurt program performance. However, as the working set size of the thread increases beyond the LLC capacity, data locality plays an important role in performance. The larger the working set size, the larger impact remote penalty hits performance. For example, remotely accessing a working set of 256MB causes a 44% slowdown compared with a 26% slowdown in the case of a 16MB working set.

Figure 4(b) shows the factor of LLC contention on performance. We draw the performance of two thread-to-node assignments. In this test, data sharing among threads was disabled and data was bound to each thread avoiding remote accesses. As indicated by its name, assignment *sharing LLC* places all 4 threads on a single socket sharing the 12MB LLC. Assignment *separate LLC* places 2 threads on each of the two sockets. The *X* axis lists the aggregate working set size of all threads. Similar to the data locality factor study, thread affinity does not affect performance significantly as long as the working sets of the threads can fit in a single LLC. As the aggregate working set size goes beyond the capacity of one LLC, distributing threads across sockets clearly outperforms grouping threads, by as much as 33%. Unlike data locality, whose impact grows as the working set increases, the impact of the LLC contention diminishes as the working set size of a single thread surpasses the LLC capacity.

To isolate the data sharing factor, we set the size of the private area to zero and the size of the shared area to one cache line. Figure 4(c) shows the performance of sharing within a socket and sharing across socket (same number of threads per socket) with different number of threads. As shown in Figure 4(c), scheme *across socket* always incurs higher overhead than *within socket*. Scheme *across socket*'s relative degradation increases initially until reaching its peak at 4 threads and then decreases as more threads are used.

We make three important observations in Figure 4. First, the LLC contention factor and the sharing factor are contradictory.

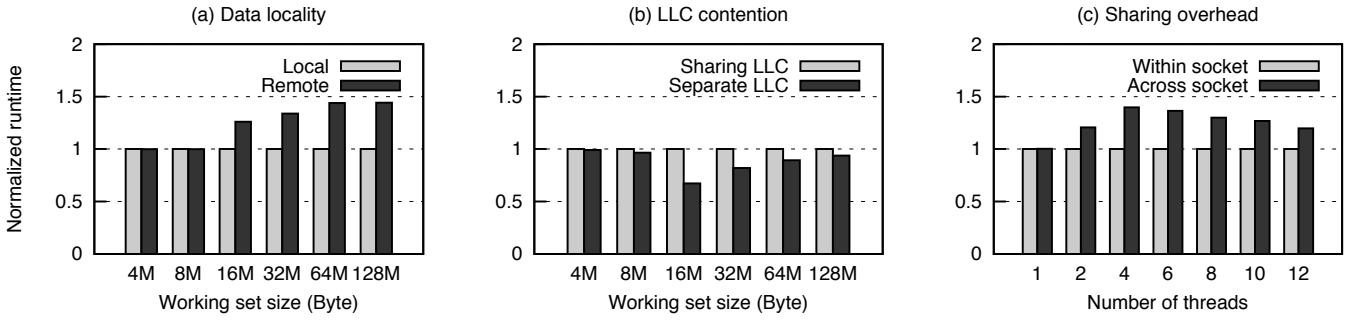

Figure 4: Single factor on program performance.

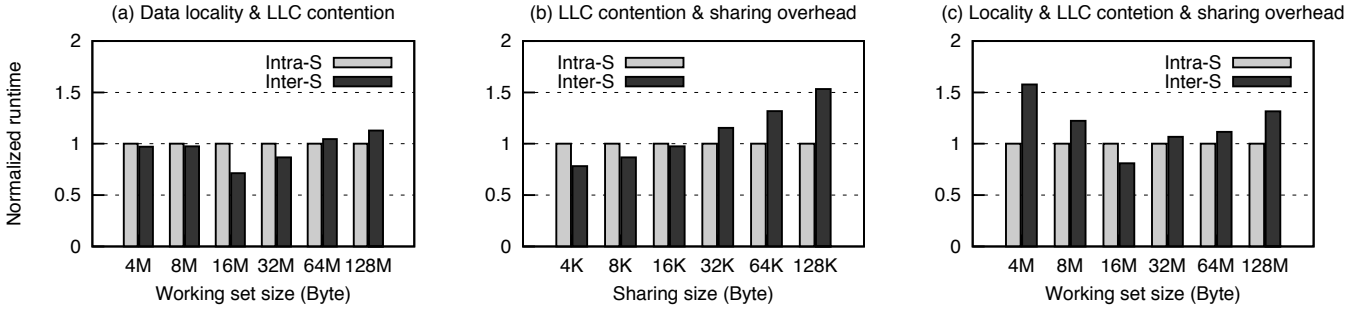

Figure 5: Multiple factors on program performance.

While distributing threads on separate sockets avoids LLC contention, grouping threads onto a single socket reduces sharing overhead. Second, depending on a program's memory allocation, one may need to switch between distributing or grouping threads to favor data locality. Finally, a program's memory footprint, thread-level parallelism, or inter-thread data sharing affect how much each factor influences performance.

2.2.2. Combining Multiple Factors . To study the influence of performance factors in more complex scenarios, we configured the benchmark program to include two or more factors in a single run. Figure 5 presents the performance comparison of two thread assignments in three different settings. The bars shown are the program runtime normalized to assignment *Intra-S*. Assignment *Intra-S* places all 4 threads on a single socket (always node 0) and *Inter-S* distributes threads on separate sockets, with 2 threads per socket. We focus on a simple memory management policy allocating a program's data to a specific node (i.e., node 0). Thus, *Intra-S* always preserves data locality while *Inter-S* has half of the threads accessing memory remotely. Besides data locality, *Intra-S* and *Inter-S* favor sharing and non-sharing the LLC, respectively.

Figure 5(a) shows the interplay between data locality and LLC contention with inter-thread data sharing disabled. Similar to Figure 4(a) and Figure 4(b), Figure 5(a) suggests that thread assignment show insignificant influence on program performance as long as the aggregate thread working set fits in one LLC. As thread working sets increase, *Inter-S* gives better performance as the effect of increased per thread LLC capacity outweighs data locality. Finally, when the working set size of a single thread passes the capacity of the LLC, data locality becomes dominant on program performance.

The second test considers two contradicting factors: LLC contention and data sharing. Since 4 threads with 16MB data benefit most from separate LLCs in Figure 5(a), we fixed the aggregate thread working set size to this value and altered the size of the shared area. The larger the shared area, the more inter-thread communication generated by the cache coherence mechanism, thus the bigger impact the sharing factor places on performance. Figure 5(b) shows that sharing LLC is destructive on performance as the sharing ratio (i.e., the ratio of the size of shared area to the size of private area) remains small. Before reaching a sharing size of 32KB, assignment *Inter-S* is more preferable. After that, the overhead on inter-socket data sharing outweighs the contention on LLC, and sharing LLC (i.e., *Intra-S*) becomes constructive on performance.

Figure 5(c) shows the results of the test run that combined all the three factors. The size of the sharing area is fixed at 128KB. The trend in this figure can be summarized as grouping threads in one socket being constructive initially, then destructive as contention on LLC dominates, and at last, constructive again as locality and sharing overhead together decide performance.

2.3. Obstacles due to Virtualization

To seamlessly share hardware resources among multiple virtual machines, the virtualization layer (usually the VMM) presents each guest OS a virtual abstraction of the machine hardware. A guest OS relies on the virtual to machine translation to access the actual hardware. For memory accesses, the physical addresses used by the guest OS need to be translated to machine addresses. The virtual CPU(s), on which application threads execute, need to be mapped onto physical cores. The mapping from virtual to machine resources has to reflect the NUMA topology for the optimizations in the guest OSes to be effective.

Current VMMs present a flat CPU and memory hierarchy

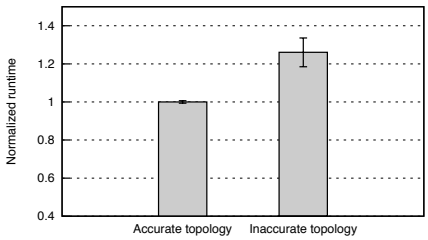

Figure 6: Inaccurate virtual-to-machine topology mapping results in sub-optimal and varying performance.

to VMs and rely on the VMM for NUMA optimizations. A common approach, which has been used by Xen 4.0, VMware ESXi, and Hyper-V, is to allocate all the memory of a VM to one NUMA node and schedule the VM's vCPUs only on this node. As such, the guest OS does not need to be NUMA-aware because all memory accesses are guaranteed to be local. Recent advances [2, 33] in VMM development allow the virtual NUMA topology to be exported to VMs for guest-level NUMA optimizations.

There are several scenarios that the virtual NUMA topology perceived by a guest OS, including the flat and the exported topologies, may become inaccurate. First, VMs with a memory size larger than the available memory in a node, or with a total amount of vCPUs exceeding the number of cores in a node, have to be split across multiple nodes. In such cases, the flat topology does not always preserve data locality. Second, VMs may be migrated to a different host due to inter-machine load balancing. Since not all guest OSes support dynamic changes to the NUMA topology without a reboot, the virtual NUMA topology of the migrated VM may not reflect its actual memory layout on the new host.

Figure 6 shows the performance of the micro-benchmark in a virtualized environment. We used the 4-thread benchmark program with a 128MB working set and a 128KB shared space in Xen VMs. We first ran the benchmark in a VM with 4GB memory and 8 vCPUs. Note that our testbed has 8GB memory and 8 cores on each node. This VM can fit in one node and the flat topology it observes is *accurate*. To create an *inaccurate* NUMA topology, we used a different "wide" VM that has 12GB memory and 8 vCPUs. Thus, the flat topology does not reflect the split memory used by this VM. We ran the benchmark 10 times for each setting with the Xen default scheduler and draw the average runtime (normalized to the first setting) and the relative standard deviation. Figure 6 shows that the inaccurate NUMA topology incurred a performance degradation of 26% and a variation of 8% compared with the runs with accurate topology.

[**Short summary**] In this section, we have shown that the NUMA architecture poses significant challenges in determining the optimal performance for parallel or multiprogrammed workload. Existing VMMs assume a simple and static NUMA topology for VMs. When this assumption does not hold, the NUMA-agnostic VMM-level scheduler is incompetent to provide optimized and predictable performance.

In the following sections, we identify a hardware-level metric that accurately reflects the high-level performance and use the metric to help VM scheduling.

3. Uncore Penalty as a Performance Index

In NUMA multicore systems, three factors: *remote access penalty*, *shared resource contention*, and *cross-chip data sharing overhead* jointly determine program performance. Last-level cache miss rate has been found quite effective quantifying the shared resource contention [7]. The authors in [40] used the sharing ratio between threads to infer the effect of thread-to-core mapping on performance. As shown in Section 2.2, the dominant factor can change depending on the interplay among the three factors. Therefore, neither of the metrics alone can reflect overall program performance. In this section, we show that the penalty to access the uncore memory subsystem sheds insight on the complex interplay.

Any data access that misses the private mid-level cache (i.e., L2 cache) and the multicore cache coherence traffic are serviced by the uncore. For homogeneous NUMA multicore systems, each core has the same configuration of the out-of-order pipeline and the core memory subsystem (i.e., the private L1 and L2 caches). Therefore, the performance discrepancy due to different thread-to-core assignments only comes from stall cycles from the uncore. The number of memory stall cycles depends on the number of misses and the miss penalty [15]:

$$\text{Uncore stall cycles} = \text{Number of L2 misses} \times \text{L2 miss penalty}$$

Since each core has a L2 cache with the same size, the number of L2 misses is independent of the thread-to-core assignment. Thus, it is the L2 miss penalty, which we call the *uncore penalty*, that makes a difference on program performance. The larger the uncore penalty, the more stall cycles a program experiences.

There are a variety of data sources that may respond to a L2 miss. Table 2 lists the possible uncore responses on Intel Nehalem-based systems [16]. The number of responses `LOCAL_DRAM` and `REMOTE_DRAM` reflects the memory-intensity of a program while their ratio show how well thread scheduling preserves data locality. The rest of the responses characterize the communication patterns between threads. The overall uncore penalty of a program is the summarized penalty of these response types. Therefore, the value of the uncore penalty sheds insight on the complex interplay among the three factors and one can use it to quantitatively compare different thread-to-core assignments, the smaller the uncore penalty, the better the scheduling decision.

3.1. Calculation of the Uncore Penalty

A straightforward way to calculate the uncore penalty is to average over the service times (i.e., latency) of individual responses. However, modern processors are able to service multiple outstanding cache misses in parallel using techniques such as non-blocking caches [21], out-of-order execution with wide instruction windows, and hardware prefetching. As such, memory stalls due to individual L2 misses can overlap with each other. Averaging over the latencies over-counts the stall cycles experienced by a program. As shown in Figure 2, uncore requests wait in the Global Queue before being serviced. In addition to the per socket GQ structure, Intel Nehalem pro-

Uncore response	Description
UNCORE_HIT	LLC hit, unshared
OTHER_CORE_HIT_SNP	LLC hit, shared by another core, same socket
OTHER_CORE_HITM	LLC hit, modified by another core, same socket
REMOTE_CACHE_FWD	LLC miss, shared by another core, different socket
REMOTE_HITM	LLC miss, modified by another core, different socket
LOCAL_DRAM	LLC miss, read from local DRAM
REMOTE_DRAM	LLC miss, read from remote DRAM

Table 2: Uncore response types on Intel Westmere processors.

cessors also provide a per core *super queue* (SQ) structure to track all accesses that miss the L1 caches, which is a super set of the uncore requests.

From the occupancy statistics of the SQ, the uncore penalty experienced by each core can be estimated. According to Little's law, for a time period of t cycles, the average time W (i.e., latency in cycles) that an uncore request stays in the SQ can be calculated as:

$$W = \frac{L}{\lambda},$$

where L is the number of uncore requests in the SQ during this period and λ is the uncore request arrival rate. As discussed in [23, 32], the penalty P associated with each uncore request can be approximated as the ratio of average uncore latency W and the average number of outstanding parallel uncore requests q [2]. Therefore, uncore penalty P can be expressed as:

$$P = \frac{W}{q} = \frac{L/\lambda}{L/t'} = \frac{t'}{\lambda},$$

where t' is the number of cycles that there is at least one uncore request in the SQ and we use L/t' to estimate its queue depth. Based on this formula, one is able to measure the uncore penalty online by monitoring the occupancy and request insertion of the SQ.

3.2. The Effectiveness of the Metric

In this subsection, we show that uncore penalty is a more reliable architectural metric than conventional metrics in the prediction of program performance. A metric, whose value change agrees with the change in program performance, is considered effective in performance prediction. Figure 7 draws the relationship between the measured uncore penalty and the runtime of our benchmark program, under different thread-to-core assignments. For comparison, we also measured the last-level cache miss rate. Note that a reduction in the LLC miss rate indicates mitigated shared cache contention for any workload, but it is almost impossible to find a unified threshold of data sharing ratio to quantify sharing overhead. Thus, we only compare uncore penalty with LLC miss rate in the figure and expect that the sharing factor alone can not accurately reflect performance.

We collected the architectural metrics using Intel VTune [17], a tool that has access to hardware performance counters. More specifically, for the calculation of uncore penalty, we measured the number of cycles that have at least one demand data/instruction load or RFO request outstanding and the number of these requests. Data store and prefetching requests were excluded from the calculation either because their stall cycles can be hidden by the out-of-order execution or because they are not on the critical path of program execution. For cache miss rate, we counted the number of LLC misses per thousand instructions. The configuration of the benchmark programs is identical to those in Figure 5.

The metric value V is normalized to its corresponding measurements in *Intra-S* using $\frac{V_{Inter-S}-V_{Intra-S}}{V_{Intra-S}}$. For example, a runtime bar with a value of -0.5 indicates that the program runtime under *Inter-S* is 50% less than the runtime under *Intra-S*. From Figure 7, we can see that uncore penalty not only reflects program runtime qualitatively but also quantitatively. To characterize the relationship between uncore penalty and program performance, we calculate the *linear correlation coefficient* r between these two metrics. The closer r approaches to 1 (or -1), the stronger a linear relationship exists between the two. As shown in Figure 7, the runtime metric is proportional to the uncore penalty with $r = 0.91$ suggesting a strong positive linear relationship. In comparison, Figure 7 also shows that LLC miss rate only agrees with runtime in a subset of runs, in which the LLC contention was the dominant factor. Overall, LLC miss rate is less accurate in predicting program performance with $r = 0.61$ across all benchmark runs.

In summary, uncore penalty is a more reliable metric for performance prediction than LLC miss rate. By sampling penalties in different thread-to-core assignments, the optimal schedule can be determined as the one with minimum penalty.

4. A NUMA-aware Scheduling Algorithm for Virtual Machines

Our measurement found that data stalls from the uncore are the sources of performance degradation and variation in NUMA systems. Application-level stall optimizations are ineffective in a virtualized environment due to the layer of virtualization between programs and the actual hardware. We therefore strive to add NUMA awareness to the more general virtual machine scheduling at the VMM level. To this end, we propose a Bias Random vCPU Migration (BRM) algorithm on top of the existing scheduler to dynamically adjust the vCPU-to-core assignment. Guided by uncore penalty on each vCPU, the migration algorithm automatically identifies the optimal assignment by minimizing system-wide uncore stalls.

[2] We use the average queue depth of the SQ as an approximation of the number of concurrent uncore requests.

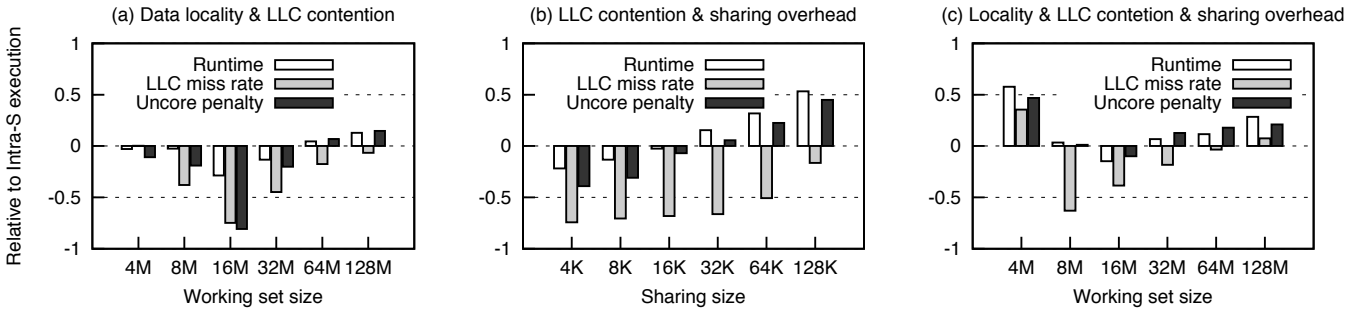

Figure 7: The relationship between architectural metrics and program performance.

4.1. Overview

Our dynamic vCPU migration approach consists of three phases.

1. **Monitoring Uncore Penalty:** The scheduler continuously monitors the uncore penalty of each vCPU by querying the PMUs on the core where the vCPU is running. The measured uncore penalty is saved in the data structure representing the vCPU and is later used by the scheduler to make migration decisions.
2. **Identifying NUMA Migration Candidate:** We rely on the guest OS to identify some vCPUs that run NUMA sensitive threads and then treat these vCPUs differently in making migration decisions.
3. **vCPU Migration:** The scheduler tries to migrate candidate vCPUs to minimize the system-wide uncore penalty. To avoid frequent migrations, a vCPU stays in its current node until there is a high possibility that it will experience less uncore stalls in another node.

In the following subsections, we present details of each phase.

4.2. Monitoring Uncore Penalty

We collect PMU data for each vCPU and calculate its uncore penalty every time the scheduler performs periodic bookkeeping. Each vCPU's uncore penalty is then used to update the system-wide penalty. After that each vCPU saves the latest value of the global penalty into a private array. The array, which is indexed by node IDs, stores the global penalties when the vCPU runs on different nodes. If the migration of the vCPU causes an increment in its own or other vCPUs' uncore penalties, the global penalty increases. Otherwise, the global penalty decreases. As such, we monitor how individual vCPU migrations affect overall uncore penalty perceived by the program. To reduce the influence of measurement noises (e.g., sudden spikes), we use a moving average of penalties over a sliding measurement window. This approach produces fairly stable measurements avoiding premature migrations and is able to adapt to phase changes.

Note that a vCPU's uncore penalty for a node can only be updated when it is running on this node. When program phase changes, the penalties for the nodes, on which the vCPU has not executed recently, are inaccurate. This can negatively affect migration decisions. To address this issue, we reset the uncore penalty for each node after a period of time. This will force the migration algorithm to explore on each node for updated penalties.

4.3. Identifying Migration Candidate

As discussed in Section 2.2, application threads whose execution is dominated by uncore stalls are likely affected by scheduling. Thus, only the vCPUs that carry these scheduling-sensitive threads need to be migrated for better performance. Considering only a subset of the vCPUs for migration also avoids significant changes to the existing vCPU migration algorithm. It is a common practice to consolidate heterogeneous applications onto one virtualized server. Our approach only applies to the application that are sensitive to the NUMA architecture. The rest of co-running applications can still be governed by the original vCPU migration algorithm for load balancing.

We rely on a guest OS to pass the candidate information to the VMM. We observe that NUMA optimizations in modern OSes and runtime libraries usually couple memory allocation policies with CPU affinity. Users may enforce CPU affinity by either launching programs using the `numactl` utility or explicitly specifying affinity with the `sched_setaffinity` system call. We consider the vCPUs, which have application threads affiliated with, to be likely NUMA sensitive, thus identify them as NUMA migration candidates. Modifications to the guest kernel are necessary. Whenever a thread with a non-empty CPU mask is executed on a vCPU, the thread initiates a call down to the VMM with the vCPU ID as the argument. Once receiving the call, the VMM tags the vCPU as a candidate in its scheduler. When the thread exits, it makes another call to clear the tag on the vCPU.

4.4. Bias Random Migration

In multiprocessor scheduling, there are two general approaches to load balancing: *push migration* and *pull migration*. In push migration, the scheduler periodically checks load balance and migrates (or pushes) threads from busy to less-busy cores if an imbalance is found. The key function in push migration is to pick the core where the migrated thread will run next. Pull migration occurs when a core becomes idle and steals (or pulls) a waiting thread from a busy core. Our vCPU migration approach works closely with push migration.

Algorithm 1 shows the BRM algorithm. Whenever the scheduler periodically checks a vCPU for migration (function `Migrate`), it updates the system-wide uncore penalty with new PMU readings (function `Update`) and saves the value in a per-vCPU data structure `v.unc`. If current penalty is larger than the recorded penalty on any other node, the probability

of migrating this vCPU increases. The node bias, which aids the selection of the migration destination, is then modified to point to the node with the minimum penalty. Otherwise, current node has the least uncore stalls and the migration probability decreases, favoring staying with current node. We bound the probability with values of 0 and 100 indicating a "must" migrate and stay situation, respectively.

When selecting a migration destination, the scheduler treats a NUMA migration candidate differently. A NUMA candidate vCPU uses our designed `BiasRandomPick` to select a core with the minimum uncore stalls while a NUMA insensitive vCPU uses the original `DefaultPick` to select a less-busy core. As indicated by its name, `BiasRandomPick` adds some randomness when making a migration decision. A random number is checked against the migration probability to decide whether to migrate or not. The higher the migration probability, the larger chance a vCPU will be migrated. Once deciding to migrate, the destination will be a core in the vCPU's biased node and its migration probability will be reset to zero. A timer is used to generate a random period, after which the uncore penalty of this vCPU is reset for new explorations.

The randomness serves two purposes. First, randomness helps produce more predictable system performance for dynamic workloads [6]. Second, more importantly, randomness avoids expensive synchronization on multiple independent cores. Without the random component, two competing or communicating vCPUs can be migrated at the same time resulting in new competitions on another node or unresolved cross-node communications. A possible solution is to have vCPU migrations synchronized on multiple cores, allowing only one migration at a time. However, such synchronizations limit the scalability of scheduling on many cores. With randomness, migrations are spread over multiple decision windows. As such, vCPUs involved in the competition or communication can be migrated sequentially, approximating the existence of synchronization. If migrations indeed resolve the performance issues, the uncore penalties of the not-yet migrated vCPUs will decrease. Further migrations of these vCPUs can be avoided.

To preserve the fairness and priorities of VMs, we allow the migration decisions made by BRM to be overridden by the default load balancing. More specifically, when a NUMA candidate vCPU is selected by a stealing core, the stealing is temporarily suspended. Only if the stealing core can not find another non-NUMA vCPU to steal, the NUMA vCPU is sent to the stealing core. As such, we preserve BRM decisions as much as possible without affecting the original load balancing.

Algorithm 1 Bias Random Migration Algorithm

```
1: Variables: Virtual CPU v; Current physical core c and node
   n; Node migration bias; Probability to migrate prob; Uncore
   penalty unc.
2:
3: /* Periodic push migration*/
4: procedure MIGRATE(v)
5:     UPDATE(v)
6:     if v is a migration candidate then
7:         new_core = BIASRANDOMPICK(v)
8:     else
9:         new_core = DEFAULTPICK(v)
10:    end if
11:    migrate v to new_core
12: end procedure
13:
14: procedure UPDATE(v)
15:    n = NUMA_CPU_TO_NODE(v.c)
16:    Update global penalty and save it in v.unc[n]
17:    for all nodes i except n do
18:        if v.unc[n] > v.vnc[i] then
19:            v.prob = ++v.prob > 100 ? 100 : v.prob
20:            if v.vnc[v.bias] > v.vnc[i] then
21:                v.bias = i
22:            end if
23:        else
24:            v.prob = --v.prob < 0 ? 0 : v.prob
25:        end if
26:    end for
27: end procedure
28:
29: procedure BIASRANDOMPICK(v)
30:        rand = random() mod 100
31:    if rand < v.prob then
32:        Select a core in node v.bias
33:        Reset v.prob
34:    else
35:        Select current core
36:    end if
37: end procedure
```

5. Implementation

We implemented the support for monitoring uncore penalty, identifying migration candidate, and our proposed BRM algorithm in Xen (version 4.0.2). When modifications were needed in the guest OS, we used the Linux kernel 2.6.32.

We patched Xen with *Perfctr-Xen* [31] to access low-level hardware performance counters. Perfctr-Xen maintains a per-vCPU data structure to store the values of hardware counters. To count events on a per-thread basis, Perfctr-Xen updates counter values at both intra-VM and inter-VM (i.e., thread and vCPU) context switches. Since our objective is to optimize scheduling at the VMM level, we disabled counter updates in the guest OS. Besides vCPU context switches, we also updated the counters every time (every 10ms) the credit scheduler in Xen burns a running vCPU's credits. The number of events occurred between two adjacent updates were used to derive a sample of the uncore penalty. We recursively calculated an exponential weighted moving average (EWMA) of the new sample and the last EWMA:

$$\text{EWMA}_{t+1} = (\text{EWMA}_t \times 7 + \text{Sample})/8.$$

We empirically set EWMA's alpha parameter to 0.125 for a good balance of stability and responsiveness.

We added two new hypercalls `tag` and `clear` to the guest Linux kernel. When a thread is scheduled, we examine its CPU mask (i.e., `cpus_allowed`). If the weight of the mask is

smaller than the number of online CPUs, the `tag` hypercall is triggered with the current running CPU ID as the parameter. Once Xen receives this call, it tags the corresponding vCPU as a NUMA migration candidate. Note that some kernel threads have CPU masks preset by the boot process. The hypercall skips any thread with a PID smaller than 1000 to avoid tagging such kernel threads. We detected program exit by tracking the `exit_group` system call in the guest kernel, and triggered the `clear` hypercall to clear the tag on the vCPU.

Other than using `/dev/random` as the interface for random number generation in the kernel space, we implemented a lightweight random number generator based on the Time Stamp Counter (TSC) register. The TSC register counts the number of cycles since reset. We sampled the values of TSC between updates and counted the number of cycles elapsed since the last update. The last two digits of the elapsed cycles were then used as a random number in the range of [0, 99]. Accordingly, we set the migration probability in the same range. The rationale behind this design is that modern hardware timer's resolution is typically in the granularity of micro-second (approximately 2430 cycles on our platform). Thus, the last two digits are likely device noises and are a good source of randomness.

6. Evaluation

In this section, we present an experimental evaluation of the proposed BRM algorithm using our synthetic micro-benchmark and real-world parallel and multiprogrammed workloads. We compare the performance of BRM with Xen's default credit scheduler and a hand-optimized vCPU binding strategy (Section 6.1). Then we study the stability of BRM in terms of runtime variations (Section 6.2). Finally, we characterize BRM's runtime overhead (Section 6.3).

We ran the experiments on the Intel NUMA machine described in Table 1. Hyperthreading was enabled and therefore the testbed was configured with 16 logical processors. To isolate the NUMA effect from other factors affecting performance, such as power management, we disabled Intel Turbo Boost in BIOS and set the processors to the maximum frequency. We implemented BRM in Xen 4.0.2 and modified Linux guest kernel 2.6.32 for vCPU tagging. The benchmark programs were ran in a VM with 16 vCPUs and 12GB memory. Since the capacity of one memory node is 8GB, the memory of the VM is split onto two nodes.

We selected the following benchmarks and measured their execution times and variations.

- **Micro-benchmark**. As discussed in Section 2.2, the micro-benchmark has multiple configurable parameters. Similar to Figure 5(c), we configured the benchmark with 4 threads and a sharing size of 128KB, and then changed the aggregate working set size from 4MB to 128MB. The execution time of the benchmark is calculated as the average runtime of individual threads.
- **Parallel workload**. We selected the NAS parallel benchmark suite [3] for the parallel workload. The NPB benchmark suite includes 9 parallel benchmarks derived from computational fluid dynamics applications. We excluded the benchmark `is` from the experiments as its execution time is less than 10 seconds. We used the OpenMP implementation of the benchmarks and set the problem size to class C. Four threads were used for each benchmark.
- **Multiprogrammed workload**. We constructed the multiprogrammed workload from the SPEC CPU2006 benchmarks [37]. Since the benchmarks in the suite are independent of each other (no data sharing), only the memory-bound benchmarks are sensitive to the scheduling on NUMA systems. We selected four memory-bound benchmarks (`mcf`, `milc`, `soplex`, `sphinx3`) and formed five workloads. For instance, workload *mcf* consisted of four identical copies of the `mcf` benchmark. The fifth workload *mixed* also had four threads, but each thread ran a different benchmark. The execution time of a workload is the average of runtimes of individual benchmarks. For the mixed workload, individual runtimes are first normalized to runtimes of the corresponding benchmarks when running solo.

We evaluated three scheduling strategies: default Xen, hand-optimized, and BRM. The default Xen scheduling strategy performs both pull and push vCPU migrations in order to place the vCPU on the least loaded core. For hand-optimized scheduling, we ran the workloads with different vCPU-to-core bindings offline and selected the binding with the best performance for individual workloads. The hand-optimized strategy keeps the best binding during the execution of workloads. BRM places (pushes) vCPUs on cores with the lowest uncore penalty. BRM allows a NUMA migration vCPU to be stolen (pulled) by an idle core only if there are no other migratable vCPUs and otherwise the system becomes imbalanced.

6.1. Improvement on Program Performance

Figure 8 shows the runtimes of workloads under different scheduling strategies. Each runtime is the average of ten program runs under the same strategy and is normalized to the runtime under default Xen scheduler. For the micro-benchmark (Figure 8(a)), BRM outperformed Xen by at least 3.1% (WSS=32M) and by as much as 19.8% (WSS=4M). Since vCPU migrations are not free, it is expected that BRM incurred performance degradations compared with the hand-optimized strategy. However, BRM had performance close to the hand-optimized strategy with no more than 3.4% degradation. Surprisingly, BRM even achieved a better performance in some tests (WSS=8M, 16M) than the hand-optimized strategy. We profiled the benchmark execution and found that there was a varying number of threads competing for a shared cache line at different iterations, leading to switching dominant factors. Different from the hand-optimized strategy, which had fixed vCPU bindings, BRM was able to find a better vCPU-to-core assignments for different iterations.

Figure 8(b) shows the results of the parallel workload. We arrange the NPB benchmarks according to their orders in Figure 1 with benchmarks on the left more sensitive to the scheduling on NUMA systems. From the figure, we can see that BRM is more effective for benchmarks that are more sensitive to the NUMA architecture. For example, BRM improved the performance of `lu` significantly by 31.7% while

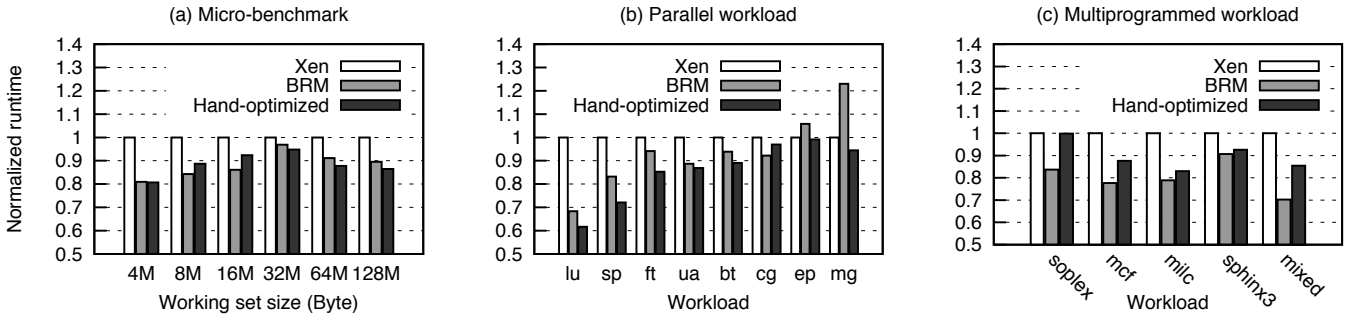

Figure 8: BRM improves program performance.

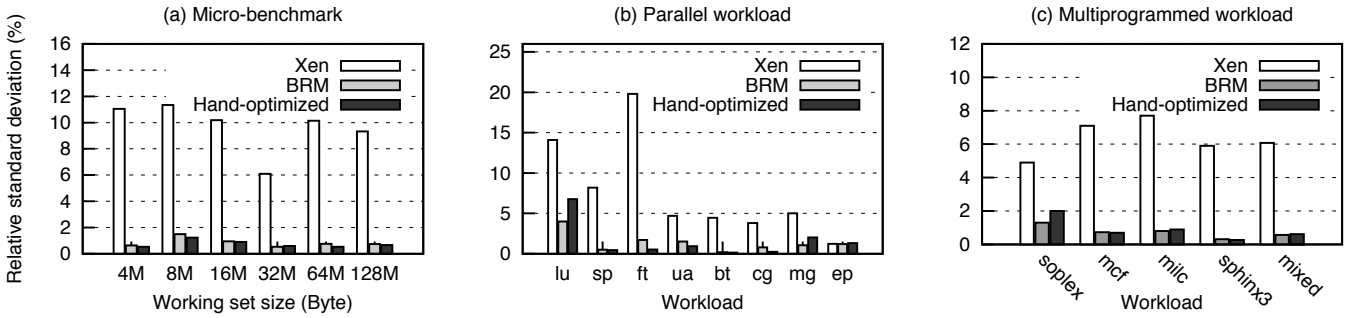

Figure 9: BRM reduces runtime variations.

only improved `cg`'s performance moderately by 7.8%. For benchmarks `ep` and `mg`, which are not sensitive to the NUMA architecture according to Figure 1, vCPU migrations in BRM failed to find sufficient room for performance optimization but added some overhead. Compared with the Xen default scheduler, BRM incurred performance degradations of 5.8% and 21.9% on `ep` and `mg`, respectively.

For multiprogrammed workloads (Figure 8(c)), BRM outperformed both the hand-optimized strategy and Xen default scheduler in all workloads. Particularly, for the *mixed* workload, the improvements are 15.1% and 29.7%, respectively. The reasons for such great performance improvements are twofold. First, as shown in Figure 1, the benchmarks we selected to construct the workloads are NUMA sensitive and there is sufficient room for optimization via scheduling. We expect that BRM causes degradations on compute-bound SPEC CPU2006 benchmarks and do not show it here due to the space limit. Second, during the execution of the benchmarks, there were significant variations in memory access frequency. Thus, at different time points in the execution, the workload may have different preference in scheduling affinity. Fixed (hand-optimized) or NUMA-unaware (Xen default) scheduling strategy was unable to adapt to the workload changes. In contrast, by sampling uncore penalties, BRM successfully identified the changes and found the optimal scheduling affinity.

6.2. Reduction in Runtime Variation

In Figure 9, we compare BRM with the other two scheduling strategies in terms of runtime variations. We calculated the *relative standard deviations (RSD)* for a set of 10 runs of individual workloads under different strategies. RSD measures the extent of variability across program executions. The smaller the RSD value, the more consistent and predictable program performance. Since the hand-optimized strategy had fixed vCPU-to-core bindings, as expected, it achieved small RSD values in all workloads. Except for `lu` (RSD=6.76%), the hand-optimized strategy was able to provide predictable performance of workloads with no more than 2% variations. The default Xen scheduler, which only takes into account the business of cores in scheduling, caused considerable variations. For NUMA sensitive workloads, the default scheduler had variations as large as 19.8% (benchmark `ft`). In comparison, BRM achieved a close level of variations to the hand-optimized strategy. On average, BRM also had no more than 2% variations. Note that BRM even outperformed the hand-optimized strategy in `lu`. A possible explanation is that, for long-lived workloads, vCPU migrations in BRM converge to similar trajectories across different runs, resulting in stable program runtimes. Although with a fixed scheduling affinity, thread progresses vary in different runs contributing to the variations of the hand-optimized strategy.

6.3. Overhead and Scalability

The main runtime overhead of BRM is due to updating the system-wide uncore penalty. As the number of vCPUs managed by BRM increases, there could be scalability issues when multiple vCPUs attempt to acquire a lock and perform an update. To quantify the overhead, we disabled the vCPU migration in BRM and the Xen default scheduler. As such, BRM still maintained the information about uncore penalties for each tagged vCPU, but without using it in scheduling. We ran the parallel workload with a varying number of threads.

Figure 10 draws the overhead of BRM normalized to the default scheduler. Note that the number of threads is equivalent to the number of tagged vCPUs in BRM as we assume a one-to-one mapping from user-level threads to vCPUs in VMs. As

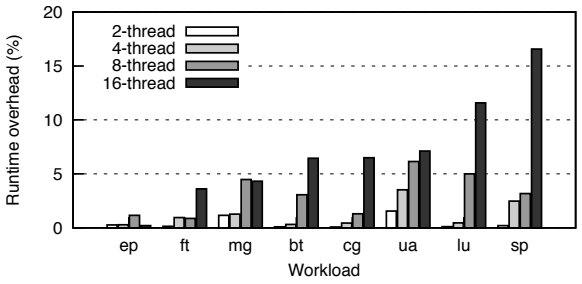

Figure 10: BRM runtime overhead.

shown in Figure 10, BRM incurred, on average, less than 2% overhead for 2-thread and 4-thread workloads. For 8-thread workloads, the overhead is more apparent but still acceptable. For example, BRM caused a overhead of 6.14% on `ua`, the largest overhead among all 8-thread workloads. Given the overhead, `ua` can still benefit from BRM scheduling as it may have a worst case degradation of 18.8% according to Figure 1.

Running 16-thread workloads is more problematic. BRM caused more than 10% overhead to `lu` and `sp`. Since a 16-thread workload has only one thread-to-core mapping on our testbed, there is no room for optimization via scheduling. It is unclear whether BRM's optimization outweighs its overhead on a larger testbed where multiple scheduling affinities exist. We plan to investigate this issue once we are given access to such systems. Although BRM may have scalability issues when its managed workload has more than 16 threads, we argue that it is still an attractive approach to optimizing program performance in existing cloud platforms. In Amazon EC2, the largest instance (i.e., the *High-CPU Extra Large Instance*) has up to 8 vCPUs, inferring a maximum concurrency of 8 for most single-instance cloud workloads. For such workloads, BRM is able to provide optimized and predictable performance.

7. Related Work

There has been significant interest in the research community in improving program throughput, fairness, and predictability on multicore systems. Existing work has addressed these issues via thread scheduling or program and system-level optimizations.

7.1. Optimization via Scheduling

The contention on the shared LLC has been long accused for severe performance degradation and unfairness [7, 9, 14, 18, 22]. Recent work shows that contentions on the hardware prefetcher [25], the memory controller [27, 30] and the DRAM bus [11] can also cause significant performance slowdown in both UMA and NUMA systems. Last-level cache miss rate has been widely used as a proxy for the contention on shared resources [7, 8, 9, 14, 26] and the similarity in thread address spaces has been used to quantify the inter-thread sharing activity [5, 35, 38].

However, on NUMA multicore systems, data locality, shared resource contention, and cross-node communication overhead interact in complex ways that the stated metrics can not accurately predict program performance. The statistics on the offcore requests shed some insight on the complex factors interplay. The metric of offcore request has been used for selecting appropriate core types in heterogeneous multcore systems [20] and for analysing cloud workloads [12]. In this work, we use the penalty of offcore requests as a runtime performance index of high-level performance and use it to guide vCPU migration in a virtualized environment.

Based on the performance metrics, scheduling decisions are made either distributing threads across different sockets for contention mitigation, or grouping them onto one socket for efficient communication. Finding the optimal thread-to-core assignment is not trivial. Some work uses offline profiling and derives models to predict the optimal mapping [26, 27, 34]; other work derives the mapping by monitoring hardware performance counters online [4, 7, 8]. Once the optimal mapping is determined, the decision is applied to the system either by user-level schedulers [36, 41] or by modified kernel schedulers [19, 29]. In a cloud environment, CPU masks enforced by user-level schedulers create isolated resource islands among co-running programs. The partition of resources breaks the enforcement of fairness and proportional sharing. We implemented BRM at the VMM level and allowed the decision made by BRM to be overwritten for load balancing.

7.2. Program and System-Level Optimization

Program-level transformations are widely used methods in performance optimization. Zhang *et al*. [40] transform programs in a cache-sharing-aware manner and Majo *et al*. [28] aim to preserve program data locality on NUMA systems. Such transformations can be further made transparent to users and automated at compiler level [39]. Modern OSes, such as Linux and Solaris also provide system support for NUMA optimizations. Memory allocations are delayed until a thread first accesses the memory address in order to guarantee local memory access. Runtime libraries, such as *libnuma*, provide system APIs to explicitly specify the location of memory allocation and CPU affinity. Modern VMMs allow a guest OS to discover its virtual NUMA topology either by reading the emulated ACPI Static Resource Affinity Table (SRAT) [2], or by querying the VMM via para-virtualized hypercalls [33].

However, program and guest system-level optimizations depend critically on how accurately the virtual architecture reflects the machine architecture. Due to dynamic load balancing in the virtualization layer, the virtual topology observed by the guest OS may become inaccurate [2], which significantly affects the effectiveness of stated optimizations. In contrast, our approach does not assume any program or system-level optimizations and directly works in the virtualization layer.

8. Conclusions and Future Work

Non-Uniform Memory Access multicore architectures impose significant challenges to the delivery of optimal and predictable program performance. The introduction of virtualization further complicates the problem by limiting the visibility of the hardware hierarchy to programs. To address these issues, we identified a hardware metric, the uncore penalty, to measure the high-level program performance online. We

then proposed Bias Random vCPU Migration, a load balancing algorithm that uses the metric to determine the optimal vCPU-to-core assignment. We implemented BRM on top of Xen's credit scheduler. For parallel and multiprogrammed workloads, BRM improved performance by up to 31.7% compared with Xen's default scheduler and provided predictable performance with, on average, no more than 2% variations.

We currently employ an intrusive approach to identifying NUMA candidate vCPUs. Non-intrusive approaches are sometimes more desirable as no changes need to be made to the guest OS. An alternative approach is to infer the NUMA sensitivity at the VMM level. If a vCPU issues uncore requests frequently, it can be considered as NUMA sensitive. Similarly, a vCPU is removed from the candidates if the change in its uncore penalty is less than a predefined threshold when migrated. However, work should be done to study the trade-off of intrusiveness and inference accuracy.

Another extension of our current work is to improve BRM's scalability. The centralized update of the system-wide uncore penalty is inevitably a bottleneck. As in [24], we could allow unprotected access to the global penalty trading accuracy for scalability. Another possible solution is to equip multiple schedulers, each of which runs a copy of BRM. As such, the lock contention is mitigated by using multiple locks. The new feature of `cpupools` [1] in Xen version 4.2 could be a good starting point of multiple schedulers.

Acknowledgements

We are grateful to the anonymous reviewers for their constructive comments. This research was supported in part by the U.S. National Science Foundation under grants CNS-0914330, CCF-1016966, CNS-0844983, and CNS-1217979.

References

[1] http://blog.xen.org/index.php/2012/04/23/xen-4-2-cpupools/.

[2] Q. Ali, V. Kiriansky, J. Simons, and P. Zaroo, "Performance evaluation of HPC benchmarks on VMware's ESXi server," in *Proc. of ICPP*, 2011.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in *Proc. of SC*, 1991.

[4] M. Banikazemi, D. Poff, and B. Abali, "PAM: a novel performance/power aware meta-scheduler for multi-core systems," in *Proc. of SC*, 2008.

[5] F. Bellosa and M. Steckermeier, "The performance implications of locality information usage in shared-memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 37, no. 1, 1996.

[6] T. L. Blackwell, "Applications of randomness in system performance measurement," Ph.D. dissertation, 1998.

[7] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Trans. Comput. Syst.*, vol. 28, no. 4, 2010.

[8] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *Proc. of USENIX ATC*, 2011.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin, "Predicting inter-thread cache contention on a chip multi-processor architecture," in *Proc. of HPCA*, 2005.

[10] U. Drepper, "What every programmer should know about memory," 2007.

[11] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt, "Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems," in *Proc. of ASPLOS*, 2010.

[12] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Proc. of ASPLOS*, 2012.

[13] A. Gulati, G. Shanmuganathan, A. Holler, and I. Ahmad, "Cloud-scale resource management: Challenges and techniques," in *Proc. of HotCloud*, 2011.

[14] F. Guo and Y. Solihin, "A framework for providing quality of service in chip multi-processors," in *Proc. of MICRO*, 2007.

[15] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Morgan Kaufmann Publishers Inc., 2011.

[16] Intel Corporation, *Intel® 64 and IA-32 Architectures Software Developer's Manual*, December 2009.

[17] Intel VTune Amplifier XE Performance Profiler., http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[18] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt, "QoS policy and architecture for cache/memory in cmp platforms," in *Proc. of SIGMETRICS*, 2007.

[19] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, "Using OS observations to improve performance in multicore systems," *IEEE Micro*, vol. 28, 2008.

[20] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proc. of EuroSys*, 2010.

[21] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *Proc. of ISCA*, 1981.

[22] P. Lama and X. Zhou, "NINEPIN: Non-invasive and energy efficient performance isolation in virtualized servers," in *Proc. of DSN*, 2012.

[23] D. Levinthal, *Performance Analysis Guide for Intel® Core i7 Processor and Intel® Xeon 5500 processors*, 2009.

[24] T. Li, D. Baumberger, and S. Hahn, "Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin," in *Proc. of PPoPP*, 2009.

[25] F. Liu and Y. Solihin, "Studying the impact of hardware prefetching and bandwidth partitioning in chip-multiprocessors," in *Proc. of SIGMETRICS*, 2011.

[26] Z. Majo and T. Gross, "Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead," in *Proc. of ISMM*, 2010.

[27] Z. Majo and T. Gross, "Memory system performance in a NUMA multicore multiprocessor," in *Proc. of SYSTOR*, 2011.

[28] Z. Majo and T. Gross, "Matching memory access patterns and data placement for numa systems," in *Proc. of CGO*, 2012.

[29] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious scheduling for energy efficiency on multicore processors," in *Proc. of EuroSys*, 2010.

[30] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proc. of MICRO*, 2007.

[31] R. Nikolaev and G. Back, "Perfctr-xen: a framework for performance counter virtualization," in *Proc. of VEE*, 2011.

[32] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt, "A case for MLP-Aware cache replacement," in *Proc. of ISCA*, 2006.

[33] D. Rao and K. Schwan, "vNUMA-mgr: Managing vm memory on numa platforms," in *Proc. of HiPC*, 2010.

[34] A. Snavely and D. M. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreaded processor," in *Proc. of ASPLOS*, 2000.

[35] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proc. of EuroSys*, 2007.

[36] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Proc. of ISCA*, 2011.

[37] The SPEC CPU2006 Benchmarks., http://www.spec.org/cpu2006/.

[38] R. Thekkath and S. J. Eggers, "Impact of sharing-based thread placement on multithreaded architectures," in *Proc. of ISCA*, 1994.

[39] Q. Yi, "Automated programmable control and parameterization of compiler optimizations," in *Proc. CGO*, 2011.

[40] E. Z. Zhang, Y. Jiang, and X. Shen, "Does cache sharing on modern CMP matter to the performance of contemporary multithreaded programs?" in *Proc. of PPoPP*, 2010.

[41] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling." in *Proc. ASPLOS*, 2010.

[42] S. Zhuravlev, J. Saez C., , S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Computing Surveys*, vol. 45, no. 1, 2013.