Jericho: Achieving Scalability Through Optimal Data Placement on Multicore Systems

Stelios Mavridis[1], Yannis Sfakianakis[1], Anastasios Papagiannis, Manolis Marazakis and Angelos Bilas[1]
Foundation for Research and Technology - Hellas (FORTH)
Institute of Computer Science (ICS)
100 N. Plastira Av., Vassilika Vouton, Heraklion, GR-70013, Greece
{mavridis, jsfakian, apapag, maraz, bilas}@ics.forth.gr

***Abstract*—Achieving high I/O throughput on modern servers presents significant challenges. With increasing core counts, server memory architectures become less uniform, both in terms of latency as well as bandwidth. In particular, the bandwidth of the interconnect among *NUMA* nodes is limited compared to local memory bandwidth. Moreover, interconnect congestion and contention introduce additional latency on remote accesses. These challenges severely limit the maximum achievable storage throughput and IOPS rate. Therefore, data and thread placement are critical for data-intensive applications running on *NUMA* architectures. In this paper we present Jericho, a new I/O stack for the Linux kernel that improves affinity between application threads, kernel threads, and buffers in the storage I/O path. Jericho consists of a NUMA-aware filesystem and a DRAM cache organized in slices mapped to NUMA nodes. The Jericho filesystem implements our task placement policy by dynamically migrating application threads that issue I/Os based on the location of the corresponding I/O buffers. The Jericho DRAM I/O cache, a replacement for the Linux page-cache, splits buffer memory in slices, and uses per-slice kernel I/O threads for I/O request processing. Our evaluation shows that running the *FIO* microbenchmark on a modern 64-core server with an unmodified Linux kernel results in only 5% of the memory accesses being served by local memory. With Jericho, more than 95% of accesses become local, with a corresponding 2x performance improvement.**

I. Introduction

Modern servers have switched from SMP to *NUMA* memory architectures, for scaling the number of cores as well as the capacity and performance of memory. *NUMA* architectures consist of nodes, each node having its own *local* memory controller and high speed interconnect to other *remote* nodes. By increasing the *NUMA* nodes in a system, we can scale memory in terms of capacity as well as aggregate throughput.All these design choices result in different memory latency and bandwidth towards local and remote memory, with *remote* accesses being significantly slower [1], [2]. Application performance varies over a disturbingly wide range, depending on the affinity of threads to memory pages. Moreover, due to possible contention in memory controllers and queuing delays in the interconnect, applications may experience further performance degradation.

Applications seriously affected by *NUMA* effects include scientific applications using MapReduce and similar frameworks, virtual machine workloads, and other I/O-intensive applications, especially when application threads access independent file sets. Focusing on the later, a major cause of performance degradation is the weak buffer affinity policies in the Linux page cache.

The Linux kernel has been *NUMA*-aware since version 2.5 [3]. System memory is organized into zones, each of them corresponding to a single *NUMA* node. Accordingly the page cache became *NUMA*-aware by using the *first-touch* policy. When a miss occurs, the newly allocated buffer is allocated from the issuer's *local NUMA* node. This eliminates *remote* accesses as long as all later accesses happen from a CPU core in the same *NUMA* node. The problem with current versions of the Linux kernel is that the task scheduler implements a weak affinity task placement policy. This means that task migrations will not always keep tasks in the same *NUMA* node as their buffer, breaking affinity.

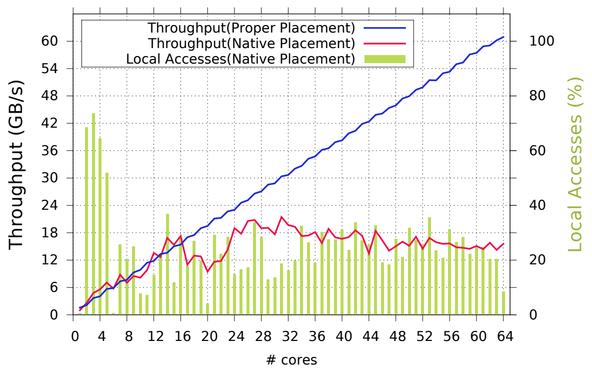

Fig. 1. Comparison of proper and native task placement with *FIO*.

Figure 1 shows how the *FIO* microbenchmark [4] scales on a 64-core *NUMA* server. The entire dataset fits in the server's page cache, i.e. there is no I/O device activity. There is one thread per core, each accessing its own data file. The *proper placement* curve presents performance under optimal placement, where the buffers for each of the data files are from a single *NUMA* node and each task only accesses buffers for its local *NUMA* node. The *native placement* curve shows an execution without enforcing affinity. The bars in the background of the graph show the ratio of *local* memory accesses in the *native placement* run. For the *proper placement* run, the *local* ratio is always close to 100%. Without explicit placement for achieving locality, the Linux scheduler migrates threads, thereby leading to *remote* memory accesses, and performance does not scale with more than 16 threads. With explicit placement, threads run only on cores of the *NUMA* node where their accessed data reside, generating exclusively *local* memory accesses, with throughput scaling up to 64 threads. The throughput difference between the two configurations is

[1]Also with the Department of Computer Science, University of Crete, Heraklion, Greece.

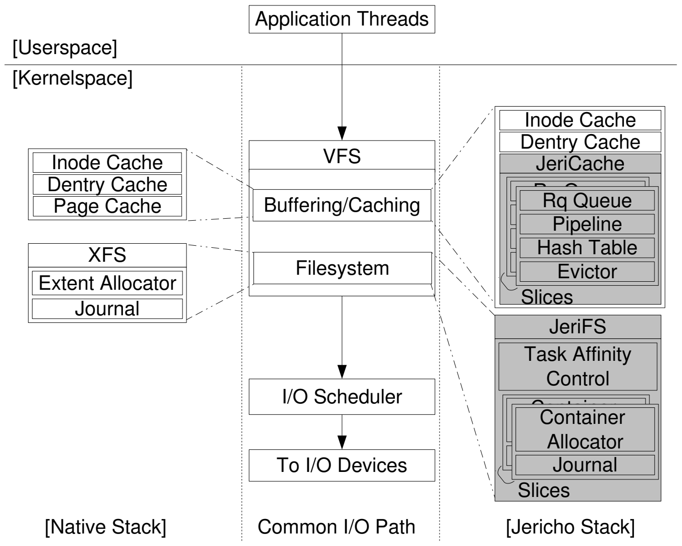

Fig. 2. Comparison of native IO stack versus *Jericho*.

3.3x with 64 threads, showing the dramatic impact of locality management on workloads executing on a *NUMA* server.

This paper introduces *Jericho*, a redesigned I/O stack aiming to achieve optimal affinity of data and processing contexts in manycore systems. *Jericho* consists of a *NUMA*-aware filesystem and a replacement for the Linux page cache. This approach is necessary to enforce placement constraints for both tasks and I/O buffers.

In order to accomplish our design goals a radical redesign of the I/O stack is necessary. As shown in Figure 2, our I/O stack modifies key components of the VFS layer. The existing page cache is bypassed and our own *JeriCache* is used instead. With the existing page cache, although the first-touch policy results in buffers being allocated locally to the I/O-issuing tasks, later task migrations (after context switches) may break affinity. Our page cache replacement, *JeriCache*, is organized as as a group of independent caches. We use the term 'slice' to describe a single cache instance, limited to using memory from a single *NUMA* node and caching a specified range of blocks from an underlying storage device. Mapping storage block ranges to *NUMA* nodes allows us to implement policies based on the actual data set of each application, instead of tracking I/O requests and I/O buffers in the Linux kernel. Having a custom filesystem, *JeriFS*, allows us to determine at the time of each I/O request whether the I/O-issuing task is placed at the same *NUMA* node as the requested data. The storage space managed by this filesystem consists of a set of block ranges from the underlying storage that correspond to the *JeriCache* slices. This arrangement allows to place files to use storage from a specific storage block range and, moreover, I/O buffers from a single *NUMA* node.

We compare *Jericho* with the unmodified Linux kernel and we find that for 64 threads we improve sequential read I/O throughput to 1.8x over the baseline system, and sequential write I/O throughput by 2.5x. Similar improvements are achieved for random IOPS performance.

Overall, our contributions in this paper are:

- We quantify the impact of thread and data placement on a *NUMA* server.
- A simple locality management scheme, allowing proper affinity with negligible overhead.
- A *NUMA*-aware filesystem, capable of enforcing correct task placement.
- A page-cache design capable of controlling page placement.

The rest of this paper is organized as follows. Section II describes the design of *Jericho*. Section III presents our experimental results and compare our approach with two unmodified Linux kernel versions. Section IV reviews prior related work. We summarize our conclusions in Section V.

II. Design

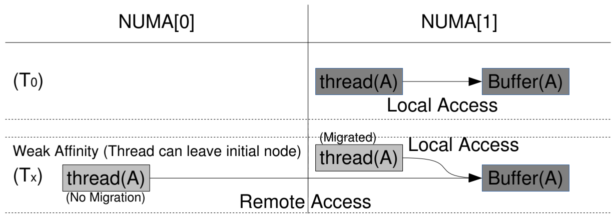

Fig. 3. Locality Management in Linux.

A. Locality Management in Linux

In this paper, we address the issue of how to improve thread-to-data affinity for data-intensive workloads running on a *NUMA* servers. Intuitively, to achieve consistently good performance we need to keep each thread close to its data, avoiding as much as possible the migration of individual processes to a different processor core. Migration from one core to another can be very expensive, due to the loss of 'warm state' in L1 and L2 caches. This cost is particularly severe for migrations across *NUMA* nodes, where L3 cache state is also lost for the migrating thread.

The Linux kernel offers primitives for allocating memory from specific memory nodes, as well as primitives to influence thread scheduling based on preferences about the cores to be used. Their use by applications is limited however. To ensure local memory accesses they use a first-touch policy, where buffer placement is decided at the first access. The scheduler's *weak-affinity* policy however allows a task to be migrated across nodes. This in turn negates any effect of the memory placement policy (Figure 3).

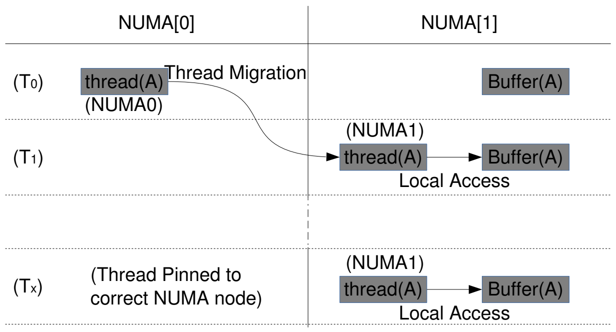

Fig. 4. Locality Management in *Jericho*.

B. *Locality Management in* Jericho

Achieving proper task/data affinity requires task and buffer placement. In our design task placement policies are applied at the filesystem. Buffer placement is handled by our page cache. In the following subsections, we describe the essential aspects of *Jericho*.

C. JeriFS

JeriFS is a custom filesystem that implements key VFS operations such as read, write, mkdir, open, close, fsync . The filesystem uses the inode and directory (dentry) cache. We bypass the existing page cache, using our own *JeriCache* design (detailed in Section II-D).

Exploiting the fact that filesystem code runs on the same context (albeit in kernel space) as the user application, we can migrate the user context without having to modify scheduler code. At every filesystem operation, the filesystem checks the current affinity of the issuing thread and acts accordingly (see Figure 4). Whenever the thread is attempting to access remote buffers the filesystem alters the thread's affinity, limiting it to run only on the CPU core of the *NUMA* node where the buffer is placed. Finally, it forces a context switch, effectively migrating the process to a local node. When the process resumes execution on its 'home' node the I/O operation is guaranteed to only cause local memory references. Although migrating a process is an expensive operation, the cost is amortized since every process issues multiple subsequent filesystem requests.

Having ensured tasks run only on CPUs of the 'home' node, correct memory placement must be ensured. Similar to other modern filesystems, *JeriFS* uses extents and containers[1], for block allocation. This allows for high concurrency of allocations and better disk layout. We divide the underlying storage into slices, each slice having a home node. Assigning each slice a 'home' node, ensures that each *NUMA* node is responsible for an equal volume of data. Deciding slice sizes and load placement policies based on number of processes or throughput,is not a suitable heuristic as these can vary greatly during execution. We instead opted for a directory based, round-robin policy, where every directory in the root of *JeriFS* is assigned a slice. This provides a user-controlled mechanism for assigning workloads to *NUMA* nodes. Users can implement policies better suited to their usage scenarios.

D. JeriCache

The *Jericho* page-cache, implemented as a kernel module, supports multiple independent write-back caches over a shared pool of page-sized buffers. For each cache instance (slice), we explicitly specify from which memory nodes to consume memory. Each cache is controlled by a private state machine in a separate kernel thread that allows for explicit placement and reduces cross-core interference.

1) Cache Operations: *JeriCache* supports a pointer-based interface to buffers, to avoid data copies and page migrations across *NUMA* nodes. The API supported by *Jericho* provides our custom filesystem with direct access to page-sized buffers managed by a cache instance. To serve a read or write request, *JeriFS* issues a *Get* call for the desired block range. Write requests to blocks holding filesystem metadata are marked (by our custom filesystem code), so that the cache handles any modifications to filesystem metadata blocks on storage in a write-through fashion. This is essential for recoverability in the event of failure.

The *waitForGet* call blocks the caller until all of the requested blocks are available in the cache. Upon completion, the caller receives a pointer to the appropriate buffers as well as an opaque handle. The cache maintains a reference count for each block, implemented as an atomic counter. For a buffer to become eligible for release from the cache, the reference count must be equal to zero. Modified ("dirty") buffers will be flushed to the underlying storage before re-use. The caller is expected to issue a *Put* call when it no longer needs the buffers provided by a previous successful *Get* call. For write (and read-modify-write) calls, the caller also needs to provide an indication if a buffer has been modified, by the means of a dirty flag. The *waitForPut* call allows a caller to block until all modified blocks, if any, have been flushed.

The *JeriCache* assists the filesystem in implementing the *fsync* system call. The filesystem determines which blocks need to be flushed to the underlying storage and then issues the *syncRange* call. As with the *Get*/*Put* calls, the user then can call *waitForSync* which blocks until all specified blocks have been written to the underlying storage device.

2) Request processing and data structures: As our design revolves around *NUMA* slices, each slice corresponds to a cache instance. Every cache instance uses a hash table as its lookup structure. Each hash table consists of double-linked bucket lists with each entry contains a packed array of elements. Element packing was inspired in part by the work in [5]. Each element describes a page-sized buffer along with a reference counter, flags and a timestamp. Packing multiple elements together is an optimization for improving processor cache utilization. Since the cacheline size on current-generation servers is 64 bytes and the common pointer size is 8 bytes, we use the first 16 bytes as pointers for the next and previous elements packed elements. In the remaining 48 bytes of the cacheline, we place identifiers for the packed elements. By placing these items in a single cacheline, we reduce CPU cache misses on sequential scans. The remainder of the packed array contains page pointers, flags and timestamps. Using our element packing has reduced our metadata overhead along with the reduction of memory accesses and the resulting reduction of CPU cache misses. Just Similar to the authors of [5] we saw a 10-25% improvement in performance in most cases.

Hash tables have been used in page cache implementations before. Linux used a hash table for the page cache until version 2.4. In 2.4 it was replaced by an *RCU* [6] radix tree, as the single lock protected hash table showed poor scalability.

JeriCache uses a hash table protected with fine-grain locks. Every bucket is protected by a lock, this coupled with the hash function distribution results in very rare collisions and lock contention. This approach achieves a good balance between memory overhead and lock contention. Additionally we made the bucket head structure fit in a single cache line, thus eliminating false sharing.

Each cache instance is supported by two running contexts

[1]Containers are similar in function to the *XFS Allocation Groups*

together with bottom-half interrupt contexts. The pipeline thread is responsible for processing cache requests in stages. The evictor thread implements the replacement policy. Interrupt contexts handle I/O completion notifications and delegate processing to the pipeline thread. In the case of cache hits, we avoid triggering the pipeline thread and we "inline" the request processing flow in the I/O issuing context. This avoids the impact of a context switch. In the case of cache misses, we need to enqueue the request for processing by the pipeline thread. The pipeline thread implements the following stages for each I/O:

- **New Stage** This is the pipeline's entry point. For each page in every request we perform an atomic lookup-insert operation. When a lookup results in a miss, a new element is inserted in the cache. If on a miss, a buffer for the newly inserted element cannot be found, the request is moved to the stalled stage until the necessary page(s) are available. On all hits, the reference counter of the found element is increased. In cases where the element is being written or read from storage and the request or Get has already acquired the element, it is then put on the Clashed stage until the element becomes available again.
- **Stalled Stage** Requests in this stage have had one or more pages resulting in a miss and the required pages could not be acquired from the cache's group. They stay in this stage until the evictor reclaims the required pages. After pages are assigned to the elements any related I/O is issued.
- **Clashed Stage** Requests in this stage have one or more elements that are either Stalled or part of an I/O (read, or flush). The request has to wait for the I/O to be completed or the element to acquire its page.
- **Execute Stage** This is the last stage of the pipeline where the user provided completion callback is executed and the request structure is returned to the free request pool.

3) Flushing, Evictions, and Replacement: Eviction is implemented by a separate thread per cache instance. The evictor thread is responsible for both flushing dirty elements and keeping the cache within user-specified size limits. The flushing and eviction of dirty elements are coupled in the current design. When creating a cache instance the user provides two watermark-type thresholds for cache occupancy to control when the evictor thread becomes active and for how long.

4) Cache Replacement Policy: Jericho implements an LRU replacement policy. Instead of a list-based LRU implementation we use timestamps to approximate staleness and reduce synchronization overheads. The evictor thread scans the buffers in the hash table and evicts buffers with a timestamp older than a threshold. If a buffer is clean and eligible for eviction, it is immediately and atomically removed from the cache. If a dirty buffer is encountered, then it is put in a queue of buffers to be flushed. The evictor then searches for the next (consecutive) buffer in the cache; if it finds such a consecutive buffer that is dirty and eligible for eviction, it appends it to the flush queue. When a consecutive buffer does not exist or cannot be evicted, the flush queue is sent to the underlying storage device as a single (usually large-sized) write request. Issuing large writes takes better advantage of storage device throughput therefore resulting in better performance. After issuing the write request, the evictor resumes the hash table's scan.

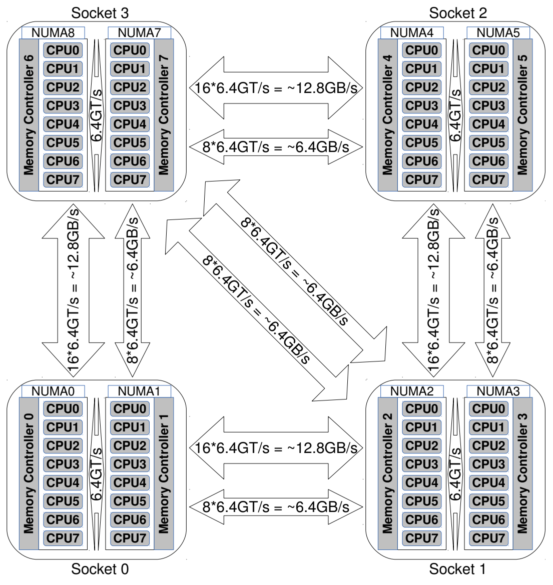

Fig. 7. Organization of an 8 *NUMA* node server and 64 cores. Unidirectional bandwidth inside arrows.

Processor socket count	4
Cores/processor socket	16
Motherboard	Tyan S8812
Processor type	AMD Opteron 6272 (2.1GHz)
Processor core caches	L1: 2x32KB (code - per 2 cores), 16KB (data) L2: 4x2MB (per 2 cores) L3: 2x8MB (per 4 cores)
DRAM (DDR3, # DIMMs)	Up to 16 (up to 512 GB, currently 256 GB)
Interconnect type	HyperTransport 3.1
Interconnect topology	Point-to-Point, asymmetric
Storage Devices	16 Solid State Disks (Samsung 830 Series) in RAID0 configuration
Storage Controllers	2x LSI MegaRAID SAS 9265-8i controllers, each with 8 Solid State Disks attached

TABLE I. EVALUATION SETUP

III. EVALUATION

A. Experimental Testbed and Methodology

Figure 7 depicts the layout of the 8-node, 64-core server we use in our evaluation [7]. Memory accesses to a local node go through a local memory controller. As each memory controller uses double channel DDR3 modules the maximum theoretical throughput is 21.3 Gb/s per memory controller [8]. Each socket contains two memory controllers with an aggregate 42.6 GBytes of throughput, and 16 processing cores. Remote accesses on the other hand have to pass through one or two interconnect links. This topology results in non-uniform memory access times. In addition the interconnect bandwidth is limited compared to the available memory bandwidth.

This system topology results in many different choices for the relative placement of threads and their corresponding data-sets result in widely varying performance levels. Many of these placements can potentially lead to congestion on certain interconnect links and overload for some memory controllers in the system [9].

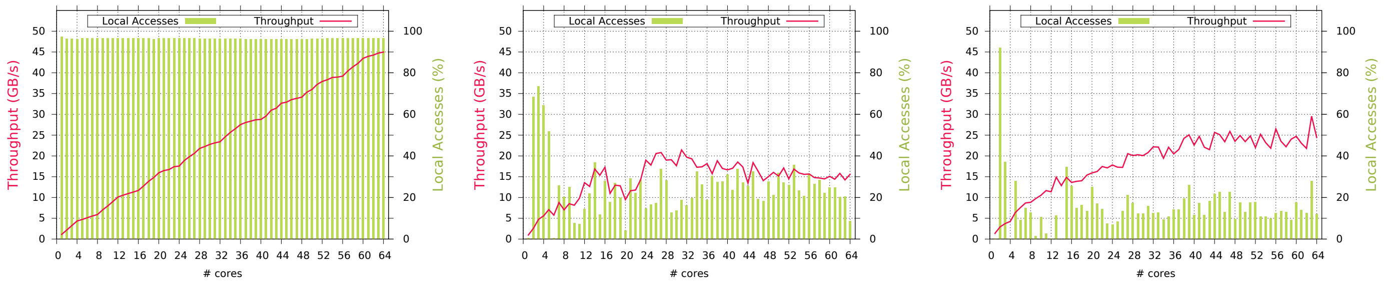

Fig. 5. *FIO* Reads for *Jericho* (left), Native 2.6.32 (middle) and Native 3.13 (right).

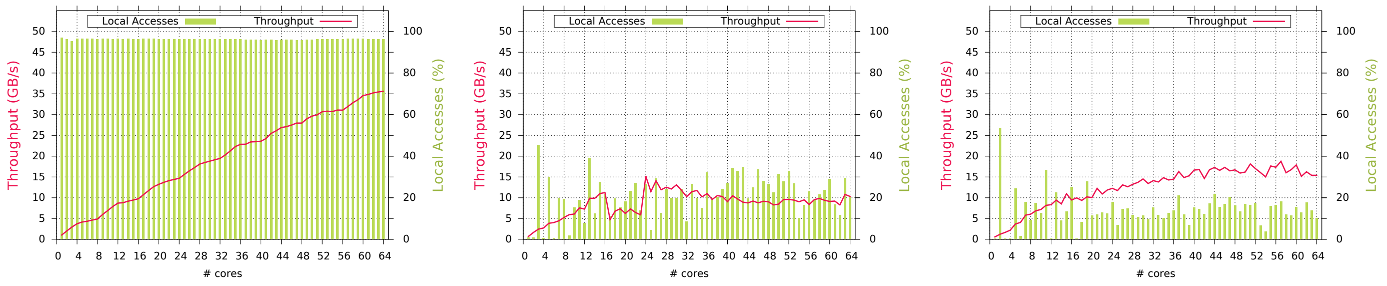

Fig. 6. *FIO* Writes for *Jericho* (left), Native 2.6.32 (middle) and Native 3.13 (right).

Table I summarizes the testbed used in our experimental evaluation. The testbed consists of 4 processor sockets and each of them contains 2 *NUMA* nodes. As a baseline for the evaluation of *Jericho* we use the native I/O stack of Linux kernel v.2.6.32 which is currently used in most enterprise Linux installations. For some of the experiments, we also use the v.3.13 Linux kernel, because it has incorporated several optimization related to *NUMA* memory organization. We use *FIO* [4] and *IOR* [10] microbenchmarks for our experimental analysis. These tools generate I/O workloads that expose issues related to CPU/memory affinity, in a controlled and reproducible manner. By using them we can stress the testbed up to its maximum attainable I/O performance limits.

Each graph presents two sets of data points for a specific system configuration. The first set, plotted as a curve, is the application throughput, as reported by the corresponding benchmark. The second set, plotted as vertical percentage bars, is the percentage of local memory accesses. We calculate the percentage of local memory accesses from samples collected with the *likwid* performance tool [11]. We collect samples of performance events related to the locality of memory accesses[2] during the execution of each experiment.

Next, we discuss the results obtained with the *FIO* and *IOR* microbenchmarks and finally we briefly discuss the impact of *NUMA*-related overheads in the presence of storage device accesses.

B. FIO *Microbenchmark*

FIO spawns a number of threads doing a particular type of I/O pattern as specified by the user. We present experimental results with up to 64 threads, with four different I/O patterns: sequential reads, sequential writes, random reads, and random writes. In all of the experiments in this subsection the I/O request size os 4KB.

[2] *UNC_CPU_REQUEST_TO_MEMORY_LOCAL_LOCAL_CPU_MEM* and *UNC_CPU_REQUEST_TO_MEMORY_LOCAL_REMOTE_CPU_MEM*

Each *Jericho* and Native run starts with a clean, freshly booted, system state. First we run a 64 thread *FIO* instance issuing sequential writes to quickly create our dataset, which consists of 64 files, of 2 GBytes each. All files fit in memory, for both the native I/O stack and *Jericho*. We then proceed with the measurement phase, issuing the various I/O patterns for a number of iterations, each lasting 30 seconds. We discard the results from the first iteration, to focus our study on the case where there is almost no device-level I/O access.

The initialization phase is subject to the system's task/memory placement policies. In the *Jericho* configuration, workload threads are constrained to run only on CPU cores of a single *NUMA* node, resulting in balanced arrangement of up to 8 threads per *NUMA* node. The native configuration results in a load-balanced arrangement, where the scheduler assigns no more than one thread per CPU core. The crucial difference is that the native configuration does not restrict thread migration across *NUMA* node boundaries. Thus, a thread migration during the initialization phase will spread file blocks across the memory of more than one *NUMA* nodes. Moreover, during the measurement phase even this 'fragmented' arrangement of workload threads and file blocks will not be retained. Further thread migrations will contribute to more memory accesses across different *NUMA* nodes. In contrast, with *Jericho* the placement of I/O-issuing threads will not change over time, resulting in better and more predictable performance.

Figure 5 shows *FIO* with sequential reads. With the unmodified 2.6.32 kernel (marked *Native 2.6.32* in the graphs) scaling is limited up to 16 threads, with almost no additional throughput with more cores. Using the unmodified 3.13 kernel (marked *Native 3.13* in the graphs) scaling is limited up to 40 threads also with no additional throughput with more threads. The more recent kernel shows improved I/O performance over the older one. Performance shows a strong correlation with the percentage of local memory accesses. *Jericho* outperforms both Native kernels, with almost linear scaling up to the maximum number of threads. With 64 threads, the improvement in

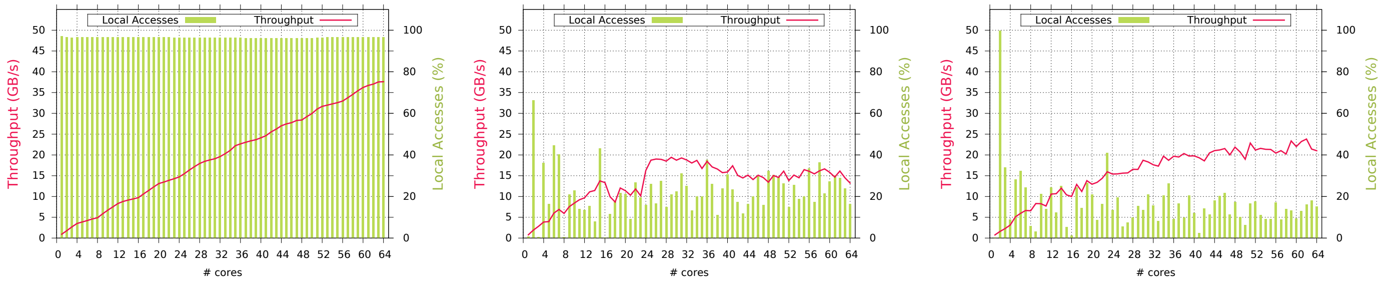

Fig. 8. *FIO* Random Reads for *Jericho* (left), Native 2.6.32 (middle) and Native 3.13 (right).

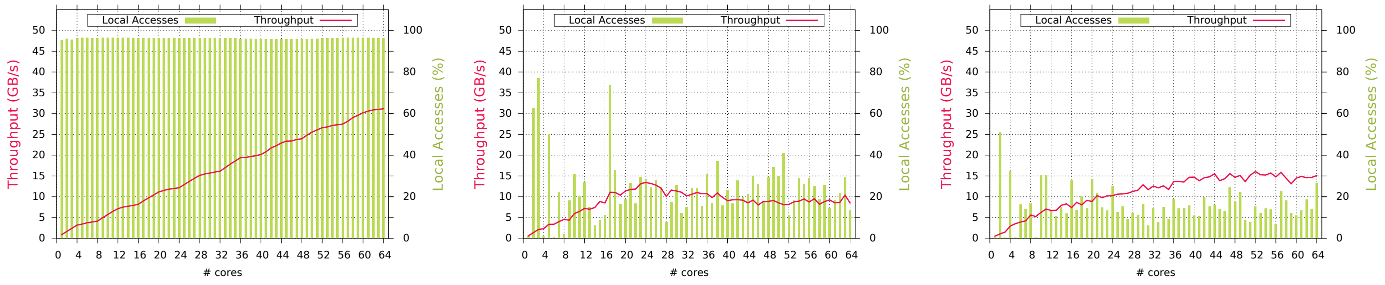

Fig. 9. *FIO* Random Writes for *Jericho* (left), Native 2.6.32 (middle) and Native 3.13 (right).

terms of throughput over *Native 2.6.32* is 2.9x and over *Native 3.13* is 1.8x.

Figure 6 shows *FIO* with sequential writes. We note a strong correlation of local accesses to throughput. With *Native 2.6.32*, the highest throughput of 15 GBytes/s is achieved with 24 threads, after which performance drops down to 10.5 GBytes/s using 64 threads. In the case of *Native 3.13*, the highest throughput of 19 GBytes/s is achieved with 58 threads, after which performance drops down to 15.5 GBytes/s GBytes/s using 64 threads. *Jericho* scales much better, reaching up to 35.5 GBytes/s with 64 threads. The improvement in terms of throughput over *Native 2.6.32* is 3.4x and over *Native 3.13* is 2.3x.

Figure 8 shows the results with *FIO* running the random read I/O pattern. We can see again the strong correlation of local accesses to throughput. In *Native 2.6.32* using 64 threads the percentage of local accesses is 16.5%, in *Native 3.13* it is 15%, and in *Jericho* it is 96.5%. The improved locality translates to a throughput improvement of *Jericho* over *Native 2.6.32* by 2.8x and over *Native 3.13* by 1.7x.

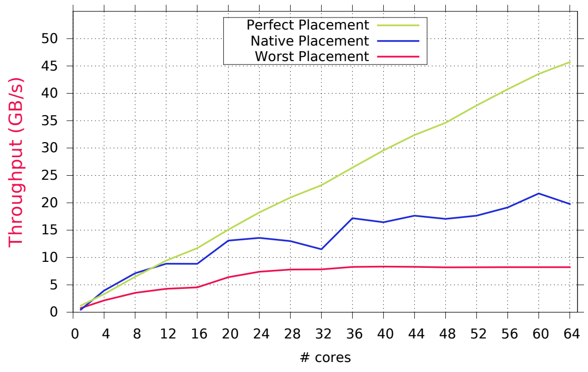

Fig. 10. Evaluation of Native, Best and Worst placement, with *FIO*.

Figure 10 illustrates that for *FIO* with the random read I/O pattern on the *Native 2.6.32* system configuration, thread placement has a large impact on application performance. The best case has 5.5x the throughput of the worst case, i.e. performance variations can be quite pronounced without constraining thread affinity. The typically occurring case (marked Native in this graph) has 2.4x the throughput of the worst case.

Figure 9 shows *FIO* with random writes. With 64 threads on *Native 2.6.32*, the percentage of local accesses is 13% and the throughput reaches 8.5 GBytes/s. With *Native 3.13* the percentage of local accesses is 26.5% and throughput reaches 15 GBytes/s. With *Jericho* 96% of accesses are in the local *NUMA* node. By minimizing the remote accesses we achieve 31 GByte/s, i.e. an improvement of throughput by 3.6x over *Native 2.6.32* and 2x over *Native 3.13*.

C. IOR *Microbenchmark*

IOR emulates a check-pointing application. In all experiments the I/O request size is 4KB. Figure 11 shows *IOR* read performance for *Jericho*, *Native 2.6.32*, and *Native 3.13*, respectively. Native runs show poor scaling, with very little performance improvement above 24 threads. We attribute this to the very low ratio of local-vs-remote accesses: 15.5% local accesses for *Native 2.6.32* and 22% local accesses for *Native 3.13*, with 64 threads. In contrast, *Jericho* results in 95% local accesses and a linear scaling curve, with higher throughput. With *Jericho* we achieve almost twice the throughput of Native, reaching 45 GBytes/s against 20 GBytes/s for *Native 2.6.32* and 17.5 GBytes/s for *Native 3.13*.

Figure 12 shows *IOR* write performance for *Jericho*, *Native 2.6.32*, and *Native 3.13*, respectively. With *Jericho* 95% of accesses are local, and throughput reaches 35 GBytes/s, when using 64 threads. For *Native 2.6.32* 86% of accesses are local (with 11 GBytes/s throughput), whereas with *Native 3.13* 95% of accesses are local (18.5 GBytes/s throughput). Despite showing the same percentage of local accesses, *Jericho* scales better than *Native 3.13*. We attribute this difference to

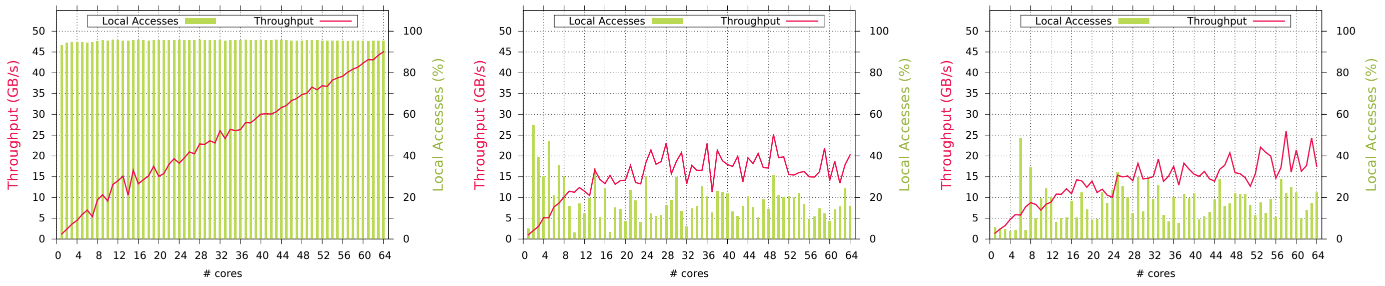

Fig. 11. *IOR* Reads for *Jericho* (left), Native 2.6.32 (middle) and Native 3.13 (right).

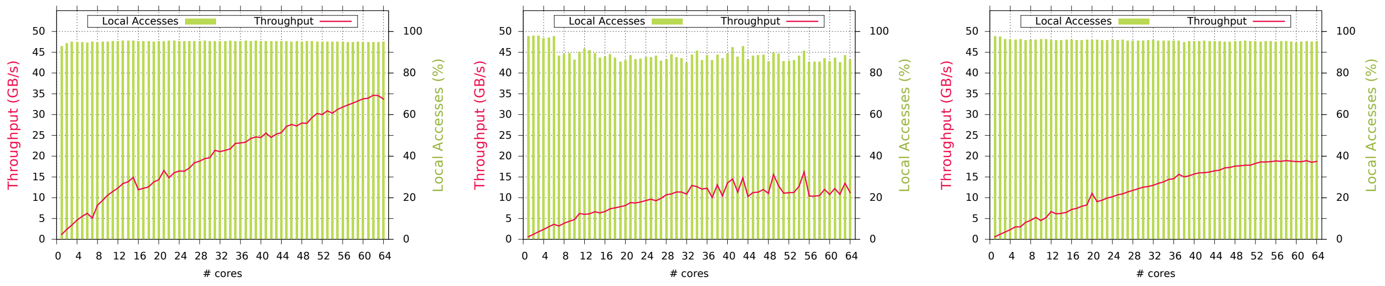

Fig. 12. *IOR* Writes for *Jericho* (left), Native 2.6.32 (middle) and Native 3.13 (right).

the distinct locks being used in each of the cache instances in *Jericho* (one per *NUMA* node). With 64 threads, the improvement of throughput with *Jericho* compared to *Native 2.6.32* is 218% and 89% compared to *Native 3.13*. Although *Native 3.13* clearly outperforms *Native 2.6.32* for the case of writes, we still find that the Native I/O stack does not scale with more than 44 threads.

D. Evaluation in the presence of device I/O

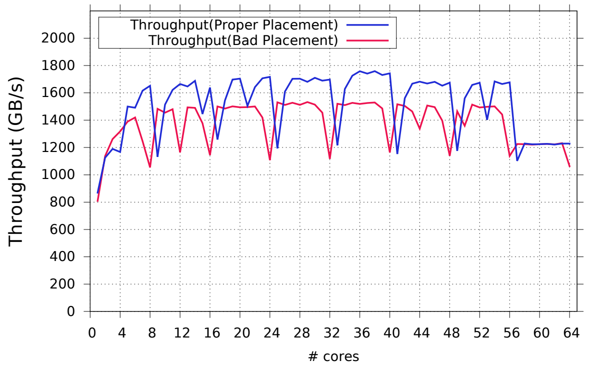

Fig. 13. *NUMA* effects in disk I/O for up to 64 cores, with concurrent *dd* instances.

Up to this point we have shown *FIO* and *IOR* results where the entire dataset fits in memory and there is no device I/O. In this subsection we experiment with device I/O, using concurrent `dd` instances using *direct* I/O, with each instance accessing a 1GB file. This allowed us to control memory an task placement using the *numactl* utility, using the *physcpubind* and *membind* parameters respectively. The I/O pattern is sequential reads, with a request size of 512KB. With this pattern, all of the storage devices (16 SSDs, on 2 controllers, RAID-0 with 32KB chunks, as per Table I) are being accessed concurrently. We compare two configurations, both using the *Native 2.6.32* I/O stack: one with ’Proper’ placement and one with ’Bad’ placement.

Figure 13 shows the effects of task and buffer placement under high I/O device activity. Thread and memory affinity are specified using the `physcpubind` and `membind` parameters of the `numactl` utility, respectively. For the *Proper* placement configuration, we ensure that threads are pinned (in groups of 8) to each of the 8 *NUMA* nodes of our testbed in a manner that avoid cross-node memory accesses. In contrast, for the *Bad* placement configuration we ensure that all memory references by all threads are remote.

We observe that even when all I/O requests are served from storage devices rather than from I/O buffers in memory *NUMA* effects are still clearly visible. In both configurations, there is a noticeable drop in I/O throughput with every 8 threads, which is exactly the number of cores in each of the *NUMA* nodes. The *Proper* configuration always outperforms the *Bad* configuration: up to 60% better, 11% better on average. With the increasing popularity of fast devices on multicore servers, a 11% difference can translate to a performance loss on the order of GBytes/s. When we reach 56 and more threads, I/O throughput drops dramatically to the roughly the same level as for 4 threads. We believe that we reach a saturation point of either an interconnect, memory controller or both. This experiment, highlighting two extremes of the possible thread placements, is indicative of the performance variability that is often observed with several I/O workloads that do not explicitly control affinity.

IV. Related Work

Due to the performance discrepancy between workloads executing on *NUMA* servers depending on thread and data affinity, researchers and industry have considered various optimizations, mostly at the application level. Prior work has mostly focused on the application level, finding ways to achieve mostly local memory accesses and to minimize the importance of initial thread and data placement. However, I/O path optimizations related to *NUMA* awareness have not received as much attention. With increasing data-sets for applications and the trend towards manycore servers with more pronounced non-uniformity in their memory access times, we believe that locality management for I/O-related memory is becoming a critical problem.

On *NUMA* systems, it is often the case that performance degradation is not primarily due to increased memory access latency. Task migrations by the scheduler may increase the volume and frequency of remote memory accesses, thereby increasing contention for the memory controllers and the system interconnect. Prior work has produced algorithms and heuristics for tracking the usage patterns of memory pages and migrating threads and/or pages to improve performance. A common technique is to maintain per-page access statistics. For the case of I/O-intensive workloads, we have found that we can derive the essential information for good locality at the filesystem level, when an I/O access is being issued. Therefore, we do not maintain per-page run-time state, but only check the affinity masks of I/O-issuing threads.

With the *AutoNUMA* [12] Linux kernel patch, per-page statistics are used for thread migration decisions. Threads migrate toward the *NUMA* nodes holding the majority of their accessed pages. All pages are periodically unmapped from process address spaces, so that the next access will trigger a page fault. The custom page fault handler of *AutoNUMA* migrates such unmapped pages to the requesting *NUMA* node. *AutoNUMA* attempts to incrementally improve the affinity of threads and pages, based on the observed access pattern. However, *AutoNUMA* may incur excessive overhead by triggering a long sequence of migrations with memory-intensive workloads. Moreover, there is continuously overhead from the scanning thread that periodically invalidates page-table entries and the resulting page faults. For I/O-intensive workloads, the main limitation is that *AutoNUMA* does not track the I/O buffer pages. When application-level threads issue I/O requests, system calls copy data to/from kernel-space I/O buffers in the page cache. Only the application-level pages are tracked and therefore affected by the control logic of *AutoNUMA*.

In contrast, *Jericho* by design does not migrate I/O buffer pages, keeping these pages at all times at the *NUMA* node that corresponds to the corresponding slice of our filesystem. With *Jericho* there is no need to gather per-page statistics about locality. I/O-issuing tasks are properly placed by design, guaranteed to do only local accesses. Using our filesystem layer we migrate tasks as needed, based on readily available information about the assignment of files to *JeriCache* slices. To fully control the location of I/O buffer pages, we bypass the page cache by having our custom filesystem issue I/O requests to *JeriCache* slices. These design changes are necessary for I/O intensive workloads. Unlike explicitly allocated memory, e.g. for application-level data structures, page cache buffers are not fully 'owned' by a single process. On the contrary it is common to have I/O-issuing threads fetch data for processing threads. For instance, this is the case with the qemu machine emulator and the PostgreSQL database server. As discussed earlier, in such scenarios page migration is not feasible, as kernel-space I/O buffers are not directly linked to the application-level threads.

Contention-aware scheduling [13] characterizes pairs of threads from the perspective of how much interference they would experience if they were co-located, and then separates threads that are likely to interfere. However, on *NUMA* systems this approach needs to be augmented with the capability to migrate memory pages across *NUMA* nodes and also with criteria for avoiding unnecessary thread migrations [14]. Our work focuses on the affinity of I/O-issuing threads, where this prior work is not directly applicable. As the affinity controller is implemented in the user-space, it cannot identify and then migrate re-usable I/O-related buffers across *NUMA* nodes Its scope is limited to the application-owned private memory pages. The same limitation applies to the *Carrefour* system [9]. The *Carrefour* algorithm aims to minimize interconnect contention, via a combination of techniques: page replication, page migration, and task clustering. *Carrefour* makes page placement decisions based on samples from hardware performance counters and adjusts the placement of threads based on the observed contention level. As our approach focuses on locality we avoid replication and migration of pages. A limitation of our approach is that we are not utilizing memory space and throughput from than one *NUMA* nodes. However, this limitation does not impact our target workloads - i.e. sets of threads operating on independent file-sets. Our approach guarantees that the I/O-issuing threads are 'pulled' towards the *NUMA* nodes that host their corresponding data, thus eliminating unnecessary, expensive remote accesses.

The *SSDFA* user-level filesystem [15] is an attempt to improve the scalability of the I/O stack on multicore servers for fast storage devices. *SSDFA* is implemented on top of a native VFS filesystem. Unlike stackable filesystems [16], [17] *SSDFA* does not 'hook' into the Linux VFS, using instead direct-path mechanisms for I/O buffers and device access. Filesystem requests are served from a custom-build page cache. Similarly to our work, *SSDFA* pins threads and device interrupt handlers to specific cores, taking into consideration affinity as well as load balancing.

SSDFA relies on one I/O handling thread per storage device, whereas in *Jericho* we have one thread for each cache instance, allowing more configuration options. We can have one cache instance per device, as in *SSDFA*, or we can have one cache instance per CPU core, to further reduce contention in the I/O issue path. Moreover, an important optimization in *Jericho* is the inlining technique to avoid context switches when the I/O-issuing thread runs on a CPU with proper affinity. This optimization has significant impact in the case of fast devices, where the overhead in issuing I/O requests becomes a noticeable part of overall I/O latency.

Our design bears similarities to the recent multi-queue block layer design [18], which improves device I/O scalability by modifying the I/O-issue path, both at the software and hardware levels. At the software level, the multi-queue block layer increases the number of the I/O queues; instead of a single queue as in the standard Linux block layer, the new design allows having per-core or per-socket *software staging queues*), thereby eliminating contention during I/O issue. This flexibility is matched at the hardware controller level, with the possibility to utilize multiple independent *hardware dispatch queues*, responsible for actually issuing the I/O requests to the underlying fast storage devices, e.g. to PCIe SSD devices. In our work, *JeriFS* slices and the *pipeline* threads provide similar scalability benefits. The threads operating on top of a

JeriFS slice and its corresponding cache are issuing their I/O requests independently of each other, without going through a single serialization point. Moreover, in our work we explicitly change the CPU affinity mask of threads to reduce the impact of *NUMA*-related effects. In *Jericho*, I/O requests are actually issued to the underlying storage devices by the per-cache *evictor* and *pipeline* threads (to serve writes and read misses, respectively). Our design is thus able to make use of multiple independent *hardware dispatch queues* if this capability is available in the storage devices underlying the cache instances.

Jericho can be applied for VM workloads, but details of the virtualization technology need to be considered, specifically the question of which thread actually issues the I/O requests to the underlying storage devices. Additional complexity arises in the case where the hypervisor uses 'helper' threads to handle I/O on behalf of VMs. This is the case with kvm [19] which relies on I/O threads dynamically spawned by the qemu machine emulator. *Jericho* sets the CPU affinity mask not only for the issuing task, but also for the parent task. In this scenario, the I/O-issuing threads are not the ones actually producing or consuming the data being transferred to/from the storage devices. In this scenario, we would need to additionally set the affinity masks of the 'vCPU' threads (i.e. the threads that execute the VM's instruction stream, one per emulated core). *Jericho* would still offer performance benefits, by 'pulling' the I/O helper threads of the VM towards the appropriate *NUMA* node, but this benefit would be short-lived, as the I/O helper threads only have a transient lifetime. A useful extension to our work would be to identify the 'vCPU' threads and change their affinity masks as well. In the case where the hypervisor supports device assignment [20], *Jericho* will simply change the CPU affinity mask of the I/O issuing threads to eliminate remote memory accesses. In this manner, *Jericho* achieves the benefits of affinity-aware thread placement, exactly as in the case of native (host-level) workloads.

Jericho has been designed with server workloads running on a single *NUMA* multicore server. With the rising popularity of analytics workloads using *Hadoop*, a useful extension of our work would be to add support for the co-ordinated allocation and management of caches on multiple nodes. This would be similar to the PACman system [21]. A major finding from this work is that optimizing the I/O phase of jobs results in a huge improvement of job completion times. *PACman* follows an all-or-nothing policy for evictions, trying to keep the entire dataset of each job and evict incomplete datasets. We could provide an additional optimization for the I/O phase of *Hadoop* workloads with our thread placement technique.

V. Conclusions

In this work we have identified scalability limitations of the Linux kernel due to *NUMA* effects, using targeted intensive tests of the common I/O path. We present an evaluation of *NUMA* effects with our *Jericho* I/O stack that supports slices. We demonstrate significantly improved scalability, for both I/O throughput-intensive and IOPS-intensive tests. Most of these limitations are not observed at relatively low core counts (8-12), which is the currently common core count for servers, but become severe with more than 24 cores. With 64 cores, our *Jericho* I/O stack improves sequential read I/O throughput by 2.9x over the baseline system and sequential write I/O throughput by 3.4x. We demonstrate similar improvements for random IOPS performance. Overall, our approach and results will be more useful and relevant with upcoming larger-scale *NUMA* server platforms, with more pronounced non-uniformity in remote memory access times and cross-core synchronization overheads.

In this paper we have not explored dynamic policies for thread placement in our *Jericho* I/O stack. In our current design, we use static configurations for the allocation of system memory. This can lead to under-utilization of the memory in nodes that have lower load. We are currently considering an extension of our design to allow a local *NUMA* node with high load to utilize free memory in remote *NUMA* nodes with lighter load. This extension would effectively increase the effective size of our *JeriCache* caches by allowing 'spill-over' of I/O buffers to remote *NUMA* nodes with low memory.

Acknowledgments

We thankfully acknowledge the support of the European Commission under the 7th Framework Programs through the EuroServer (FP7-ICT-610456), IOLANES (FP7-ICT-248615), and HiPEAC3 (FP7-ICT-287759) projects. We would also like thank the following colleagues at FORTH for their assistance: Konstantinos Chasapis, Yannis Klonatos, Spyros Papageorgiou, and Markos Foundoulakis for useful comments and discussion.

References

[1] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman, "Numa-aware algorithms: the case of data shuffling." in *CIDR*, 2013.

[2] L. Bergstrom, "Measuring numa effects with the stream benchmark," *arXiv preprint arXiv:1103.3225*, 2011.

[3] C. Lameter, "Numa (non-uniform memory access): An overview," *ACM Queue*, vol. 11, no. 7, p. 40, 2013.

[4] J. Axboe, "Flexible i/o tester." [Online]. Available: https://github.com/axboe

[5] Z. Shao, J. H. Reppy, and A. W. Appel, "Unrolling lists," in *Proceedings of the 1994 ACM conference on LISP and functional programming*, vol. 7, no. 3. ACM, 1994, pp. 185–195.

[6] P. E. McKenney, D. Sarma, and M. Soni, "Scaling dcache with rcu," *Linux Journal*, vol. 2004, no. 117, Jan. 2004. [Online]. Available: http://www.linuxjournal.com/article/7124

[7] TYAN, "Ft48-b8812_v1.2a service engineer's manual," 2012. [Online]. Available: http://www.tyan.com

[8] JEDEC, "DDR3 SDRAM STANDARD," http://www.jedec.org/standards-documents/docs/jesd-79-3d.

[9] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on numa systems," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2013, pp. 381–394.

[10] W. Loewe and T. McLarty, "Parallel file systems benchmark," 2003. [Online]. Available: https://github.com/chaos/ior

[11] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 207–216.

[12] J. Corbet, "Autonuma: The other approach to numa scheduling," 2012. [Online]. Available: https://lwn.net/Articles/488709/

[13] S. Blagodurov, S. Zhuravlev, and A. Fedorova, "Contention-aware scheduling on multicore systems," *ACM Transactions on Computing Systems*, vol. 28, pp. 8–45, 2010.

[14] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, “A case for numa-aware contention management on multicore systems,” in *Proceedings of the 2011 USENIX Technical Conference*, 2011, pp. 1–16.

[15] D. Zheng, R. Burns, and A. S. Szalay, “Toward millions of file system iops on low-cost, commodity hardware,” in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2013, pp. 69:1–69:12.

[16] E. Zadok, I. Badulescu, and A. Shender, “Extending file systems using stackable templates,” in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATC ’99. Berkeley, CA, USA: USENIX Association, 1999, pp. 57–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268708.1268713

[17] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair, “Versatility and unix semantics in namespace unification,” *Trans. Storage*, vol. 2, no. 1, pp. 74–105, Feb. 2006. [Online]. Available: http://doi.acm.org/10.1145/1138041.1138045

[18] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, “Linux block io: Introducing multi-queue ssd access on multi-core systems,” in *Proceedings of the 6th SYSTOR International Systems and Storage Conference*. ACM, 2013, pp. 22:1–22:10.

[19] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: The Linux Virtual Machine Monitor,” in *Proceedings of the Ottawa Linux Symposium*, 2007.

[20] J. Liu, W. Huang, B. Abali, and D. K. Panda, “High performance vmm-bypass i/o in virtual machines,” in *Proceedings of the annual conference on USENIX ’06 Annual Technical Conference*. USENIX Association, 2006, pp. 29–42.

[21] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, “Pacman: Coordinated memory caching for parallel jobs,” in *Proceedings of the USENIX NSDI Conference*, 2012, pp. 267–280.