2012 IEEE 26th International Parallel and Distributed Processing Symposium

A Source-aware Interrupt Scheduling for Modern Parallel I/O Systems

Hongbo Zou , Xian-He Sun , Siyuan Ma , and Xi Duan

Department of Computer Science
Illinois Institute of Technology
Chicago, IL, USA 60616
Email: {zouhongbao@gmail.com, sun@iit.edu, sma9@iit.edu, xduan@iit.edu}

***Abstract*—Recent technological advances are putting increased pressure on CPU scheduling. On one hand, processors have more cores. On the other hand, I/O systems have become more complex. Intensive research has been conducted on multi/many-core scheduling; however, most of the studies follow the conventional approach and focus on the utilization and load balance of the cores. In this study, we focus on increasing data locality by bringing source information from I/O into the core interrupt scheduling process. The premise is to group interrupts associated for the same I/O request together on the same core, and prove that data locality is more important than core utilization for many applications. Based on this idea, a *source-aware* affinity interrupt-scheduling scheme is introduced and a prototype system, SAIs, is implemented. Experiment results show that SAIs is feasible and promising; bandwidth shows a 23.57% improvement in a 3-Gigabit NIC environment and in the optimal case without the NIC bottleneck, the bandwidth improvement increases to 53.23%.**

Keywords-interrupt scheduling; Source-aware; Parallel I/O;

I. INTRODUCTION

The peak throughput of processors has been improving at a phenomenal rate of 50% to 100% per year during the last three decades [23]. Since 2004, the emergence of multi/many-core [7] architectures has changed the landscape of computing [41] and further accelerated these improvements. Compared to processor performance improvements, data access performance (both latency and bandwidth) has improved a snail's pace. The memory speed has only increased by roughly 9% each year. Additionally disk throughput has only doubled over the past two decades [23]. This performance gap between processors and I/O, known as the I/O-wall problem, is predicted to continually expand in the foresee future [9][23]. This gap has become the critical issue limiting the sustained performance of parallel applications. Parallel I/O techniques can help relieve this problem by creating multiple data paths between processors and I/O, and is considered by many as the primary solution to overcome the I/O-wall problem. However, parallel I/O further complicates the already complicated multi-core scheduling and simply, ignoring the fact of parallelism in I/O systems is not an appropriate approach. In order to reap the full benefits of a parallel I/O on multi-core system, a reexamination of processor scheduling is required. Scheduling schemes must be refined to match the complications of parallelism and adjust to data intensive applications. In this study, we undertake a reexamination of interrupt scheduling for parallel I/O on multi/many-core systems.

Conventionally, interrupt scheduling distributes interrupts to lightly loaded cores with consideration of only core utilization, fairness and power consumption [20]. Consequently, the core handling an interrupt is likely not the core consuming the requested data (i.e. the core running the application processes did not issue the I/O request). This results in a transfer of data between the two cores local caches.

This problem is exacerbated further in parallel I/O. In a Parallel File Systems (PFS), multiple server nodes are employed to serve one I/O request in order to improve the I/O bandwidth. This increased I/O bandwidth, consequently, incurs more Network Interface Card (NIC) interrupts on the client side. In a conventional load balanced system, interrupts are evenly distributed to cores for handling, whereas these interrupts may come from the same data request. This inevitably causes data movement among caches. Under the conventional scheduling approach, parallel I/O leads to even more frequent data movement on the client side due to the increased bandwidth of the system. In addition, when the bandwidth of parallel I/O increases, returned data often have to be swapped out of the L1/L2 cache. This in turn causes more memory access. The I/O performance will be more penalized due to this memory access.

Therefore, core-utilization based interrupt scheduling is not an appropriate approach for interrupt handling in parallel I/O systems. To improve this algorithm, interrupts for the same I/O request, and for concurrent parallel I/O requests, can be grouped onto the same core to increase data locality. In this study, a novel source-aware interrupt scheduling scheme is proposed to optimize I/O performance. This scheduling scheme is based on the source-aware idea, which correlates I/O interrupt handlers to their data consuming process. In source-aware nomenclature, the original I/O request is called the source, and all the interrupts serving for the same source are called peer interrupts. In this design and implementation, a single core is chosen to be the core where the data request process is running. Although the data request process could be migrated to another core while it is blocked upon an I/O operation, it is rare to see such a migration happen during the I/O blocking, especially in an I/O intensive system. For this reason, our scheme schedules the I/O interrupts for the same source process onto the core which runs the corresponding I/O request. Due to the scarcity of process migration as mentioned above, the scheme generally avoids the data movement among the cores. Please note that source-aware is distinct from processor affinity [31], which is a data sharing correlation between the core and the processes, if the remnant cache data/states in the core could be used by the processes/threads in the future. Then processor affinity directly describes the candidate cores to which interrupts could be delivered and executed.

Several challenges exist in the source-aware scheduling scheme, including:

- How to identify which process an I/O interrupt belongs to;
- How to inform the I/O interrupt scheduler the location of the process issuing the requests;
- How to provide a light-weight solution upon current multi-core system;

These challenges have motivated this research. The main contribution of this paper is four-fold.

- Conducted a quantitative study to reveal performance issues of interrupt scheduling in parallel I/O;
- Designed a source-aware interrupt scheduling scheme for parallel I/O;
- Implemented a source-aware I/O interrupt scheduler prototype, named SAIs;
- Performed experimental testing to verify the feasibility and effectiveness of SAIs;

Several interrupt scheduling schemes have been proposed and implemented recently to guide the Advanced Programmable Interrupt Controller (APIC) [3][18] in optimized interrupt scheduling [3][17][18][20][25]. However, these interrupt scheduling strategies are mainly focusing on improving cores utilization, rather than data locality. Therefore, the *source-aware* I/O interrupt scheduling is a complement to these existing solutions.

Because there is not a data locality issue associated with interrupt scheduling in parallel I/O write operations, our study focuses on parallel I/O read. The rest of this paper is organized as follows: Section 2 surveys APIC interrupt management mechanism and I/O Interrupt scheduling on Multi-core. Section 3 introduces the concept of *source-aware* interrupt scheduling and its associated quantitative analysis. The general *source-aware* interrupt scheduling scheme and one specific implementation are proposed in Section 4. Section 5 and 6 present and analyze the experimental and simulation results, respectively. In Section 7, we discuss related works. Finally, Section 8 concludes the paper.

II. BACKGROUND

A. APIC Mechanism

In modern computer system, the Advanced Programmable Interrupt Controller (APIC) provides interrupt support on X86 architecture processors, such as Intel 64 and AMD 64. There are two components in the X86 APIC systems, the Local APIC and the I/O APIC [3][18]. Typically, each core has a Local APIC, and the system has a single I/O APIC component shared by the multiple devices connected on a peripheral bus. The main function of the I/O APIC is to receive external interrupts events from its associated I/O devices (e.g. NIC, Hard disk etc.) and route them to one or more Local APICs as interrupt messages. The Local APIC primarily accepts interrupts message sent from I/O APIC and delivers them to the associated core for further handling. In general, I/O APIC routes the interrupts to the local APICs based on the interrupt redirection table. This table identifies which cores could handle the interrupts for the specific device [33]. The I/O APIC extracts the available cores information from the table and puts it into the interrupt message as the destination address. The actual handling of I/O interrupts takes place in the softirq interrupt thread [12], which is mostly performed on the core that received the interrupt. To maximize multi-core utilization and power consumption, some interrupt scheduling schemes, such as irqbalance [20], have been developed and applied to dynamically change interrupt scheduling policy. However, this balance scheduling may harm the parallel I/O when interrupts are scattered to the multiple lightly loaded cores rather than the cores requesting data [10][12]. When the interrupt handling and the application are executed on different cores, the system overhead will increase as more I/O data access would be required due to the increased amount of inter-core data movements.

B. I/O Interrupt Scheduling on Multi-core

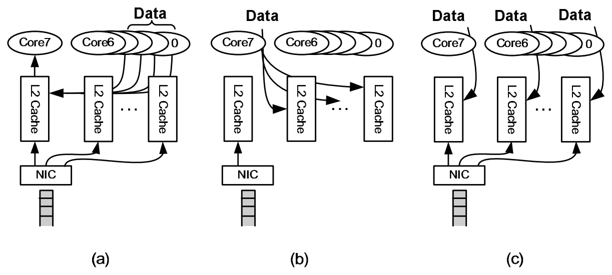

Figure 1. The data location with Different I/O Interrupt Scheduling.

In general, interrupt scheduling schemes can be classified into three types (shown in Figure 1) with or without the source-aware feature [16]. In Figure 1, (a) describes round-robin modes, in which the incoming interrupts are handled by cores in turn. This mode is good for core utilization or load balancing, but it damages the peer interrupt source-aware described earlier. This mode is the default interrupt scheduling configuration for the Linux with Intel processor. (b) shows dedicated modes, in which there is a special core to handle all the incoming interrupts. This mode also ignores source-aware of peer interrupts. For example, with an AMD processor, the Linux default interrupt scheduling is configured to operate in lowest priority mode, this causes the incoming interrupts to be handled only on core 7. The schemes described in (a) and (b) make it difficult to maintain source-aware on current multi-core systems. Scheme (c) illustrates the proposed source-aware interrupt scheduling mode, where source-aware scheduling arranges interrupts from the same application process onto the same target core. This mode guarantees that the interrupted data are processed and consumed on the same cores, which improves the cache affectivity and reduces inter-core data movement.

III. SOURCE-AWARE INTERRUPT SCHEDULING

To increase data locality in parallel I/O, a source-aware interrupt scheduling scheme (called source-aware scheme) is proposed. It instructs the I/O APIC to deliver the interrupt requests to the core where the data request application process is running (named data consuming core). The underlying assumption is every core has a dedicated private cache, which is generally true with current and foreseeable multi-core microprocessors. Figure 2 shows a simple case to explain the basic idea of the source-aware scheme. In this case, there is one I/O client and *n* I/O server nodes (I/O server *1* to *n*). On the I/O client, there are *n* computing cores (Core *1* to Core *n*) sharing one NIC and one I/O APIC. The application processes (APs, from AP *A* to AP *N*) are executing on the *n* cores concurrently. If AP *A*, AP *B*, and AP *C* request the data block from the PFS concurrently, the PFS will return the data block *A*, block *B*, and block *C* to the requested AP, respectively.

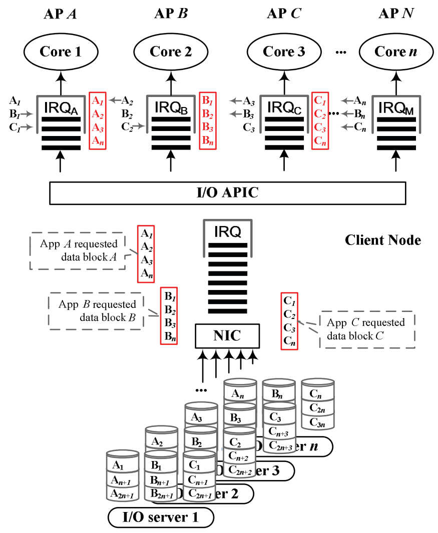

Figure 2. Source-aware Interrupt Scheduling Design.

On the I/O client, if the received data strips sequence is (A_1, A_2, A_3, …, A_n), (B_1, B_2, B_3, …, B_n), and (C_1, C_2, C_3, …, C_n), there are two ways for interrupts to be delivered. By default, the I/O APIC is instructed to use a balance scheme, hence the interrupts are spread to all the cores based on their load information. In this case, data strips within the same data request could be handled on totally different cores. This fact will lead to inevitable data migration from the core handling the interrupt, to the core consuming the data. In Figure 2, the small arrow on the side of data strips shows the data migration path among the cores. These migrations are mitigated if the APIC adopts the source-aware scheme. Here the data strip will be handled directly by the core that hosts the AP and consumes the data. The red frame at the side of the core stands for the result of the source-aware scheduling.

With the addition of the source-aware concept, there are four possible scheduling polices: (i) select the core that generated the request, (ii) select the core which runs the process that produced the I/O request (maybe different than (i) if a rescheduling may occurred during I/O blocking), (iii) select the least-loaded core, or (iv) select a specific dedicated I/O core. The last two policies are the conventional (source-unaware) scheduling approaches. The second is a source-aware policy should be more efficient than the first. However, since the process migration rarely happens during a blocking I/O, the expected performance difference between the first two polices is trivial. As a topic for future study, the four presented policies could be integrated and the second source-aware policy could be implemented.

To clarify the advantage of source-aware scheduling scheme, a quantitative analysis is included as following.

A. Assumptions for Analysis

In a general PFS, there are N_C I/O client cores and N_S I/O server nodes. A data Block *X* will be split into N_S data strips (X_1, X_2, …, XN_S) over the N_S I/O server nodes. When an application requests Block *X* , the I/O server *i* needs to return the data strips X_i. For simplicity, we assume N_C can exactly divide N_S, and all data strip X_i have the same size. Hence we can use *P to represent* the processing time of one data strip, and *M to represent* its migration time from one processor core to another. Experiments in latter sections show that data migration is much more expensive than interrupt handling with a high speed multi-core processor. Therefore, we can deem $M >> P$. For further analysis, let T_p be the total processing time of data strips on each core; T_M be the total strip migration time between the cores; T_R be the rest time spent on network and server side. Both T_p and T_M could be calculated in terms of *P* and *M.*

Notice that interrupt scheduling can affect T_p and T_M, but have no influence over T_R, because T_R is a variable only related to the time of network transmission and server response. It can then be estimated that T_R is equal under different interrupt scheduling policy. Additionally, data strip processing and data strip migration can happen simultaneously. This overlapped part, referred as T_O, is under the impact of many factors besides the scheduling policy, hence hard to evaluate. However, as the processing and migration time becomes shorter, it is less probable that they will happen concurrently. So if our interest lays only in the interrupt scheduling, as what appear in this paper, we can assume that T_O is proportional to $Min(T_{p,}\ T_M)$.

By the above assumptions, the total time of an I/O request can be decomposed into four parts by equation (1). For various interrupt scheduling policy, T_R is a constant; T_p and T_M are variables; and T_O is proportional to the minimum of T_p and T_M.

$$T = T_R + T_P + T_M - T_O \qquad (1)$$

It is still difficult to express T_p with *P,* partly due to the possible concurrency of multiple strip processing. With N_C cores and N_S strips, T_p could be at most $N_S \times P$ if all the strips are handled by one core, or at least $P \times (N_S/N_C)$ if strip processing take the advantage of all N_C cores. T_M is much easier to calculate. In most CPU design, only one strip migration can happen at any time. So we can obtain T_M by:

$$T_M = M \times \# migration \quad (2)$$

In the rest of this section, we are going to compare two different interrupt scheduling policies, balanced and *source-aware*. For simplicity, assume that the number of I/O servers is the multiple of the number of cores, hence $N_S = \alpha \times N_C$, where α is a positive integer.

B. Single I/O Request

For balanced scheduling, interrupt will be distributed evenly across different cores. So if all the interrupts are invoked closely, we will have $T_p = P \times (N_S/N_C) = P \times \alpha$. But it is too optimistic. The time gap between each interrupt could be large. So $T_p >= P \times \alpha$. The faster the network and storage server is; the closer T_p to its bottom boundary is.

The disadvantage of balanced scheduling is the number of incurred strip migration. Since at the end of a request, all the strips will be moved to a single core, the cost of load balance is high. The migration cost is $T_M = M \times (N_S \times (N_C - 1)/N_C)$. Meanwhile, notice that $T_O = Min(T_P, T_M)$ and $M >> P$, we can deduce $T_O <= T_p$.

According to above analysis, inequality (3) evaluates the efficiency of balanced scheduling.

$$T_{Balance} \geq T_R + M \times \alpha \times (N_C - 1) \quad (3)$$

For *source-aware* interrupt scheduling, it has a higher strip handling cost. By processing all the strips on only one core, we have $T_p = P \times N_S$. On the other hand, there is no strip migration cost, since all the strips are scheduled to the same core at the very beginning. Consequently, the total time can be expressed as:

$$T_{Source-aware} = T_R + P \times N_S \quad (4)$$

Though N_S is slightly larger than $(N_C - 1) \times \alpha$, we can still draw the conclusion that $T_{Balanced} - T_R >> T_{Source-aware} - T_R$ because $M >> P$.

C. Multiple I/O Request

Let N_R be the number of I/O requests submitted by the client. Even in a light loaded system, N_R is generally larger than N_S.

When several I/O requests on one client are divided into smaller requests on the servers, there is no migration cost for *source-aware* scheduling, and T_p increases to $P \times N_S \times N_R$. So, the evaluation becomes:

$$T_{Source-aware} = T_R + P \times N_S \times N_R \quad (5)$$

Similar to the analysis of source-aware scheduling, the variable part of balanced scheduling is increased by a factor of N_R, where N_R is given by inequality (6). For this reason, we still have $T_{Balanced} - T_R >> T_{Source-aware} - T_R$.

$$T_{Balance} \geq T_R + M \times \alpha \times (N_C - 1) \times N_R \quad (6)$$

Note that the time difference of the two methods is now subject to N_R, the number of I/O requests. Since $N_S = \alpha \times N_C$, the difference between N_S and $\alpha \times (N_C - 1)$ is negligible. The time difference is proportional to three factors: the number of servers (N_S), the extra time consumption of data strip migration over strip processing (*M-P*), and the number of requests (N_R). Factor (*M-P*) is entirely determined by hardware, hence fails to gain our interests. So the other two factors are those affecting the potential performance improvements of the source-aware scheduling.

Although it seems sensible to enlarge the performance gain by simply increasing N_S, an implicit connection between these two factors invalidates such behavior. Let $Size_{req}$ represents the size of an I/O request, and then we can build a coarse relationship between N_R and N_S in (7). When the client bandwidth is large enough, an increase of N_S allows the client to exploit more benefits from source-aware scheduling. When the bandwidth becomes a bottleneck, increasing N_S implies the decrease of N_R in (7), which will in turn reduce the advantage of source-aware scheduling.

$$N_R \times N_S \times Size_{req} \leq Bandwidth_{Client} \quad (7)$$

D. Multiple Programs on One Client

Now consider the case when more than one program runs on the client. Assume the number of programs is N_P, and N_R/N_P is the number of requests issued by a single program. The analysis of balanced scheduling can largely follow the inequality (6). While *source-aware* scheduling is different. Depending on the relation of N_P and N_C, there are two different scenarios.

1) $N_C >= N_P$

For this case, only N_P cores will get used during the interrupt handling. If the workload is heavy, interrupts will be handled concurrently. So T_p could be as low as $P \times N_S \times N_R/N_P$. It is implying a shorter time cost compared to (5):

$$T_R + P \times N_S \times N_R \geq T_{Source-aware} \geq T_R + P \times N_S \times N_R / N_P \quad (8)$$

2) $N_C < N_P$

In this situation, all the cores will be busy processing interrupts from different programs. For simplicity, it is assumed N_P is a multiple of N_C. Each core will hold N_P / N_C programs for execution. So the lower boundary of T_p becomes $P \times N_S \times N_R / N_C$ for both scheduling methods. The balanced scheduling can almost always reach this boundary, while the T_p of *source-aware* scheduling varies according to the workload. At the worst case, where no two cores are handling interrupts simultaneously, T_p can be as large as $P \times N_S \times N_R$.

$$T_{Balance} - T_{Source-aware} \geq (N_C - 1) \times N_R \times \alpha \times (M - P) \quad (9)$$

As to the strip migration cost of balanced scheduling, the analysis in section 3.3 still works. Additionally, the number of programs doesn't affect the total migration cost, because all the requests from one program will eventually be transferred to the same core. Therefore, we can express the performance difference of two scheduling as inequality (9). Accordingly, as $M >> P$, the source-aware scheduling will still have better performance.

The above analysis gives us an intuition of how our approach affects the overall performance. As we can see, the source-aware method generally shortens the response time. And it is most effective under the heavy workload scenario where a client runs N_C programs simultaneously. Its effectiveness, however, does rely on the proportion of T_R in the whole I/O request handling. If network peak bandwidth is a limitation, more efficient interrupt scheduling will not make much of a difference on the overall performance. In fact, more efficient interrupt handling can move the performance bottleneck from interrupt handling to network. For this reason, to explore the full potential of source-aware interrupt handling, we adopt a combined implementation and simulation testing in the experiment section.

IV. SAIs Interrupt Scheduler

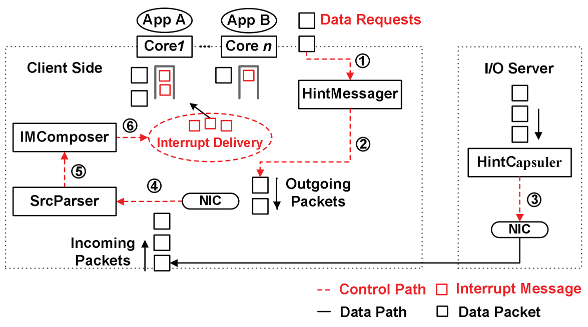

Figure 3. SAIs System Architecture.

With the above quantitative analysis, we propose a novel source-aware interrupt scheduling (SAIs) for parallel I/O interrupts in this section. A general SAIs system design under Parallel Virtual File System (PVFS) [29] is presented herein. PVFS is a choice of implementation. The design can be extended to other parallel file systems as well.

A. System Design

SAIs dynamically directs incoming interrupts to the affinitive core based on the affinitive core ID (*aff_core_id*) information. The *aff_core_id* is the identifier of the core that the application is running on and the I/O request has been sent out from. The *aff_core_id* could be put into each I/O request and guide interrupt scheduling when the data returns. SAIs consists of three core components on the client side: HintMessager, SrcParser, and IMComposer as shown in Figure 3.

- HintMessager – encapsulates the *aff_core_id* into data request (for example, we can use PVFS_hint to convey *aff_core_id* in PVFS).
- SrcParser – analyzes the IP packet header and retrieves the *aff_core_id* that interrupt should be delivered.
- IMComposer – guides the I/O APIC/MSI to compose interrupt message with the *aff_core_id* which describes the destination address of the local APIC.

And, there is a core component on I/O server to put *aff_core_id* into the return I/O data packets (this is an optional component for different implementations).

- Hintcapsuler – encapsulates the *aff_core_id* into every return data packet on the I/O server.

Because SAIs uses the application level information (*aff_core_id*) to instruct system level interrupt scheduling, the implementation of SAIs includes some modifications to the networking protocol, system interrupt scheduling, and parallel file system. The modifications on I/O server side could be involved into SAIs, depending on the implementation method. In our prototype, because PVFS is employed to serve application I/O requests, PVFS_hint message can convey *aff_core_id* information. We only make some minor modifications on the I/O server side in our prototype.

B. Implementation Mechanism

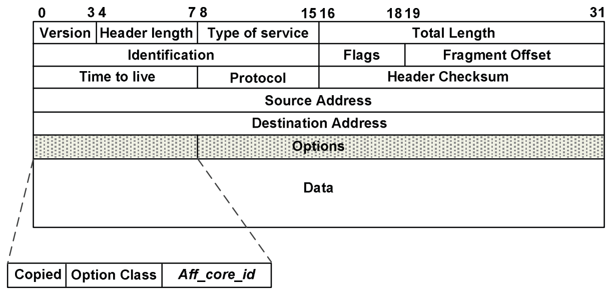

Figure 4. IP Packet Structure with Aff_core_id.

The detailed implementation mechanism of SAIs under PVFS is shown in Figure 3. When an application process, for example App A, needs to request data from PVFS, the *aff_core_id* will be packed into data request by HintMessager as a hint parameter (which is described by step 1 and 2). After that, App A sleeps to wait for the arrival of return data. When the data request is received by I/O server, HintCapsuler puts *aff_core_id* into all the return data packets (step 3). Because PVFS uses TCP/IP to transfer I/O data between client side and I/O server nodes, *aff_core_id* could be encapsulated into a network packet to return to the client side. To avoid the extra cost on network protocol design, the options field of the IP level will be reserved to convey *aff_core_id*. An options field is an additional header field with maximum size of 32-bit word which may follow the destination address field [24]. The options field also may be an 8-bit simple options field which could be terminated with an EOL (0x00) option. In addition, options field of the IP packet head could be parsed by NIC device driver on the client side before the interrupt is generated. The detailed description is shown in Figure 4. The 8-bit simple options field consists of three sub-fields: copied, option class, and option number. The value of 1-bit Copied field and 2-bit option class field are both set to 1 following TCP/IP protocol description. With 5-bit option number field describing the affinitive core, a maximum $2^5 = 32$ cores could be identified by SAIs.

On the client side, the NIC device driver analyzes the incoming MAC frames and composes them as IP packets. When the IP packet is ready to deliver to system IP module, SrcParser parses the IP packet header to extract *aff_core_id* from options field in the NIC device driver. After the *aff_core_id* extract operation, the NIC device driver issues one softirq interrupt message, in which *aff_core_id* has been added as the destination address of the local APIC by IMComposer (described by step 4 and 5). The interrupt message is then delivered to the affinitive core for processing (step 6). When the according core completes interrupt handling and packet processing, inter-core signals are sent to wake the application process. To avoid the application process being migrated to another core when the data returns, SAIs enforces that the application process should be bundled on the core which requested data before data return. In a nutshell, a data request causes multiple return packets from multiple sever nodes at the same time, but SAIs guides all the interrupts to the cores corresponding to the *aff_core_id* encapsulated in the packets.

V. EXPERIMENTAL EVALUATION

The SAIs scheduler has been integrated into the client side kernel and NIC drivers to verify its benefit for parallel I/O application. Our performance evaluation is based on the analysis and compares the four commonly used metrics: bandwidth, cache miss rate, processor utilization, and, cpu_clk_unhalted.

A. Experimental Setup

Our experiments were conducted on a 49-node Sun-Fire Linux-based cluster. This cluster is composed of one Sun-Fire 4240 head node and 48 Sun-Fire 2200 compute nodes. The head node is configured with two Quad-Core 2.7 GHz AMD Opteron Series 2384 processors (512KB dedicated L2 cache per core), 8 GB memory and three 1 Gigabit Ethernet Ports with BCM5715C controller. Every compute node is configured with two Quad-Core 2.3 GHz AMD Opteron Series 2376 processors (512KB dedicated L2 cache per core), 8 GB memory and three 1 Gigabit Ethernet Ports with BCM5715C controller. The head node has 4X 146GB 10K-RPM SAS hard drives. Each compute node has a 250GB 7.2K-RPM SATA-II hard drive. The cluster has been connected by the Cisco Catalyst 4948 10/100/1000BASE-T switch. In our testing, PVFS 2.8.1[29] was set up as parallel file system which is accessed by the I/O client node. PVFS was configured with one metadata server node and variable I/O server nodes (from 8 to 16, 32, 48 nodes) with a 64KB strip size. The I/O client is configured on the head node. Because parallel I/O read is the most frequent operations and TCP is the most widely used transport protocol in PVFS, our experiments mainly focused on parallel file system read with a TCP configuration.

B. Experimental Description

The performance of SAIs and Irqbalance are evaluated and compared in our experiments with the running of Interleaved or Random Benchmark (IOR) [4]. IOR is a parallel file system benchmark which is developed by Lawrence Livermore National Laboratory to test the performance of various parallel I/O patterns. Because IOR includes general parallel I/O operations and various real parallel I/O patterns, it is widely accepted as a benchmark for parallel I/O test. Each I/O operation in IOR writes or reads a contiguous block of buffer (transfer size up to the entire memory available) to/from the parallel file system. Because every IOR request (the size is configured as transfer size) involves parallel I/O, the return data generates multiple concurrent I/O interrupts. Based on the conventional load balance core scheduling scheme data needs to be merged on every IOR request. Therefore, IOR is an ideal benchmark for the SAIs performance evaluation. To make the experiment match general application situations, we have added some computing tasks into IOR. These computing tasks encrypt the data collected by every IOR request. IOR is available with three APIs: MPI-IO, POSIX, and HDF5. In the experiment, MPI-IO tests are conducted on the client side parallel accessing PVFS with different transfer sizes from 128KB, 512KB, 1 MB, to 2 MB. The number of PVFS I/O server nodes number varies from 8, 16, 32, to 48. The strip size of every I/O server node is 64KB. In the experiments, the client side executes an IOR process to read a 10GB size file from PVFS. The performance has been measured by Oprofile [28] and Linux inbuilt "sar" system monitor tool [26]. Consistent results are obtained across repeated runs. All results presented in the paper are averaged with at least three runs. Since no data locality issue has been observed at the core interrupt scheduling level in parallel I/O write, our experiments mainly focus on parallel I/O read.

C. Bandwidth Comparison

The bandwidth comparison experiments have been conducted with 1 Gigabit and 3 Gigabit NIC (combined three 1 Gigabit NICs). On the 1 Gigabit NIC, SAIs employs multiple IOR processes and 1 Gigabit NIC to parallel access PVFS. Although SAIs shows better performance than Irqbalance on I/O bandwidth with 1 Gigabit NIC, the limited network bandwidth is a major bottleneck and reduces the potential performance improvement of the application. Therefore, SAIs has moderately improved IOR I/O bandwidth with 1 Gigabit NIC. The bandwidth peak speed-up ratio is 6.05%.

Figure 5 shows the performance improvement of SAIs with 3 Gigabit NIC. The IOR processes are executed on the I/O client side to access data on the PVFS concurrently. Each process reads a total 10GB data from PVFS. It lists the I/O bandwidth comparison and speed-up under the two scheduling schemes with various transfer sizes and number of I/O servers. As we can see in Figure 5, SAIs improves the I/O bandwidth in all cases. Especially, when the number of I/O servers is increased to 48, the speed-up reached a maximum of 23.57%. Note the maximum bandwidth in Figure 5 doesn't exceed 3 Gigabit. Hence this result complies with the analysis in section 3.3. The analysis states that the performance advantage of SAIs can rise with the number of servers, if the bandwidth doesn't become a bottleneck.

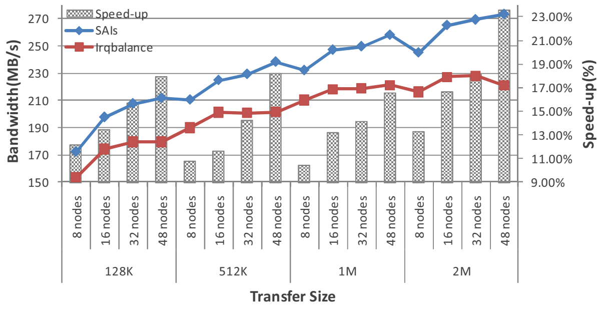

Figure 5. Bandwidth Comparison with 3-Gigabit NIC.

D. Cache Miss Rate Analysis

Figure 6 shows the ratio of L2 cache miss rates (# cache misses / # accesses) of the two interrupt scheduling schemes with a 1 Gigabit NIC. The experimental results expose the major cause of the bandwidth improvement. As we discussed in the second paragraph of the Introduction Section, a cache miss leads to an extra data movement between the two cores. So reducing the number of cache misses caused by an I/O interrupt means a reduction of data movement. However, increasing the number of I/O servers leads to more interrupts and higher I/O throughput on the client side. Therefore, even with a lower cache miss rate, the 48 server's configuration does cause more cache misses, which may lead to more data movement too. Figure 6 shows that our method works well when the number of I/O server's increases, since the cache miss rate is smaller than that of the Irqbalance scheduling.

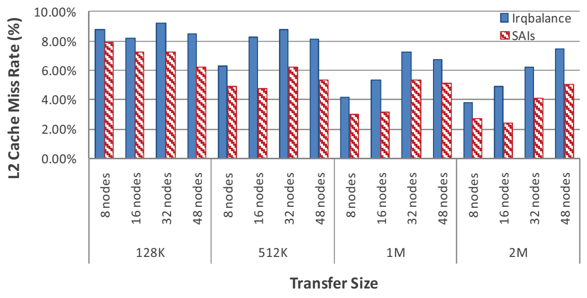

Figure 6. L2 Cache Miss Rate Comparison with 1 Gigabit NIC.

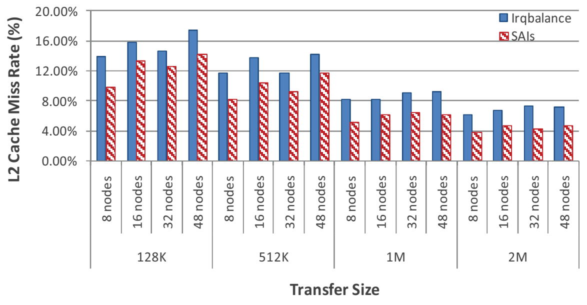

Figure 7. L2 Cache Miss Rate Comparison with 3-Gigabit NIC.

In Figure 7, the L2 cache miss rates have been compared using a 3-Gigabit NIC. The results show that the cache miss rates have increased with the increase of the network bandwidth, leaving a big improvement space for SAIs. In this experiment, the L2 miss rate is reduced almost 40% by SAIs.

E. CPU Utilization Analysis

Although SAIs improves the I/O bandwidth noticeably in comparison to Irqbalance, the improvement is less than its potential as shown in our analysis. To further explore the possible reasons, the CPU total utilizations has been displayed in Figure 8.

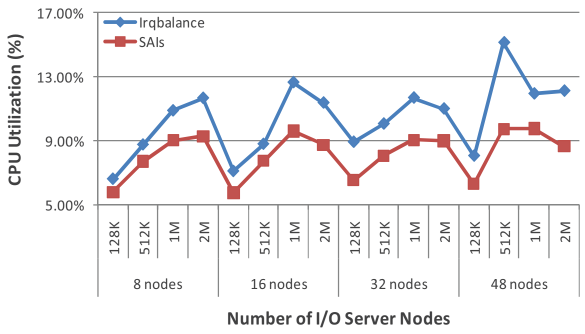

Figure 8. CPU Utilization Comparison with 1 Gigabit NIC.

In Figure 8, the CPU utilization is collected under a single application running with a 1 Gigabit NIC. The CPU exposes its low utilization with the maximum of 15.13%, whatever the interrupt scheduling scheme is selected. This is because that the bandwidth of a 1 Gigabit NIC is lower than the processing capacity of CPUs, even with only one core (2.7GHz). When the processor is processing faster than NIC receiving speed, the NIC will be the main bottleneck in parallel I/O access. Therefore, there are many more CPU cycles idling to wait for the NIC to receive data. Therefore, while the SAIs scheduling still shows a better bandwidth than that of Irqbalance with 1Gigabit NIC, the improvement is small. This observation also supports the rational of assumptions for equation (1), and rules out the possibility that the parallel interrupt handling for core utilization could offset the data movement cost.

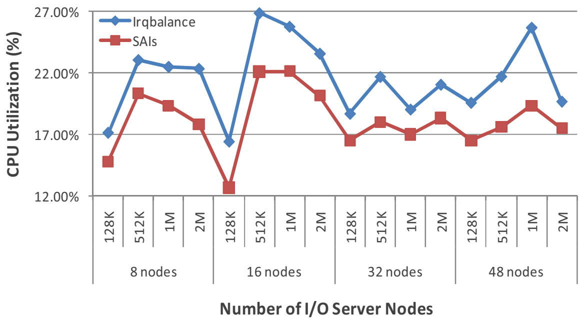

Figure 9. CPU Utilization Comparison with 3-Gigabit NIC.

In Figure 9, the CPU utilization has been listed with 3-Gigabit NIC. The results show that the Irqbalance employs more CPU cycles on data movement. Although 3 Gigabit networking bandwidth still cannot saturate all core computing capacity, the increasing CPU utilization shows a possible linear relation between CPU capacity and network speed. We will further verify this linear relation by conducting a simulation in Section 6.

F. CPU Waiting I/O Time Analysis

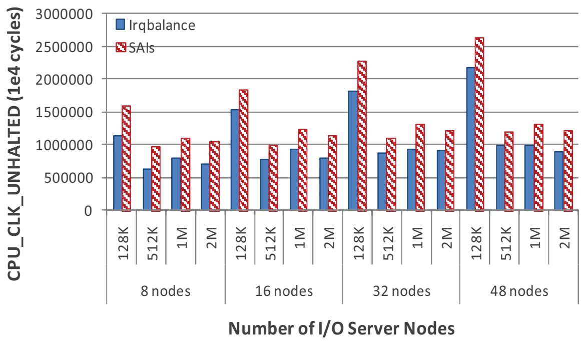

Figure 10. CPU I/O Wait Comparison with 1Gigabit NIC.

To further analyze the CPU utilization for I/O handling, the CPU_CLK_UNHALTED event has been collected with mask 0x00 by Oprofile in the experiments. This event provides the number of clocks that the CPU is not in a halted state [30]. In our experiments, we collect this event to analyze the halted time that CPU waits on I/O data. For the 1 Gigabit NIC experiment, the results are shown in Figure 10, SAIs has a maximum of 27.14% improvement on CPU_CLK_UNHALTED time. When a data-intensive parallel application, such as IOR, reads data from the PVFS, the CPU halted cycles are mainly contributed by two parts: 1. the time that the I/O core (in which I/O interrupts are handled) is halted and waited for data to be received by the NIC; 2. the time that application core (in which IOR is running) is halted and waited for data when data misses in the cache. Obviously, SAIs scheduling the I/O interrupt to its affinitive application core removes the time cost of part 2. Therefore, SAIs obtains a larger CPU unhalted time.

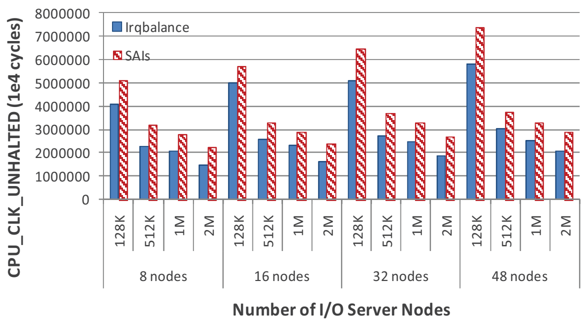

Figure 11. CPU I/O Wait Comparison with 3-Gigabit NIC.

Figure 11 presents the CPU_CLK_UNHALTED events to compare the application's waiting time for I/O read with 3-Gigabit NIC. SAIs has a maximum of 48.57% improvement on CPU_CLK_UNHALTED time. The results verify that SAIs reduces the I/O waiting time for each read and increases the total I/O bandwidth.

G. Multiple Clients I/O Bandwidth Testing

Because the *source-aware* interrupts scheduling mainly optimizes parallel I/O performance on the client side, multiple client performance testing has been conducted and analyzed in this subsection to evaluate scalability. The experiment configured 8 I/O server nodes and a variable number of client nodes (from 4, 8, 16 nodes to 56 nodes) with a 3 Gigabit NIC connection. Every client node ran multiple IOR application processes (transfer size = 1M). The I/O bandwidth with the different interrupt scheduling schemes has been collected and compared in Figure 12 (The bandwidth is a summary of the whole clients).

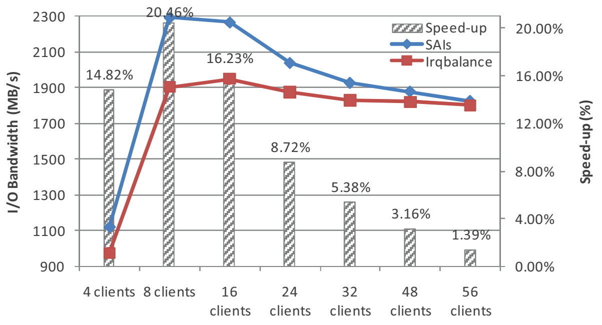

Figure 12. Multiple Clients I/O Bandwidth Comparison.

Figure 12 shows that SAIs has improved parallel I/O bandwidth. When the number of clients is 8, the improvement is up to 20.46%. With the further increase in number of clients, the I/O bandwidth decreases slowly. Therefore, 20.46% is the maximal improvement for 8 I/O server nodes in our experiment. Because the bandwidth of 8 I/O server nodes is saturated by 8 clients, the increasing client nodes over 8 will gradually reduce the bandwidth on a single client. The drop in bandwidth implies a drop in N_R, the number of requests. According to formula (5) and (6) in Section III, this in turn will reduce the difference in effectiveness between SAIs and irqbalance scheduling as shown in Figure 12. When the number of client nodes is greater than 32, 8 I/O nodes are not enough to serve the increased parallel I/O requests. In these overloaded worst cases, SAIs still improves application performance slightly. The experimental results verify that source-aware interrupt scheduling improves or maintains the parallel I/O performance on the multiple clients' situation. However, the peak network bandwidth of the I/O servers will eventually limit the performance benefits of SAIs. To understand the potential performance improvement of SAIs, simulation is conducted in the following section in order to remove the physical network constraint.

VI. CACHE DATA MIGRATION COST SIMULATION

Because the speed of NIC generating I/O interrupts is much slower than the speed of processors handling I/O interrupts on the client side in our experiments, the experiments do not demonstrate the full potential of our interrupts scheduling scheme. To detect the possible performance improvement brought by source-aware scheduling, we conduct a simulation in memory to evaluate data movement cost. In our simulation node (the head node of the Sun-Fire Linux-based cluster), the system configures 4X 2GB DDR2-667 Single Rank Memory, which could provide 5333 MB/s (about 41.66 Gigabit/s) peak bandwidth [22] for the parallel I/O access. Because the main contribution of SAIs is removing the extra data movement cost incurred by interrupts scheduling, our simulation focused on keeping source-aware to avoid cache misses and reduce data movement. In addition, the interrupt handling cost depends on the interrupt processing routine rather than scheduling scheme, thus the interrupt handling cost is a constant cost to any interrupt scheduling. The data processing method of SAIs is simulated by a pair of threads (named Si-SAIs), in which one thread parallel read data strips from multiple different files on a RAM disk [34] in the memory and the other one combines the returned data strips together into the requested data. Si-SAIs employs the system resource sharing feature of the threads to keep it source-aware. We use two independent processes (named Si-Irqbalance) to simulate the Irqbalance data processing method completing the same job as Si-SAIs. The independent processes are possible to be scheduled and executed on separated cores. The major performance issue, extra data movement, has been reproduced by our simulation (shown in Figure 13). The multiple I/O nodes are simulated with different files stored in memory. Each I/O thread or process read includes 64KB data strip from every file. The transfer size is 1M, which has been verified to be the best buffer size in our previous testing. The simulation results are obtained across repeated runs. All results are averaged with at least three runs.

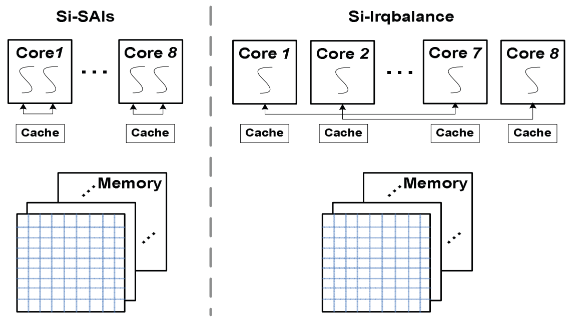

Figure 13. Memory Parallel I/O Access Simulation Design.

Figure 14 shows the testing results of the simulation. In Figure 14, the bandwidth reaches up to 3576.58 MB/s (about 27.94 Gigabit/s) when the CPU utilization is 49.47%. And, the corresponding speed-up is up to 53.23%. The L2 cache miss rate has reduced 51.37% on this peak bandwidth improvement. When the application number equals the number of cores, the CPU capacity is saturated by applications and utilization reaches 99.47%. After the CPU keeping 100% utilization, Si-SAIs and Si-Irqbalance sustain almost the same performance (about 2500MB/s or 19.53 Gigabit/s) for the parallel I/O for the all case. With the results of simulation, we conclude that two Quad-Core 2.7 GHz AMD Opteron Series 2384 processors (head node processor configuration) could handle parallel I/O bandwidth up to 27.94 Gigabit/s and 19.53 Gigabit/s on average.

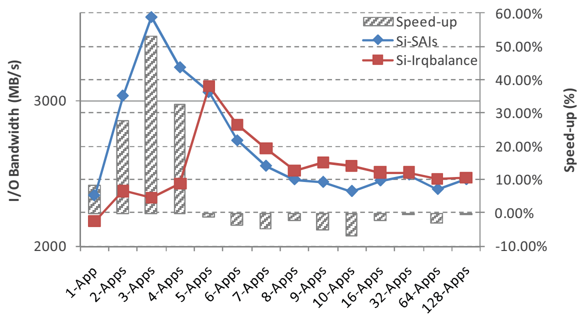

Figure 14. Simulation I/O Bandwidth Comparison.

Analysis and experimental results show that SAIs is very promising. It has its merit. On the other hand, SAIs has its limitations. SAIs is designed for parallel I/O systems. It is not a general interrupt scheduling. Its effectiveness depends on the assumption that the underlying system is I/O intensive and that the system has plenty of network bandwidth. SAIs may serve well as a complement of existing processor scheduling schemes for datacenters with high-speed networks connections and for data intensive applications. But in general, extending the source-aware concept and integrating source-aware scheduling with existing interrupts scheduling mechanisms is a subject of future study.

VII. RELATED WORKS

The prevalent interrupt scheduling schemes adopted by current multi-core OSes are round-robin and dedicated modes. Round-robin mode distributes interrupts from I/O APIC to local APICs in turn, and, dedicated mode delivers the incoming interrupts to a fixed core. Irqbalance[20] is a intelligent interrupt scheduling loadable module, which makes interrupts scatter on every possible core based on the cores' load statistics. Actually, Irqbalance is a variant of round robin scheduling mode. In addition, a patent of interrupt load distribution system proposed by Toshikazu Nakagawa [38] gave anther interrupt scheduling method with the consideration of processor load balance. While the existing works have shown good potential on CPU utilization, uncoordinated attempts to distributed interrupts to different core can also result in some bad side effects [40].

Several processor data locality research efforts have been conducted for network performance. These research efforts partly exploited the potential scenario and cost of data movement among cores. The impact of the data movement incurred by parallelization strategies of packet processing on the general-purpose monolithic OS has been analyzed by Salehi et al. [21] and Willmann et al. [32]. As for multi-core systems, Foong et al. [1][2][5] and Narayanaswamy et al. [10][11] have shown the in-depth analysis of processor data locality problem, but there analysis has not been considered for parallel I/O situations. To enable users tuning applications performance to keep data locality and reduce data movement above multi-core systems, VTune [19] and autopin [37] have been developed by Intel and T. Klug et al.

Though these tools suggest an optional data-core mapping, they cannot detect the application core information and change the source-aware automatically while processes are running. In addition, the latest Intel Ethernet Controller 82575/82576 or 82598/82599 [17] allows assigning interrupts to processor cores manually. But, the assignment is static, which is too inflexible to meet the change of the data request source. Effective and adaptive load balancing on multiprocessor systems has been studied in [6][36][39]. Brecht et al. proposed a dedicated processor core packet processing solution to improve data locality [36], but this solution sacrificed the parallelism. Scogland et al. have suggested a user-level library called SyMMer [39], which monitors the system loads changing and re-schedules the MPI application processes running for keeping data locality. In contrast, our study is distinguished from the previous researches in the sense that we proposed a novel source-aware interrupt scheduling scheme, which uses the source information of parallel-I/O to optimize core interrupt scheduling. The basic idea is that the interrupts associated for the same I/O request, even if they come from different file servers, should be grouped together to one core. This simple idea carries a long way, since data locality is much more important in performance than core utilization in modern computers. The source-aware interrupt scheduling scheme reduces cache misses and data movements between caches.

There are also research which improve the processor data locality in intra-node communication [13][14]. However, these optimizations are good for data exchange among cores rather than the data locality of interrupt scheduling. Suggestions of keeping data locality for high-performance networking has been proposed recently in [7][15][27][35]. These systems can also benefit from SAIs to achieve high parallel I/O bandwidth with less system modification and integration.

VIII. Conclusion and Future Works

We have proposed a novel source-aware affinity interrupt scheduling scheme and prototyped it with a new scheduler called SAIs for parallel I/O systems. SAIs groups interrupts associated for the same I/O request together to be handled on the same core. The new scheme ties interrupt processing and data consumption, to reduce cache miss rate and data movement on client side. Experimental results show that SAIs obtains noticeable better I/O bandwidth than that of the conventional utilization based scheduling mechanisms. SAIs, which has been integrated into the Linux kernel, has reported an improvement up to 23.57% in the 3-Gagebit NIC configuration of our testing environment. To explore the full potential of the source-aware scheduling, simulations are also conducted that remove the NIC bottleneck. The simulation results show that SAIs can improve the I/O bandwidth up to 53.23% accompanied with 51.37% cache miss rate reduction. The successful implementation of SAIS shows that the newly proposed source-aware affinity mechanism is feasible and effective. The analysis and experimental results demonstrate the potential of source-aware interrupt scheduling for data intensive applications where network bandwidth is not the performance bottleneck.

The proposed source-aware interrupt scheduling is very promising and leads to a considerable performance improvement. However, it is just a beginning. To put the source-aware interrupts scheduling in actual use, we need more studies. We list four different interrupts handling policies in Section 3. Our current study is on one policy with one special application, parallel I/O interrupts, in mind. Our current result is not a general solution of interrupt scheduling. It is a complement and alternative. In the future, we plan to extend the source-aware concept to other applications and to study the integration of different policies and scheduling algorithms for a robust, general solution.

IX. Acknowledgements

The authors would like to thank Dr. Yong Chen and other members and former members of the Scalable Computing Software Laboratory at the Illinois Institute of Technology for their insightful feedbacks and discussions on this paper. This research was supported in part by National Science Foundation under NSF grant CCF-0621435, CNS-0751200, and CCF-0937877.

References

[1] A. Foong, J. Fung, and D. Newell, "An In-Depth Analysis of the Impact of Processor Affinity on Network Performance," *In Proceeding of the 12th IEEE International Conference on Networks* (ICON 2004), November 16-19, 2004.

[2] A. Foong, J. Fung, D. Newell, S. Abraham, P. Irelan, and A. Lopez-Estrada, "Architectural Characterization of Processor Affinity in Network Processing," *In Proceeding of the IEEE International Symposium on Performance Analysis of Systems and Software* (ISPASS 2005), May 16, 2005.

[3] AMD, "AMD64 Architecture Programmer's Manual Volume 2: System Programming", AMD Corporation, 2009.

[4] ASC Sequoia Benchmark Codes, IOR summary, https://asc.llnl.gov/sequoia/benchmarks/#ior.

[5] B. Veal and A. Foong, "Performance Scalability of a Multi-Core Web Server," *In Proceeding of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (ANCS'07), Dec. 2007.

[6] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf, "A Flexible Architecture Integrating Monitoring and Analytics for Managing Large-Scale Data Centers," In Proceedings of the 8th International Conference on Autonomic Computing (ICAC 2011), Jun. 2011.

[7] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *In Proceeding of the 41st Annual IEEE/ACM International Symposium on Microarchitecture* (MICRO-41), Nov. 2008.

[8] F. Inoue, H. Ohsaki, Y. Nomoto, and M. Imase, "On Maximizing iSCSI Throughput using Multiple Connections with Automatic Parallelism Tuning," *In Proceedings of the 5th IEEE International Workshop on Storage Network Architecture and Parallel I/Os* (SNAPI), Sep. 2008.

[9] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA – Preparatory Data Analytics on Pera-Scale Machines," *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium* (IPDPS 2010), Apr. 2010.

[10] G. Narayanaswamy, P. Balaji, and W. Feng, "An Analysis of 10-Gigabit Ethernet Protocol Stacks in Multicore Environments," *In Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects* (Hot Interconnects - '07), August 22-24, 2007.

[11] G. Narayanaswamy, P. Balaji, and W. Feng, “Impact of Network Sharing in Multi-core Architectures,” *In Proceedings of the 17th International Conference on Computer Comunications and Networks* (ICCCN ’08), Aug. 2-7, 2008.

[12] H.-C. Jang and H.-W. Jin, “MiAMI: Multi-Core Aware Processor Affinity for TCP/IP over Multiple Network Interfaces,” *In Proceedings of the 17th IEEE Symposium on High Performance Interconnects* (Hot Interconnects - ’09), Aug. 26-27, 2009.

[13] H.-W. Jin, S. Sur, L. Cai, and D. K. Panda, “LiMIC: Support for High-Performance MPI Intra-Node Communication on Linux Cluster,” *In Proceedings of the 2005 International Conference on Parallel Processing* (ICPP-05), Jun. 2005.

[14] H.-W. Jin, S. Sur, L. Cai, and D. K. Panda, “Lightweight Kernel-Level Primitives for High-Performance MPI Intra-Node Communication over Multi-Core Systems,” *In Proceeding of IEEE International Conference on Cluster Computing* (Cluster 2007), Sep. 2007.

[15] H. Sivakumar, S. Bailey, and R Grossman, “PSockets: The Case for Application-level Network Striping for Data Intensive Applications using High Speed Wide Area Networks,” *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC 2000), Nov., 2000.

[16] H. Zou, W. Wu, X-H Sun, P. DeMar, M. Crawford, “An Evaluation of Parallel Optimization for OpenSolaris Network Stack,” *In Proceedings of the 35th IEEE Conference on Local Computer Networks* (LCN 2010), Oct. 11-14, 2010.

[17] Intel, “Assigning Interrupts to Processor Cores using an Intel R 82575/82576/82598/82599 Ethernet Controller,” http://download.intel.com/design/network/applnots/319935.pdf , Sep. 2009.

[18] Intel, “Intel 64 and IA-32 Architectures Software Developer’s Manual Volume 3A: System Programming Guide, Part 1,” Intel Corporation, Mar., 2010.

[19] Intel, “VTune Performance Analyzer”, http://software.intel.com/en-us/intel-vtune/.

[20] Irqbalance, http://irqbalance.org/.

[21] J. D. Salehi, J. F. Kurose, and D. Towsley, “The Effectiveness of Affinity-Based Scheduling in Multiprocessor Network Protocol Processing”, *IEEE/ACM Transactions on Networking*, Vol. 4(4), Aug., 1996.

[22] “JEDEC standard: DDR2 SDRAM Specification,” JESD79-2F, Nov. 2009.

[23] J. Hennessy and D. Patterson. Computer Architecture: A Quantitative Approach. The 4th edition, Morgan Kaufmann, 2006.

[24] Jon Postel, “Internet Protocol-DARPA Internet Program Protocol Specification”, RFC 791, Sept. 1981.

[25] Linux Cross Reference, http://lxr.linux.no/.

[26] Linux man page - sar, http://linux.die.net/man/1/sar.

[27] “Lustre File System Networking: High-Performance Features and Flexible Support for a Wide Array of Networks”, A White Paper from Lustre File Systems, Jan., 2008.

[28] Oprofile, http://oprofile.sourceforge.net/.

[29] Parallel Virtual File System, http://www.pvfs.org/.

[30] “Performance Monitoring Events - AMD Family 11h Processors”,http://developer.amd.com/cpu/CodeAnalyst/codeanalystlinux/Documents/CodeAnalyst-Linux-help/pmes_fam11h.htm.

[31] “Processor Affinity White Paper for Multiple CPU Scheduling,” TMurgent Technologies, Nov. 3, 2003.

[32] P. Willmann, S. Rixner, and A. Cox, “An Evaluation of Network Stack Parallelization Strategies in Modern Operating System,” *In Proceedings USENIX Annual Technical Conference*, May 30 – June 3, 2006.

[33] R. Love, “Linux Kernel Development, 2nd Edition,” Novell Press, ISBN-10: 0672327201, 2005.

[34] RAM disk – Linux Kernel Documentation: Using the RAM disk block device with Linux, http://www.mjmwired.net/kernel/Documentation/ramdisk.txt/

[35] S. Miura, T. Okamoto, T. Boku, T. Hanawa, and M. Sato, “RI2N: High-bandwidth and fault-tolerant network with multi-link Ethernet for PC clusters,” *In Proceedings of IEEE International Conference on Cluster Computing* (Cluster 2008), Sep., 2008.

[36] T. Brecht, G. Janakiraman, B. Lynn, V. Saletore, Y. Turner, “Evaluating Network Processing Efficiency with Processor Partitioning and Asynchronous I/O,” *In Proceedings of the EuroSys 2006*, Apr. 18-21, 2006.

[37] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, “auto-pin – Automated Optimization of Thread-to-Core Pinning on Multicore Systems,” *Transactions on High-Performance Embedded Architectures and Compilers*, Vol. 3(4), 2008.

[38] T. Nakagawa, “Interrupt Load Distribution System for Shared Bus Type Multiprocessor System,” Patent no.: US 6,237,058 B1, May. 2001.

[39] T. Scogland, P. Balaji, W. Feng and G. Narayanaswamy, “Asymmetric Interactions in Symmetric Multi-core Systems: Analysis, Enhancements and Evaluation,” *In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (SC 2008), Nov., 2008.

[40] V. Anand and B. Hartnet. “TCP/ IP Network Stack Performance in Linux Kernel 2.4 and 2.5,” *In Proceedings of the Linux Symposium, Ottawa June 2002*, Jun. 2002.

[41] X.-H. Sun and Y. Chen, “Reevaluating Amdahl's Law in the Multicore Era,” *In Proceedings of Journal of Parallel and Distributed Computing*, vol. 70 (2), Feb., 2010.