Jigsaw: A High-Utilization, Interference-Free Job Scheduler for Fat-Tree Clusters

Staci A. Smith[*]
smiths949@cs.arizona.edu
Department of Computer Science
The University of Arizona
Tucson, AZ, USA

David K. Lowenthal
dkl@cs.arizona.edu
Department of Computer Science
The University of Arizona
Tucson, AZ, USA

ABSTRACT

Jobs on HPC clusters can suffer significant performance degradation due to inter-job network interference. Approaches to mitigating this interference primarily focus on reactive routing schemes. A better approach—in that it completely eliminates inter-job interference—is to implement scheduling policies that proactively enforce network isolation for every job. However, existing schedulers that allocate isolated partitions lead to lowered system utilization, which creates a barrier to adoption.

Accordingly, we design and implement Jigsaw, a new job-isolating scheduling approach for three-level fat-trees that overcomes this barrier. Jigsaw typically achieves system utilization of 95-96%, while guaranteeing dedicated network links to jobs. In scenarios where jobs experience even modest performance improvements from interference-freedom, Jigsaw typically leads to lower job turnaround times and higher throughput than traditional job scheduling. To the best of our knowledge, Jigsaw is the first scheduler to eliminate inter-job network interference while maintaining high system utilization, leading to improved job and system performance.

CCS CONCEPTS

• **Computer systems organization → Parallel architectures**.

KEYWORDS

Fat-tree, inter-job network interference, scheduling, utilization

ACM Reference Format:
Staci A. Smith and David K. Lowenthal. 2021. Jigsaw: A High-Utilization, Interference-Free Job Scheduler for Fat-Tree Clusters. In *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '21), June 21–25, 2021, Virtual Event, Sweden.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3431379.3460635

1 INTRODUCTION

Schedulers on most HPC clusters allocate dedicated nodes to each job but do not take network resources into account. This means that multi-node jobs often compete with each other for the network, which can lead to substantial job performance degradation [6–8]. The resulting performance degradation can be problematic for administrators as it leads to higher job turnaround times and lower system throughput, and the variability is also problematic for users when tuning code or determining hours to request.

Due to the potential for large job slowdowns, there has been significant research into alleviating inter-job network interference. Many mitigation approaches focus on avoiding congestion through routing techniques, both in hardware [28] and software [10, 30]. These routing techniques all make a best effort to route traffic *after* the job scheduler places jobs onto the cluster. They cannot make any guarantees about worst-case interference, and they require more complicated hardware and software algorithms for routing.

A conceptually more elegant approach is to have the job scheduler enforce complete network isolation of application traffic for each job via resource allocation policies. In addition to the obvious performance benefit, this decreases the burden on system software and avoids the complexity of hardware adaptive routing. This is not a new idea; for example, the scheduler on BlueGene/L systems allocates only electrically-isolated partitions, or *midplanes*, to jobs [1]. Much more recently, researchers have also studied schedulers that can achieve similar isolated network partitions on fat-tree based systems [26, 37]. However, one major difficulty with such *job-isolating schedulers* is that in order to ensure that performance is not degraded, the scheduler must provide each job access to the same guarantee of underlying bandwidth in its isolated partition as there is on the full system. This requirement necessarily leads to constraints on job-to-node assignment, which tends to lower system utilization. In fact, it is quite difficult to create job-isolating schedulers that achieve high system utilization, and this creates a key barrier to widespread adoption. In particular, no job-isolating scheduler to date has delivered system utilization generally over 90%, whereas traditional job schedulers usually deliver utilization above 97% under sufficient system demand.

In this work, we remove the aforementioned utilization barrier with *Jigsaw*, the first scheduling approach we know of that *simultaneously* achieves job-level network isolation, full bandwidth of job allocations, *and* high system utilization on general three-level fat-trees. When scheduling a job, Jigsaw obeys novel conditions for node and link allocation that guarantee the availability of full bandwidth in every allocated partition. Because the conditions are precise—in contrast to the heuristic conditions used by previous approaches—Jigsaw is able to reduce fragmentation of system resources, which is the key to high utilization. Our experiments show that Jigsaw achieves 95-96% system utilization in most cases,

[*] Smith is currently at Google, Inc.

HPDC '21, June 21–25, 2021, Virtual Event, Sweden
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8217-5/21/06…$15.00
https://doi.org/10.1145/3431379.3460635

without sacrificing the bandwidth available to jobs. This combination typically leads to lower average job turnaround time and higher throughput than traditional job scheduling under even modest assumptions about performance improvement from job-level isolation.

Our work makes the following contributions:

- We develop a new job scheduler, Jigsaw, that is the first to provide job-level isolation while both maintaining full interconnect bandwidth and fully utilizing system resources.
- We develop novel formal conditions on node and link allocation and use them to prove that Jigsaw allocates full-bandwidth partitions. The proof includes an intermediate result that three-level fat-trees are rearrangeable non-blocking, which as far as we know is the first such proof.
- We show that Jigsaw improves system utilization significantly—by 4-7 percentage points—compared to previous job-isolating schedulers. This utilization increase, combined with job isolation, leads to a decrease in average job turnaround time by up to nearly 50% compared to traditional job scheduling.

Our work shows that scheduling approaches to eliminate inter-job network interference on fat-trees can provide strong guarantees on job performance, maintain high system utilization, and remove the need for complex routing approaches to manage network contention. Guaranteeing isolated network partitions to jobs also allows application developers to focus on minimizing *intra*-job network interference through existing techniques [11, 13, 17], without concern that outside interference will frustrate their efforts. The use of a scheduler like Jigsaw would improve job performance on current HPC clusters while maintaining the high system utilization that administrators often require.

2 BACKGROUND

In this section, we discuss the fat-tree topology, inter-job network interference, and existing mitigation strategies for interference.

2.1 Fat-Tree Networks

The fat-tree is a popular network topology for HPC computing clusters. Originally proposed by Leiserson [23], a fat-tree is a tree whose links become “fatter" (have higher bandwidth) at each level going up from leaves to root. Because the link bandwidth increases, the links near the root of the tree do not create a bottleneck when nodes communicate across the tree.

In practice, fat-tree networks are constructed using high-radix routers connected in a folded Clos topology [20]. Current high-radix routers allow three-level fat-trees to scale up to over ten thousand nodes, and the folded Clos topology is easily wired out of routers and links with uniform radix and bandwidth.

2.2 Inter-Job Network Interference

In recent years, inter-job network interference has been identified as a culprit behind job performance variability on HPC systems. On torus- and dragonfly-based clusters, it can cause production applications to slow down by 100-150% [6, 7, 30]. Although full fat-tree networks have uniform bandwidth at each level and are rearrangeable non-blocking, they too can suffer from network interference due to job placement, communication patterns, and routing [16]. Under static routing [36], which is typically used on fat-tree clusters, multi-job workloads can lead to network hotspots despite the attempt to balance all possible paths across links [10, 22, 30]. As a result, communication intensive benchmarks slow down by as much as 120% in controlled experiments [30], and production applications slow down by as much as 66% in simulations [18].

2.3 Existing Mitigation Approaches

We break existing approaches to mitigating inter-job network interference into routing-based and scheduling-based approaches.

2.3.1 *Routing-Based Approaches.*

There are currently a few approaches to decreasing inter-job network interference using routing mechanisms [10, 22, 30]. These approaches do not require changes to the job scheduler; they all work by attempting to route packets in a way that reduces congestion *after* job placements are already fixed. An advantage of this strategy is that it does not introduce any scheduling constraints, leaving node utilization unaffected. However, these routing mechanisms cannot guarantee that the worst-case performance degradation for jobs is small. Thus, jobs may experience significant performance degradation anyway, degrading throughput and turnaround time despite high node utilization. We discuss more specifics of these approaches in Section 7.

In addition to the approaches above, adaptive routing [28] is sometimes used to balance network traffic. Adaptive routing uses local information at each switch to try to detect network hotspots and send packets down less congested paths, which adds considerable complexity to the network hardware. Until recently, fat-tree networks have typically used simpler, static routing instead [36]. To the best of our knowledge, the recent Summit and Sierra systems [34] are the first HPC fat-tree-based clusters to adopt adaptive routing. We do not yet know how effectively adaptive routing balances traffic for general, production workloads on these systems. However, adaptive routing is used in dragonfly networks, where it is clearly not sufficient to mitigate all network interference under production workloads [7, 8, 30]. Perhaps most importantly, adaptive routing, like any routing mechanism, cannot guarantee freedom from inter-job interference.

2.3.2 *Scheduling Approaches.*

Alternatively, inter-job network interference can be completely eliminated by using a scheduling policy that guarantees isolated network partitions to each job in the system. At least two such scheduling policies have been proposed for fat-tree based clusters [19, 37]. In contrast to routing approaches, job-isolating scheduling requires new node placement constraints, and these can lead to system fragmentation and lower node utilization. Throughout Sections 3 and 4 we refer to both internal and external fragmentation of nodes and links, which are analogous to the phenomena of the same names in memory allocation. Specifically, internal fragmentation of nodes or links occurs when a given scheme requires that a leaf allocate all nodes or links to the same job, but the job does not use all nodes or links (analogous to wasting part of a virtual page). External fragmentation of nodes or links occurs when there are enough nodes or links for a job, but making the allocation violates node and link conditions (analogous to having sufficient free memory, but it is not contiguous).

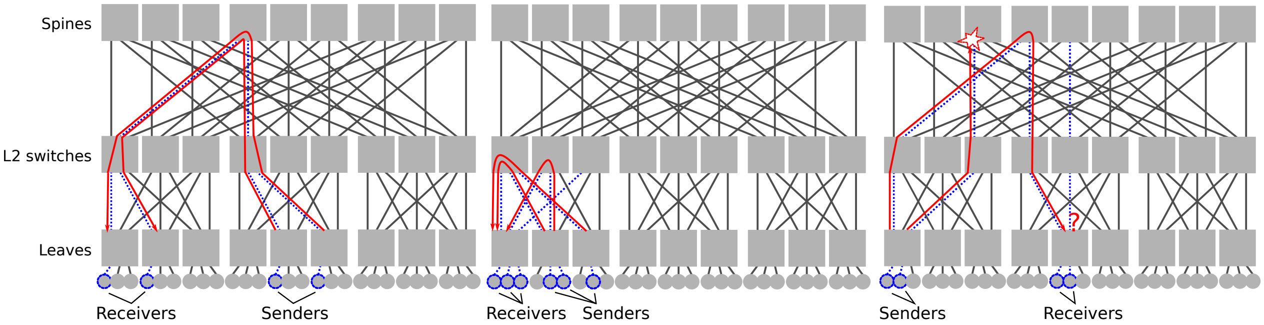

Figure 1: Violations of full interconnect bandwidth constraint. On the left, different numbers of uplinks and downlinks cause tapering; if two senders pair with two receivers, the flows must share some of the same links. In the center, arbitrary node allocations cause sharing despite sufficient links; if three senders pair with three receivers, two flows are forced to share the left-most link. On the right, poorly-chosen uplinks and downlinks, though balanced, lead to a lack of connectivity; if two senders pair with two receivers and the flows are mapped to different links at the first hop, one flow reaches a dead end at the top (denoted with a red star), and one receiver cannot receive a message (denoted with a red question mark).

In previous job-isolating scheduling approaches, node utilization has dropped by an average of 10% compared to baseline scheduling [26, 37] due to fragmentation. Since these approaches guarantee interference-freedom, however, worst case performance degradation due to inter-job interference is zero—potentially improving throughput and turnaround time despite lower node utilization. In addition, when there is no inter-job interference on the network, application developers can focus on minimizing network contention *within* the job. Because the only possible traffic on the job's network partition comes from the job itself, the developer can improve performance by leveraging knowledge of the application with existing techniques for reducing communication contention [5, 11, 13, 17, 25].

Thus, the main disadvantage of job-isolating scheduling approaches is lowered utilization; in other respects, scheduling approaches have significant advantages over current routing-based approaches. The next section describes the motivation and formalization behind Jigsaw.

3 JIGSAW THEORY

In this section, we first motivate Jigsaw's novel three-level fat-tree conditions for allocation of nodes and links to jobs. Then, we give Jigsaw's conditions themselves, which (1) provide each job with strict isolation and bandwidth comparable to a dedicated fat-tree network, as well as (2) allow the system to achieve high utilization. Finally, we outline a proof that these conditions are necessary and sufficient to lead to Jigsaw achieving the full bandwidth described in (1).

3.1 Motivation

In this work we strive to develop a job scheduler that simultaneously achieves three goals, the first two of which are constraints. First, the job scheduler must allocate isolated partitions that are free from inter-job interference. Second, the job scheduler must allow access to full interconnect bandwidth inside each partition. Finally, the job scheduler strives to provide high utilization—at least 95%—so that turnaround time and throughput will typically be better than with existing job schedulers.

Satisfying any *two* of these three goals is straightforward. For example, a scheduler can trivially provide isolation and allow access to full interconnect bandwidth by running jobs one at a time on the entire machine. Of course, unless *every* job uses all nodes—which is not the case on any supercomputer we know of—utilization will suffer due to excessive fragmentation of nodes (and links). Alternatively, a scheduler can allow access to full interconnect bandwidth and high utilization by allowing jobs to share network links. This is how current job schedulers on high-performance computing systems operate; the downside is that the sharing of network links violates network isolation. Below, we discuss each of the three goals in more detail.

Isolation. The first constraint, isolation, ensures that jobs do not interfere with each other on the network. This requires that each node *and each link* be (exclusively) assigned to at most one job. Most existing job schedulers ensure only node isolation.

Full interconnect bandwidth. The second constraint, full interconnect bandwidth, ensures that an allocated partition has the bandwidth properties of the fat-tree itself. Specifically, the partition assigned to a job must be *rearrangeable non-blocking*: any permutation of traffic among the nodes of a job can be routed such that only one flow travels over any of the job's links. One consequence of enforcing this constraint is that each leaf or L2 switch must have at least as many uplinks allocated to a job as downlinks (see Figure 1, left). Violating this principle can lead to two flows of an arbitrary permutation being forced to share a link. Another consequence is that we cannot allow full generality in node-to-job assignment (see Figure 1, center). The example shows three leaves with one, two, and three nodes (in blue) assigned to a job, and the left-most down-link into the left-most leaf is shared by two flows. Finally, every link in the allocation must be carefully selected for overall connectivity. Figure 1 (right) shows an example in which balanced uplinks and downlinks have been selected independently at each switch. Although a sufficient number of links is present, some links

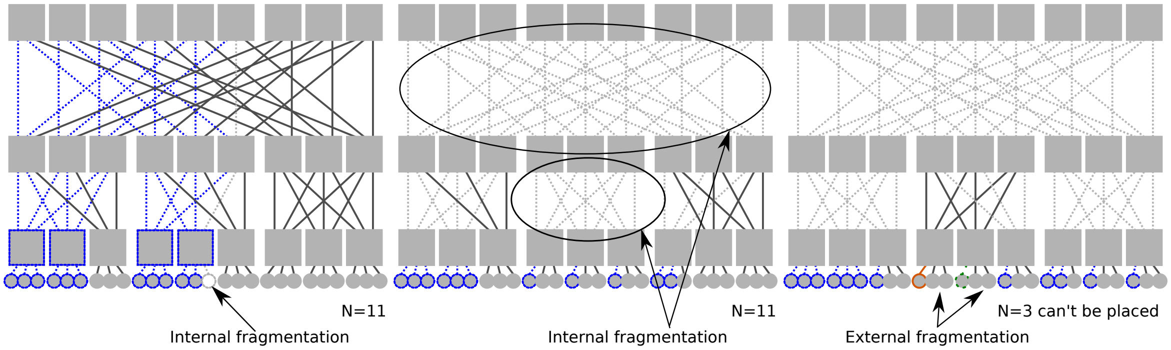

Figure 2: Fragmentation of nodes or links in prior approaches. On the left, internal fragmentation of nodes under the LaaS approach is shown; the wasted node is highlighted grey. In the center, internal fragmentation of links under the TA approach is shown; links are implicitly reserved for the first job that can physically reach them, although in practice the job may not utilize all links. Again, the possibly wasted links are highlighted grey. On the right, external fragmentation of nodes under the TA approach is shown; enough nodes and links are available for a three-node job, but it cannot be placed due to TA's policy.

are wasted as they cannot reach all allocated switches; effectively, the allocation is again tapered as it was in the leftmost example.

The intuition provided by these examples will be formalized in Sections 3.2 and 3.3.

High utilization. While the third goal, high utilization, is not a constraint, it requires allowing sufficient flexibility in assigning nodes and links to jobs. This generally helps in reducing fragmentation of nodes and links. The challenge is to provide high utilization while maintaining the isolation and full bandwidth constraints. No prior research that we know of has been able to do this; typically, prior approaches suffer excessive fragmentation because precise node and link allocation conditions for three-level fat-trees are not developed. This leads to heuristic approaches that simplify scheduling algorithms but lead to wasted nodes or links. Specifically, previous job-isolating scheduling techniques have suffered from internal node fragmentation (LaaS [37]) or both internal link fragmentation and external node fragmentation (TA [19]).

For example, insisting that all nodes on a leaf switch be assigned to a single job, as LaaS [37] does, leads to internal node fragmentation (see Figure 2, left). In addition, insisting that nodes in a job can use any conceivable paths between them, as TA does [19], leads to internal link fragmentation (see Figure 2, center). Finally, requiring that a job must be assigned to a single leaf if it can fit (another part of the TA algorithm) leads to external node fragmentation (see Figure 2, right). In contrast, we develop precise allocation conditions that are both necessary and sufficient for isolation and full interconnect bandwidth. Thus, unlike the existing approaches discussed above, our scheduler can consider every legal placement that is possible for a job at scheduling time.

3.2 Formal Conditions

This section describes our conditions, which lead to the Jigsaw scheduler (described in Section 4) achieving all three goals: isolation, full interconnect bandwidth, and high utilization. We categorize the conditions by the goal to which they contribute.

Before describing the conditions, we note that a three-level fat-tree is composed of a set of independent two-level fat-trees connected together at the third level by spine switches. For brevity, we refer to these two-level subtrees as simply *trees* for the remainder of this section.

3.2.1 Isolation. The conditions needed to ensure isolation follow directly from the description in the previous section. If nodes i and j are assigned to two different jobs, then $i \neq j$; and, if links k and l are assigned to two different jobs, then $k \neq l$. It is clear that these conditions are necessary and sufficient to ensure job isolation.

3.2.2 Full Interconnect Bandwidth. Here, we must constrain node and link assignments; unconstrained node or link assignments can lead to violation of full interconnect bandwidth, as was shown in Figure 1. We list the conditions[1] below; Section 3.3 discusses the proof that the conditions below are necessary and sufficient to guarantee full interconnect bandwidth. Note that we take it as a given that every leaf and L2 switch must be allocated the same number of uplinks as downlinks; Figure 1 (left) showed clearly that without this assumption, tapering of the fat-tree occurs.

(1) The N nodes of a job must be evenly distributed across T trees with n_T nodes each, plus an optional remainder tree with $n_T^r < n_T$ nodes.
(2) Within each of the T trees, the nodes must be distributed across L_T leaves with n_L nodes each. In the remainder tree, they must be distributed across L_T^r leaves that also have n_L nodes each, plus an optional remainder leaf with $n_L^r < n_L$ nodes.
(3) The nodes must be allocated across a set of identical trees and one remainder tree; and all leaves in the allocation must have the same number of nodes except a single remainder leaf in the remainder tree. Note that $N = T \cdot n_T + n_T^r = T(L_T \cdot n_L) + (L_T^r \cdot n_L + n_L^r)$ by these conditions.

[1] Two of these conditions that apply only to *two-level* fat-trees, namely the latter half of (2) and the entirety of (4), were first identified in prior work on LaaS [37].

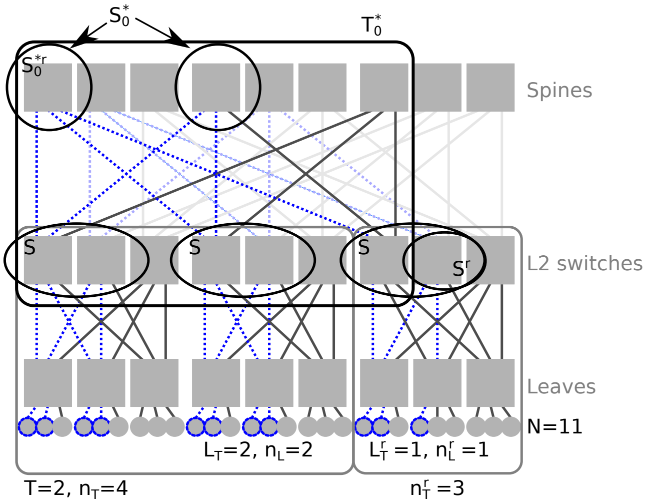

Figure 3: A legal job allocation fitting the formal conditions described in this section.

(4) Within each allocated *two-level* tree, the L_T (or L_T^r) leaves must connect to a common set of L2 switches S, while the remainder leaf must connect to a set of L2 switches $S^r \subset S$.

(5) Each of the allocated trees must use a consistent set of L2 switches S; which means that in every tree, the set S must contain L2 switches at the same set of indices within the tree. Formally, each tree S must contain the L2 switches at i_1, i_2, ..., i_{n_L} for some set of i_k that is common across all trees in the allocation.

(6) At the top level, the i^{th} L2 switch of each tree is part of a full-bipartite graph with a subset of the spines; we denote as T_i^* this set of L2 switches, spines, and links. The partition T_i^* has the same structure as a two-level fat-tree, and so it has conditions similar to (2) and (4). Namely, if the i^{th} L2 switch of an allocated tree is in S, then it must be allocated a balanced number of uplinks to the spines; each such switch must connect to the same set of spines S_i^*, except if it is in the remainder tree. In this case it must connect to a subset of the spines $S_i^{*r} \subseteq S_i^*$.

3.2.3 High Utilization. As mentioned in the previous section, high utilization requires avoiding wasting nodes and links. Two conditions help to achieve this. At the node level, the number of nodes assigned to a job must be exactly the number of nodes that the job requested. Denoting the number of requested nodes by N_r and the number of assigned nodes by N, this means that $N = N_r$. At the link level, every leaf and L2 switch must be allocated the same number of uplinks as downlinks; note that this condition also is part of the set of full interconnect bandwidth conditions, but here it guarantees that the minimum number of links are used.

3.2.4 Summary. Figure 3 shows an example of a legal allocation fitting all node and link conditions, with T_0^* highlighted. The allocation of links to a job must be enforced via modified routing, and the remainder leaf and tree require special care when routing to ensure that all nodes can be reached via allocated links. This is described in Section 4.

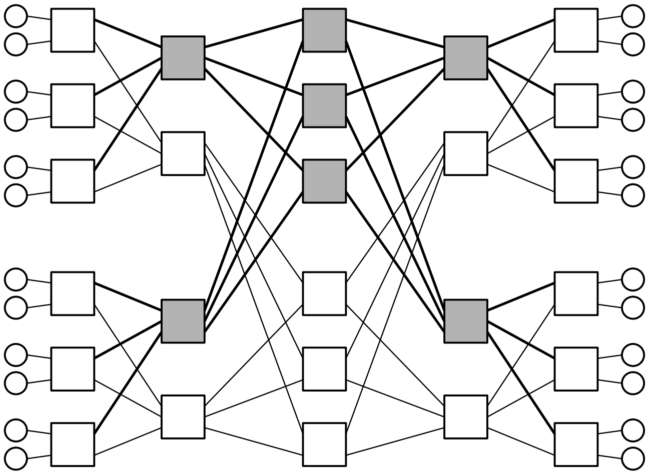

Figure 4: The Clos network equivalent of a three-level fat-tree. Every node in the fat-tree is duplicated as both an input node (left) and an output node (right), and the tree is "unfolded", sideways, to form a Clos network.

3.3 Full Bandwidth Proof Sketch

The node and link allocation conditions fulfill the full interconnect bandwidth property discussed in the motivation above. Specifically, the conditions are both *necessary* and *sufficient* for a job's allocation to be rearrangeable non-blocking. Here, we provide a sketch for the proof of this fact; the full proofs of necessity and sufficiency are too lengthy to be included in the body of this paper, but they are provided in their entirety in the extended version of our paper [31].

Formally, a network is *rearrangeable non-blocking* if, for any permutation of traffic among its nodes, there exists a routing that maps at most one flow of traffic to every link. To prove the necessity of the conditions listed in Section 3.2.2, we show that if each individual condition does not hold, then the allocation cannot support a particular traffic permutation among its nodes without contention. Specifically, we pick two subsets of nodes $\mathcal{A}$ and $\mathcal{B}$ of size n, and we show that if $\mathcal{A}$ sends n flows to $\mathcal{B}$ they will be confined to fewer than n links. Figure 1 showed concrete examples of unavoidable flow conflicts that occur when the conditions are not met; in the extended version of our paper [31], we formally prove conflicts *always* occur in such cases.

To prove the sufficiency of the conditions, we must take an arbitrary partition that satisfies them and show that an arbitrary permutation can be routed with at most one flow per link. To do this, we view the three-level fat-tree as an equivalent Clos network, shown in Figure 4. The Clos network has distinct input nodes (left) and output nodes (right); each level of switches from left to right is referred to as a *stage* of the network. The Clos network that corresponds to a three-level fat-tree has five stages.

The general method of the proof draws inspiration from the existing result that two-level fat-trees are rearrangeable non-blocking. First, we take the entire fat-tree in Figure 4 to illustrate the method.

Suppose that $\mathcal{P}$ is an arbitrary permutation of input nodes to output nodes, so that every input node sends a flow to exactly one unique output node. We use Hall's Marriage Theorem [15] to find a subset of flows in $\mathcal{P}$ such that the subset contains exactly one flow coming from each leaf and one flow going to each leaf. (Hall's Marriage Theorem guarantees that such a subset of flows exists).

We can route all flows in the subset across the same center-stage network (for example, the grey network in Figure 4), with each link at the first and last stage carrying exactly one of the flows. Now, notice that the center-stage network is itself a smaller Clos network, equivalent to a two-level fat-tree. It is already known that a two-level fat-tree is rearrangeable non-blocking, so the flows can be routed with at most one per link across the center-stage network as well [3]. Thus, this subset of flows is satisfactorily routed across the three-level fat-tree.

We then remove the subset of flows and the links used from the network, leaving behind only the white switches in Figure 4. The remaining portion of the network forms a (smaller) three-level fat-tree, and the remainder of $\mathcal{P}$ forms a permutation of its input nodes to its output nodes. Thus, we repeat the previous steps until every flow in $\mathcal{P}$ has been routed.

Now, to extend this proof to a partition satisfying our allocation conditions, we notice that such a partition has a form quite close to a full three-level fat-tree. Given the conditions on node placement and link allocation, we can actually augment the partition to form a complete three-level fat-tree by simply adding additional nodes to the remainder leaf and additional leaves to the remainder tree (along with corresponding additional links). Thus, to extend the proof to such a partition A, we augment A as just described, and we augment the permutation $\mathcal{P}$ over A with additional flows between the added nodes. Then, we apply the same method of proof while taking care to route the flows in $\mathcal{P}$ across links actually allocated to A.

Thus, we have formalized placement conditions that are both necessary and sufficient for the resulting partition to have bandwidth equivalent to a full fat-tree network. No looser conditions can guarantee partitions that satisfy this property, and tighter constraints like those used in previous work will necessarily exclude some possible placements. In the next section, we use these constraints to create a novel job-isolating scheduling algorithm that outperforms previous approaches.

4 JIGSAW IMPLEMENTATION

We now describe Jigsaw, a new job-isolating scheduling approach that leverages the theory developed in the previous section. Viewed from a high level, Jigsaw simply enforces the conditions described in the previous section, so that Jigsaw always allocates isolated partitions, and those partitions have full interconnect bandwidth.

However, Jigsaw also must achieve high utilization. It turns out that permitting *any* allocation satisfying the formal conditions in Section 3.2 is too permissive. First, the conditions we found allow a wide variety of legal allocations—the number of possible allocations is exponential in the size of the tree, making an exhaustive search for placements infeasible for large systems. Even more concerning, being maximally permissive actually leads to excessive *external* fragmentation; this is because each allocation is permitted to use a

Algorithm 1 The Jigsaw allocation routines. Once a node and link allocation is found, the resources are assigned to the job and the system routing is modified to route the job's traffic within its allocation only (see Figure 5).

```
func GET_ALLOCATION(size)
    $\mathcal{T}$ ← all two-level trees; $\mathcal{P}_{L3}$ ← all L2 up-ports
    $\mathcal{L}$ ← all leaves per tree; $\mathcal{P}_{L2}$ ← all leaf up-ports per tree
    $A$ ← an empty allocation to be populated below
    ▷ Look for a single-subtree allocation first.
    for $L_T, n_L, n_L^r$ s.t. $L_T \cdot n_L + n_L^r = size$ do
        for $t$ in two-level trees do
            if FIND_L2($A, L_T, n_L, n_L^r, \mathcal{L}[t], \mathcal{P}_{L2}[t]$) then
                return $A$
    ▷ Look for a three-level allocation if two-level failed.
    $n_L$ ← all nodes per leaf                              ▷ Jigsaw condition
    for $T, n_T, n_T^r$ s.t. ($T \cdot n_T + n_T^r = size$ & $n_L | n_T$) do
        $L_T \leftarrow \frac{n_T}{n_L}$
        $sltns$ ← an empty list of solutions per tree
        for $t$ in two-level trees do (*)            ▷ find two-level sltns
            $A'$ ← a temporary empty allocation
            FIND_ALL_L2($A', sltns, L_T, n_L, 0, \mathcal{L}[t], \mathcal{P}_{L2}[t]$)
        if FIND_L3($A, sltns, T, n_T^r, \mathcal{T}, \mathcal{P}_{L3}$, any sltn) then
            return $A$
    return no allocation found

▷ Note: FIND_L2 is similar but returns first solution found.
func FIND_ALL_L2(ref $A$, ref $sltns, L_T, n_L, n_L^r, \mathcal{L}^t, \mathcal{P}_{L2}^t$)
    for $l$ in $\mathcal{L}^t$ do
        if $l$ has $n_L$ nodes and $n_L$ free links in $\mathcal{P}_{L2}^t$ then
            update $A$ with $l$                        ▷ use $l$ in allocation
            update $\mathcal{L}^t$                    ▷ start from next leaf after $l$
            update $\mathcal{P}_{L2}^t$              ▷ remove links not free for $l$
            if $A$ contains $L_T$ leaves then                      ▷ base case
                if $n_L^r = 0$ or find remainder leaf then
                    add $A$ to $sltns[t]$ (**)
            else                                                     ▷ recurse
                FIND_ALL_L2($A, sltns, ..., \mathcal{L}^t, \mathcal{P}_{L2}^t$)
            undo all updates

func FIND_L3(ref $A, sltns, T, n_T^r, \mathcal{T}, \mathcal{P}_{L3}, L2sltn$)
    for $t$ in $\mathcal{T}$ do
        for $sltn$ in $sltns[t]$ do
            if ($sltn$ has same L2/ports as $L2sltn/\mathcal{P}_{L3}$) then
                update $A$ with $t, sltn$                ▷ use $t$ in allocation
                update $\mathcal{T}$                   ▷ start from next tree after $t$
                update $\mathcal{P}_{L3}$              ▷ remove links not fitting $sltn$
                update $L2sltn$                    ▷ set to $sltn$ if $t$ is first tree
                if $A$ contains $T$ trees then                      ▷ base case
                    if $n_T^r = 0$ or find remainder tree then
                        return true
                else                                                 ▷ recurse
                    if FIND_L3($A, sltns, ..., \mathcal{P}_{L3}, L2sltn$) then
                        return true
                undo all updates
    return false
```

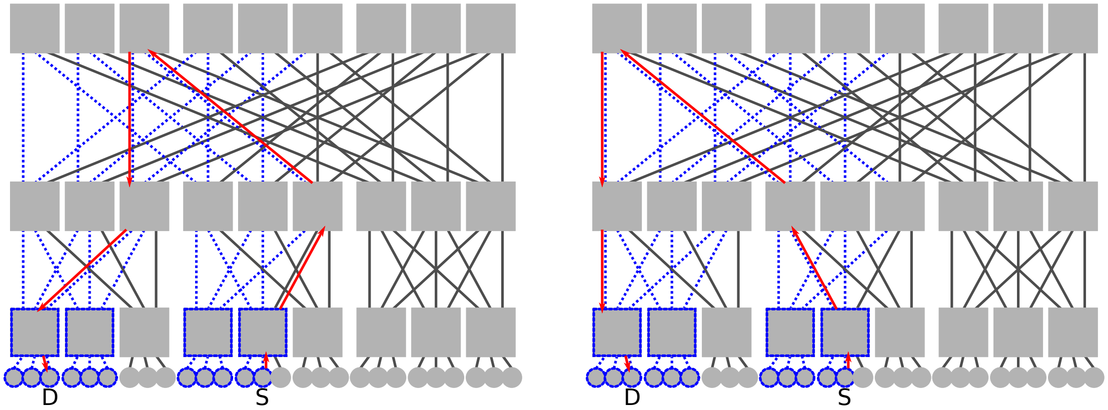

Figure 5: Changing the routing in Jigsaw; shown is a single packet routed from a source node (S) to a destination node (D). The left-hand figure shows that using traditional D-mod-k routing, the first hop (in black with a red arrow next to it) occurs on a link that is not allocated to the job. The right-hand side shows Jigsaw's routing, which uses a wraparound approach to ensure that only allocated links (in blue) are used.

different number of nodes per leaf, leading to an arbitrary number of free nodes on each leaf scattered across the machine. For larger jobs requiring N nodes, the probability that the scattered free nodes are (legally) usable becomes quite small. This tends to require the system to have to drain far more than N nodes before a placement can be found, leading to periodic dips in utilization. Similarly, there may be enough links available for a node allocation, but they do not connect to common L2 switches or spines.

Jigsaw handles both of the above issues simultaneously with one restriction to our set of formal conditions. Specifically, Jigsaw requires job allocations that span three levels to use all nodes per leaf except for the remainder leaf. This restriction reduces the number of placements to search (making the Jigsaw running time fast; see Section 6.4) as well as reducing the external fragmentation due to scattering of free nodes across the machine.

Pseudocode for Jigsaw is given in Algorithm 1 (see previous page). The function `GET_ALLOCATION` looks for a Jigsaw-compliant node and link allocation for a job requesting *size* nodes. Jigsaw first looks for two-level (single-subtree) allocations.[2] If the job cannot be allocated in a single subtree, `GET_ALLOCATION` moves on to look for three-level allocations instead. In this case the allocated nodes are divided across subtrees, with each subtree containing a valid two-level allocation for its subset of nodes. Thus we must find all possibilities for valid two-level allocations of subsets of nodes **(*)** via `FIND_ALL_L2`. Once these are stored in *sltns*, we look for a global combination of sub-allocations in `FIND_L3`.

The functions `FIND_ALL_L2` and `FIND_L3` are analogous in structure. In `FIND_L3` we use an exhaustive, recursive-backtracking search for a valid combination of sub-allocations within trees. Jigsaw looks for the next tree with a sub-allocation that fits the global solution—namely, the tree must contain an allocation using the right set of L2 switches and having appropriate L2 up-ports available to connect to the other trees. If enough matching trees are found, the solution is immediately returned (base case). Note that `FIND_ALL_L2` follows the same structure, but since it is searching for *all* valid allocations, it adds each completed allocation to *sltns* and continues the search **(**)**, rather than immediately returning.

Finally, once Jigsaw returns an allocation, the routing tables must be adjusted. This is necessary because traffic must be routed only on links within the allocation, and standard D-mod-k routing is completely unaware of the Jigsaw allocation (see Figure 5). A valid adjusted routing can easily be obtained by mapping normal D-mod-k routing onto the partition and using wraparound for ports on remainder switches. The actual changing of the routing tables can be done on the fly, for example via the subnet management software on an InfiniBand system, as shown in prior work [10].

In a wide variety of tests, Jigsaw proves to achieve high utilization and throughput with scalable scheduling time on the large clusters. The next section describes our evaluation in detail.

5 EVALUATION SETUP

This section describes our experimental infrastructure and evaluation setup. All of our code, traces, and results are available in a repository [32].

To evaluate the different scheduling approaches, we simulate job queues on various fat-trees. We compare Jigsaw to several other job-isolating scheduling approaches described in this section, and also to *Baseline*: a traditional, unconstrained scheduler. We evaluate all approaches in terms of average system utilization, given by

$$U = \frac{\text{time spent by jobs on all nodes}}{\text{potential time on all nodes}} = \frac{\sum_{j=1}^{\text{jobs}} N_j \cdot t_j}{N_{\text{system}} \cdot t_{\text{total}}}.$$

Here N_j and N_{system} are the number of nodes in job j and the total number of nodes in the system, respectively; t_j and t_{total} are the runtime of job j and the total time spent running the jobs in the trace, respectively. Because clusters are not drained and restarted often, for realism we include only the steady-state portion of the simulation in this computation.

In addition, we evaluate performance in terms of job turnaround times and system throughput. Job *turnaround time* is the length of time between when a given job arrives in the queue and when it

[2] As LaaS shares a few conditions with Jigsaw, its algorithm is similar up to here.

finishes running. Thus, turnaround time is related to wait time (the time between arrival and beginning to run), but also encompasses the performance of the job once it is running. We measure throughput via *makespan*, which is the length of time between when the first job arrives in the queue and when the last job running on the system completes. Shorter turnaround times and makespan are both desirable.

5.1 Clusters and Traces

We evaluate on a range of full (maximal-size) fat-trees, where we vary the switch radix to attain different node counts. We run experiments with four different cluster sizes: 1024, 1458, 2662, and 5488 nodes (built with radix-16, radix-18, radix-22, and radix-28 switches, respectively).

We use job queue traces from several LLNL clusters, including Thunder and Atlas [12] as well as Cab [33]. We also use synthetic traces generated in the same way as the ones in the original LaaS paper [37]; these are modeled on a trace from the Julich JUROPA cluster. Because we generate the synthetic traces, they allow us to effectively scale our experiments to different-sized fat-trees.

In the synthetic traces, the job sizes are drawn from an exponential distribution, and the job run times are drawn from a uniform random distribution. In the traces from HPC clusters, the job size distribution is roughly exponential in shape but contains more job sizes that are powers of two; the job run times are skewed towards short-running jobs with only a handful of long-running jobs.

The synthetic traces have all jobs arriving at time zero. For the Thunder and Atlas traces, we discard actual job arrival times, making all jobs available at time zero for these traces as well. Thus we can test the scheduling approaches under heavy cluster usage; we are not particularly interested in cases where the system utilization is low due to a lack of pending jobs. However, to analyze turnaround time effectively, it is necessary to run experiments with realistic job arrival times as well. Therefore we retain the arrival times in the Cab traces, though we scale arrival times by 0.5 in the Aug-Cab and Nov-Cab traces because of low baseline utilization in those months. Table 1 contains a summary of the traces.

Table 1: Characteristics of job queue traces used in our evaluation. All traces contain single-node jobs.

Trace name	System nodes	Number of jobs	Max job nodes	Job run times (s)	Arrival times
Synth-16	–	10,000	138	20-3000	N
Synth-22	–	10,000	190	20-3000	N
Synth-28	–	10,000	241	20-3000	N
Aug-Cab	1296	30,691	257	1-86,429	Y
Sep-Cab	1296	87,564	256	1-57,629	Y
Oct-Cab	1296	125,228	258	1-93,623	Y
Nov-Cab	1296	50,353	256	1-86,426	Y
Thunder	1024	105,764	965	1-172,362	N
Atlas	1152	29,700	1024	1-342,754	N

5.2 Schemes for Comparison

We compare Jigsaw to the following approaches, in addition to Baseline. The first two are existing approaches, while the last is a theoretical bounding approach of our own.

5.2.1 Links as a Service (LaaS). In Links as a Service (LaaS) [37], the scheduler allocates dedicated network links (and nodes) to each job. Unlike Jigsaw, LaaS does not develop explicit placement conditions for three-level fat trees; instead it reduces the three-level problem to a two-level problem and leverages easier-to-identify conditions for *two levels*. Specifically, to reduce three levels to two levels, entire leaves take the place of nodes, L2 switches take the place of leaves, and spines take the place of L2 switches. The drawback of the LaaS approach is that this forces job sizes to be rounded up to the nearest multiple of the leaf size, causing internal fragmentation (see Figure 2, left).

5.2.2 Topology-Aware Scheduling (TA). Unlike LaaS and Jigsaw, in the topology-aware scheduling (TA) [19] approach, links are not explicitly allocated to jobs. Instead, TA follows a set of node-allocation rules that avoid all placements in which two jobs could conceivably contend for links under an arbitrary routing. The full set of rules involves containing small-enough jobs within single leaves, larger jobs within single subtrees, and only jobs larger than a subtree across the entire machine; a job of a given type will not be able to share leaves or subtrees with other jobs of certain types. The drawbacks of the TA approach are internal fragmentation due to the implicit allocation of links (see Figure 2, center), along with external fragmentation caused by the requirement that jobs must be assigned to single leaves and/or single subtrees, if possible (see Figure 2, right).

5.2.3 Least-Constrained Scheduling with Link-Sharing (LC+S). Jigsaw only uses a subset of the possible legal placements determined by the conditions we developed in Section 3.2. As we explained in Section 4, allowing the full set of possible placements (i.e., the least possible constraints) actually produces lower utilization than Jigsaw because of external fragmentation of nodes and links. Thus, to reduce link fragmentation, we add a relaxation to the least constrained approach that allows jobs to share network links as much as possible while maintaining interference levels that are expected to be negligible (though not zero). Specifically, given a job j that requires, on average, $x\%$ of the cluster's peak bandwidth per link, we search for and allocate that amount of bandwidth, leaving the rest of the link available to future jobs.

Clearly, this approach is of theoretical interest only; it is impractical for real systems because the required bandwidth for each job is not provided to the job scheduler. But in our evaluation, we suppose that this information is made available.

We note that this relaxation can also be combined with LaaS or Jigsaw, but we find that a least constrained approach benefits the most from additional flexibility in link allocation. Thus, we include for comparison *least constrained with link sharing* (denoted LC+S) as a theoretically near-optimal, low-interference scheduler.

5.3 Implementation Details

We have implemented all scheduling approaches inside the original code base released with LaaS [37]. In addition, we have added backfilling capability to the scheduling simulator (which initially supported only FIFO scheduling). We use the EASY backfilling approach, in which a single job at the head of the queue is assigned a reservation when backfilling occurs, and as many jobs as possible in the lookahead window are backfilled [29]. EASY backfilling is available in modern resource managers such as Flux [2].

As seen in Algorithm 1, the Jigsaw implementation uses a brute-force search for a valid job placement, returning the first allocation found. The Jigsaw search is fast in practice, as are the LaaS and TA searches. While the search space for LC+S is much larger, a valid allocation for LC+S is *usually* found quickly; however, its worst case search time for a single job allocation can be hours (which typically occurs when no allocation currently exists). Therefore, in our LC+S implementation we guard against the worst case by adding a timeout to each job scheduling event.

5.4 Experiment Parameters

Finally, we describe specific parameters used in our experiments. We address, in turn, the performance improvement scenarios for jobs run in isolation (used in our turnaround time and makespan analysis); the bandwidth needs of jobs (used for the LC+S experiments); and miscellaneous parameters used across experiments.

5.4.1 Job Performance Scenarios. When we evaluate job turnaround times and makespan, we take into account the fact that some jobs will likely perform better when run in isolation. For each job trace, we explore different job performance scenarios, each with its own set of assumptions about which jobs will perform better and by how much. We use four of the scenarios from the TA scheduling paper [26], namely the 5%, 10%, 20%, and “V2" scenarios. In addition, we create one scenario of our own, which we call *Random*, that is *less* optimistic about performance improvement than the above scenarios. In the $x\%$ scenarios, each job larger than four nodes speeds up by $x\%$ under job-isolating scheduling approaches compared to Baseline. In the V2 scenario, jobs are randomly assigned to different speed-up buckets depending on their size, with a minimum possible speed-up of 0% and a maximum possible speed-up of 30%; within a bucket, the amount of speed-up scales linearly with job node count (see the TA paper for details [26]). Finally, in our Random scenario, only jobs larger than 64 nodes ever speed up, and each such job randomly speeds up by either 0%, 5%, 15%, or 30%.

5.4.2 Link-Sharing Parameters. For the LC+S experiments, we have to determine average bandwidth needs for each job in our traces. For our evaluation, we randomly assign jobs in the traces to one of four classes with bandwidth needs ranging from 0.5 GB/s per link to 2.0 GB/s per link. We assume a peak link bandwidth of 5 GB/s for the cluster. Based on existing empirical evidence, we cap total utilization of each link at 80%, above which performance degradation is expected to increase sharply [30].

5.4.3 General Parameters. Finally, we select general parameters for all the experiments. The three synthetic traces have mean job sizes of 16, 22, and 28, and we simulate them on 1024-, 2662-, and 5488-node clusters, respectively. We simulate the Thunder, Atlas, and Cab traces on the 1458-node cluster. This is the smallest of our experiment clusters that is larger than all three of Thunder, Atlas, and Cab. Note that we could also have chosen the 1024-node cluster given the maximum job sizes in the three traces. However, that cluster has eight nodes per leaf, meaning the leaf size evenly divides many of the job sizes in the Thunder, Atlas, and Cab traces. In general, this is not true for most clusters and job traces—but it slightly improves system utilization for LaaS, which would skew our evaluation (although Jigsaw still outperforms LaaS even in this case). Therefore, we evaluate on the 1458-node cluster for the sake of generality. Finally, we use a backfill window of 50 in our experiments, and we use a timeout of 5 seconds for LC+S.

6 RESULTS

In this section we evaluate the different job-isolating scheduling approaches—TA scheduling (denoted TA), LaaS, and Jigsaw—and include for comparison the (unrealistic) bounding approach LC+S. In the results presented here, all approaches (including Baseline) use EASY backfilling.

We find that Jigsaw outperforms existing job-isolating scheduling approaches on system utilization, job turnaround time, and makespan. It achieves system utilization that is usually within 3-5 percentage points of Baseline's utilization; and because its utilization is close to Baseline, Jigsaw often leads to better job turnaround times and makespan than Baseline under modest assumptions of improved job performance in isolated placements. We also examine average job scheduling time for each approach, finding that Jigsaw is fast in practice and therefore a practical choice for large clusters.

6.1 Average System Utilization

We first consider average system utilization, a metric which needs to be at or above 95% under sufficient demand for the cluster in order for many HPC system administrators to consider adopting a given scheduling scheme [9]. Figure 6 shows the average system utilization with different scheduling approaches for each of the job queue traces in Table 1.

Across traces, average system utilization with Jigsaw is typically 95-96%, versus 97-100% with Baseline. The only exceptions are Oct-Cab and Atlas, where Jigsaw utilization is only 93% and 92%, respectively. The worst case utilization under Jigsaw (and in fact all schemes, including Baseline) is on Atlas, because there were several whole-machine job requests.

We see that Jigsaw performs significantly better than both TA and LaaS, the existing approaches to job-isolating scheduling. On the traces we tested, which include those used in the TA paper and generated from the same distributions as the LaaS paper, TA resulted in the lowest system utilization at 85-88%. Because TA does not explicitly allocate links, its placement choices are the most constrained, and it suffers the most from fragmentation. For example, a small job that fits within a leaf *must* wait for enough nodes to be available on a single leaf, whereas the same small job can be spread over multiple leaves with fewer nodes when using Jigsaw.

LaaS's utilization comes next after TA's at 90-91% on all traces except one at 93%; this utilization is well below Baseline and also below the desired 95% threshold. LaaS's utilization is lower than

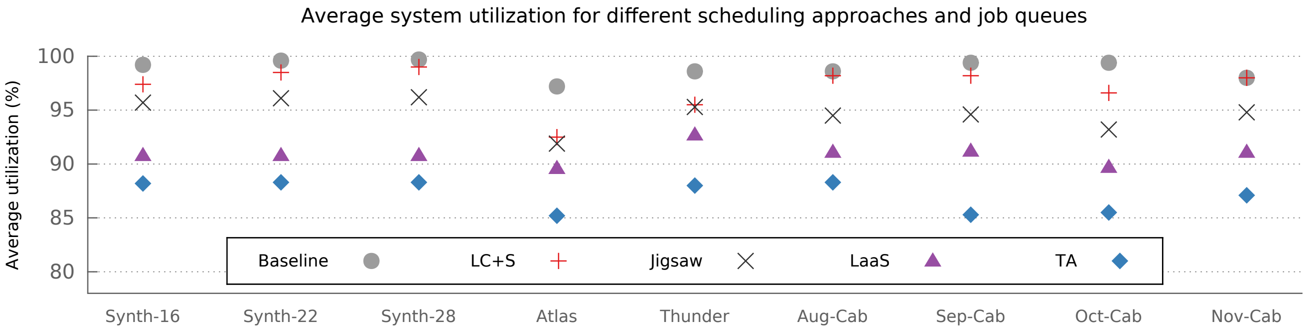

Figure 6: Average system utilization for different approaches and traces. The synthetic, Atlas, and Thunder traces have all jobs arriving at time zero for heavier load; the Cab traces use real job arrival times. The y-axis starts at 80% so that significant differences between approaches can be seen clearly and because lower utilizations never occur in our experiments.

Table 2: Frequency of instantaneous percentage utilization ranges for a trace of ~100,000 jobs on Thunder.

Approach	>=98	95-97	90-95	80-90	60-80	<=60
LaaS	4133	53720	78072	52546	21261	1717
Jigsaw	54186	51789	50299	36586	17086	1503
TA	20443	27134	46923	68262	42923	5764

Jigsaw's because of internal fragmentation of free nodes and links on the last leaf in the LaaS allocation (see Figure 2). In our experiments, the nodes lost to internal fragmentation account for 3-7% of the system on average, while Jigsaw does not incur this penalty.

We see that LC+S often leads to higher utilization than Jigsaw, however. This is because with the (unrealistic) addition of job-level link usage information along with link sharing, jobs only take as much of each link as they need so that external fragmentation of links is minimized. We see that Jigsaw achieves the same utilization as LC+S on some traces, with a maximum difference of 3.7 percentage points on Aug-Cab. Thus, Jigsaw's utilization is probably approaching the limit of what is achievable with complete isolation.

Analyzing the results in depth, Table 2 shows frequency of instantaneous utilization values for each of the three isolated scheduling approaches during simulation of the Thunder trace, where instantaneous utilization at time t is the number of nodes allocated at t divided by the total number of nodes in the system. For Table 2, we measure instantaneous utilization each time a job is scheduled or completed. Under Jigsaw, instantaneous utilization is at or above 98% for roughly a quarter of the data points, versus only a tenth of the data points under TA and virtually none of the data points under LaaS. In addition, instantaneous utilization under Jigsaw is below 80% slightly less than a tenth of the time (which is comparable to LaaS), versus almost a quarter of the time for TA. The reason that LaaS almost never achieves instantaneous utilization above 98% is that about 3% of system nodes are typically lost to internal fragmentation (allocated to jobs that do not need them) throughout the simulation. This limits average utilization under LaaS. While TA does not suffer from internal fragmentation, we can see that it experiences significant external fragmentation from the fact that utilization drops below 80% much more often than it does under LaaS or Jigsaw. This external fragmentation, which occurs because of the relatively inflexible limitations on where a job can be placed, limits the average utilization under TA. Because Jigsaw is built on the formal placement conditions we developed for three-level fat-trees, it is able to lessen the impact of external fragmentation and fully eliminate internal fragmentation, leading to the favorable utilization results shown in this section.

6.2 Average Job Turnaround Time

We now consider another metric for scheduler performance: average job turnaround time. Figure 7 shows the average job turnaround time for all jobs and for large jobs (greater than 100 nodes), respectively. The averages are normalized to the averages under Baseline. The first cluster of bars shows the worst case, in which no jobs experience any performance improvement from running in isolation. However, many studies have observed performance degradation of applications due to inter-job interference on the network [6–8, 30]. Therefore we focus on turnaround times under the different performance-improvement scenarios that were given in Section 5. The first three scenarios involve jobs larger than four nodes speeding up by a fixed amount (5, 10, or 20%) each, when run in isolation. The last two scenarios are less optimistic in that some jobs do not speed up at all, and the amount of speed up is randomized.

We show results for the Aug-Cab and Oct-Cab traces—Sep-Cab and Nov-Cab performed similarly to Aug-Cab, while Oct-Cab is the worst case (for all metrics, including utilization). For Aug-Cab, the average turnaround time with Jigsaw is better than Baseline in every speed-up scenario (see the filled portion of the bars). However, large jobs typically have to wait longer than smaller jobs due to backfilling, so in some scenarios the average turnaround time for large jobs remains above Baseline (see the additional empty portion of the bars). Specifically, the 10% and 20% speed-up scenarios are sufficient for Jigsaw to beat Baseline on large jobs, while in other scenarios average turnaround time for large jobs is a few percent higher.

For Oct-Cab, Jigsaw beats Baseline on average turnaround time in the 10% and 20% speed-up scenarios, while it is just 1-3% higher in the other scenarios. While large jobs typically wait longer, they

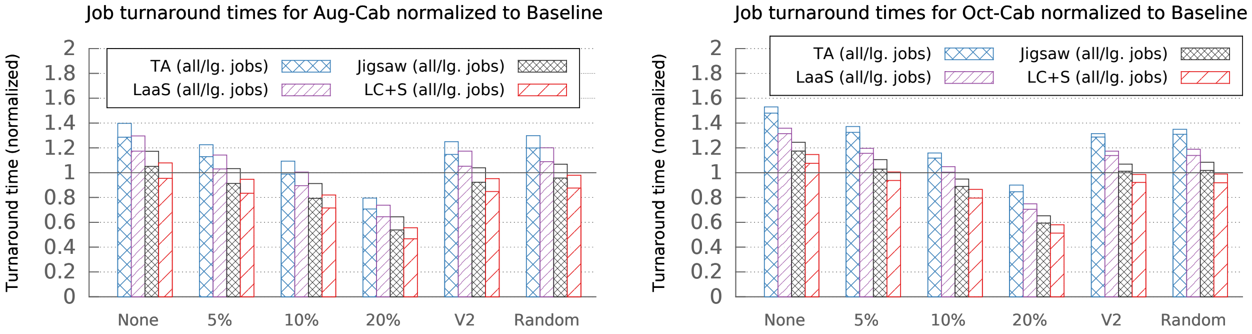

Figure 7: Average turnaround times (normalized to baseline; lower is better) for two traces with different assumptions on job performance under isolation. The filled portion of each bar shows the average for all jobs; the full bar shows the average for larger jobs only (those over 100 nodes). The first scenario is no performance improvements (the worst case). The next three scenarios assume jobs with more than four nodes all speed up by 5%, 10%, or 20%, respectively. The V2 scenario assumes jobs speed up linearly with node count based on a random assignment. The Random scenario assumes jobs over 64 nodes speed up by 0%, 5%, 15%, or 30% at random.

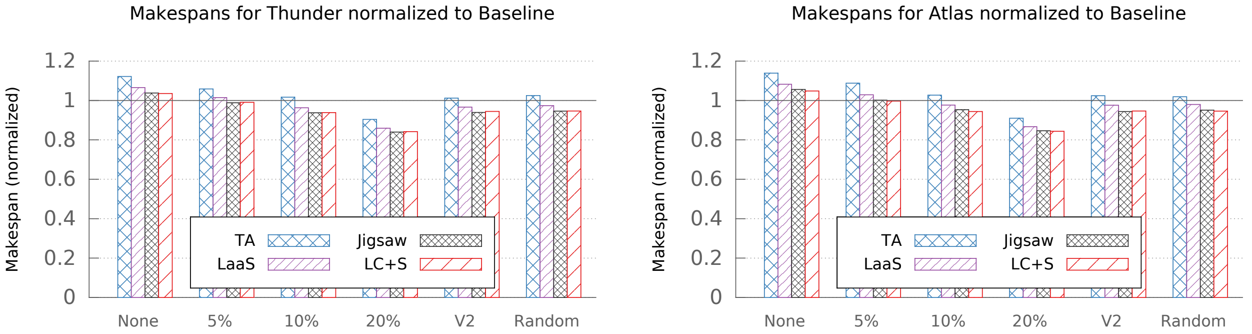

Figure 8: Makespans normalized to Baseline for two traces (lower is better). The speed-up scenarios are the same as Figure 7.

are also better than Baseline in the 10% and 20% scenarios; in others they are at most 10% worse. Again, Oct-Cab was the worst case trace for Jigsaw, yet Jigsaw still nearly meets and sometimes beats Baseline in the speed-up scenarios.

For both traces, we can see that Jigsaw turnaround times are always considerably better than TA turnaround times. Jigsaw turnaround times are also better than LaaS turnaround times; for example, on Oct-Cab with the 10% speed-up scenario, normalized LaaS turnaround times are 1.00 for all jobs and 1.05 for large jobs, versus 0.89 and 0.95 for Jigsaw.

6.3 Makespan

Finally, in Figure 8, we show makespan (normalized to Baseline) for the different approaches on the Thunder and Atlas traces. Under all speed-up assumptions, makespan with Jigsaw is the same as or better than Baseline—by up to 15%. In contrast, under TA the makespan is always worse than Baseline except for the 20% speed-up case, and LaaS is between TA and Jigsaw in effectiveness. In the worst case (with no speed-ups), makespan under Jigsaw is only 6% higher than Baseline, versus 5% higher for LC+S, 8% higher for LaaS, and 14% higher for TA. Therefore, Jigsaw has a minimal impact on makespan even in the worst case, and it achieves lower makespan than Baseline in scenarios where jobs experience performance benefits from job-isolating scheduling. It also produces similar makespan to the (unrealistic) LC+S bounding algorithm.

Table 3: Average scheduling time per job in seconds.

	Synth-16	Sep-Cab	Thunder	Synth-28
TA	0.00283	0.00252	0.00128	0.00709
LaaS	0.00294	0.00257	0.00106	0.00947
Jigsaw	0.00305	0.00283	0.00104	0.00998
LC+S	0.05550	0.09239	0.05333	0.25500

6.4 Scheduling Time

Table 3 summarizes scheduling times for four representative experiments on clusters, from smallest to largest. We can see that Jigsaw scheduling time is on a par with existing approaches (TA and LaaS), and it scales well to high node counts—up to 5488 nodes in our tests, which represents a system larger than Sierra and Summit, two of the top three HPC systems as of November 2020 [24]. In the worst case, Jigsaw's average scheduling time per job is only 10 milliseconds.

In contrast, LC+S, as the least-constrained, near-optimal approach, does not scale as well with cluster size. In the worst case, on the 5488-node cluster, average scheduling time per job for LC+S is 255 ms, an order of magnitude larger than worst-case Jigsaw time (and recall that LC+S requires a 5-second timeout for scheduling per job in order to be feasible, while Jigsaw does not). Thus, given the sharp increase in scheduling time from radix-18 to radix-28, it is not clear how much farther LC+S will scale in terms of system size.

Therefore, we see that Jigsaw finds job placements just as quickly as existing job-isolating approaches, while also beating them on all performance metrics. In contrast to LC+S, which must search the largest possible solution space, Jigsaw scales well up to more than 5000 nodes and likely far beyond. Therefore we find that Jigsaw would be practical for immediate deployment on even the largest fat-tree based systems.

7 RELATED WORK

The focus of this paper is on using scheduling policies to reduce or eliminate inter-job network interference. As we have discussed, inter-job network interference can be completely eliminated by using a scheduling policy that guarantees isolated partitions to each job. On fat-trees, two previous approaches to network isolation exist. As shown in this paper, our work improves on both existing approaches: Links as a Service (LaaS) [37] and topology-aware scheduling (TA) [19, 26].

As mentioned in Section 2, there are multiple previous techniques developed to handle *intra*-job interference, which can be used once a job-isolating scheduler removes the possibility of inter-job interference. One well known technique is topology mapping [5, 13, 17], in which the communication graph is assumed to be known and then the mapping of process-to-node is done so as to reduce the frequency and volume of communication. Another is communication scheduling [11, 25], in which collective operations are staged so that contention during the collective is naturally minimized.

Several papers have observed that network hotspots can occur on fat-trees under static routing, and they have suggested various ways to minimize hotspots by updating the routing as running jobs change [10, 22, 30]. Lee et al. proposed setting up routes for the traffic of each job as it enters the system, using the least congested paths available [22]. Domke et al. proposed Scheduling-Aware Routing (SAR) [10], which balances all potential flows in the system upon job exit or entry, based on the insight that pairs of nodes will only communicate if they belong to the same job. Our prior work proposed Adaptive Flow-Aware Routing (AFAR) [30], which detects and alleviates hotspots by using knowledge of the actual flows of traffic between nodes. All three approaches require a global controller that periodically updates system routing based on the current state (i.e. software-defined networking), and most require information or system software support that is not currently available. None of the approaches provide guarantees on worst-case interference.

Finally, several have studied the magnitude of performance degradation due to inter-job network interference. On the fat-tree interconnect, Jain et al. studied the impact of tapering and other network configurations on job performance and found that inter-job network contention can slow down applications by as much as 66% in simulations [18]. Our prior work observed that communication intensive benchmarks slow down by as much as 120% in controlled experiments on a 1296-node three-level fat-tree [30]. Alongside fat-trees, the dragonfly interconnect [21] is a popular choice for HPC clusters, so many studies of inter-job interference have focused on dragonfly as well, both on real systems and in simulation [4, 7, 14, 27, 30, 35].

8 CONCLUSION

In this paper we have addressed the problem of inter-job network interference on HPC clusters. Researchers have demonstrated that this type of interference leads to system performance degradation and individual job performance variability [6–8, 30]. Most existing mitigation techniques focus on using routing to mitigate the effects of interference, but it is also possible to leverage scheduling policies that guarantee dedicated network partitions to jobs. The downside of all such existing scheduling approaches is lowered system utilization, which discourages widespread adoption.

Thus, in this paper we have designed and implemented Jigsaw, the first interference-free scheduler for three-level fat-trees to achieve system utilization at or above 95% under most workloads. Because system utilization under Jigsaw is close to that of traditional scheduling, we find that Jigsaw typically leads to improvements in job turnaround time and throughput when job performance improvements are taken into account. In addition, Jigsaw is simpler to implement than existing routing schemes that aim to mitigate interference, while providing guaranteed freedom from interference. Since each job is guaranteed network isolation, application developers can focus their efforts on optimizing intra-job network performance without worrying about network traffic outside their control, and performance variability due to inter-job network interference is eliminated. Finally, Jigsaw schedules jobs quickly on clusters of 5000 nodes or more, making its use on existing HPC systems both practical and beneficial. Our work shows that the use of Jigsaw on fat-tree clusters greatly improves system performance at little cost to utilization.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1526015. We are also indebted to the following people for their helpful feedback: Abhinav Bhatele, Bronis de Supinski, Kate Isaacs, Nikhil Jain, Michelle Strout, Xin Yuan, and the anonymous reviewers. In addition, Stephen Herbein provided us with the Cab traces used in our experiments.

REFERENCES

[1] N. R. Adiga, G. Almasi, Y. Aridor, R. Barik, D. Beece, R. Bellofatto, G. Bhanot, R. Bickford, M. Blumrich, and A. A. Bright. 2002. An Overview of the Blue Gene/L Supercomputer. In *Supercomputing*.

[2] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I. IngÃşlfsson, Joseph Koning, Tapasya Patki, Thomas R.W. Scogland, Becky Springmeyer, and Michela Taufer. 2020. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems* 110 (2020), 202–213.

[3] Vaclav E. Benes. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic* (1st ed.). Academic Press.

[4] Abhinav Bhatele, Nikhil Jain, Yarden Livnat, Valerio Pascucci, and Peer-Timo Bremer. 2016. Analyzing Network Health and Congestion in Dragonfly-based Systems. In *International Parallel and Distributed Processing Symposium*.

[5] Abhinav Bhatele and Laxmikant V Kale. 2008. Application-specific topology-aware mapping for three dimensional topologies. In *International Symposium on Parallel and Distributed Processing*.

[6] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. 2013. There goes the neighborhood: performance degradation due to nearby jobs. In *Supercomputing*.

[7] Abhinav Bhatele, Jayaraman J. Thiagarajan, Taylor Groves, Rushil Anirudh, Staci A. Smith, Brandon Cook, and David K. Lowenthal. 2020. The Case of Performance Variability on Dragonfly-Based Systems. In *International Parallel and Distributed Processing Symposium*.

[8] Sudheer Chunduri, Kevin Harms, Scott Parker, Vitali Morozov, Samuel Oshin, Naveen Cherukuri, and Kalyan Kumaran. 2017. Run-to-run Variability on Xeon Phi Based Cray XC Systems. In *Supercomputing*.

[9] Bronis de Supinski. 2020. Personal Communication. (2020).

[10] Jens Domke and Torsten Hoefler. 2016. Scheduling-Aware Routing for Supercomputers. In *Supercomputing*.

[11] A. Faraj and X. Yuan. 2005. Message Scheduling for All-to-All Personalized Communication on Ethernet Switched Clusters. In *International Parallel and Distributed Processing Symposium*.

[12] Dror G. Feitelson. 2018. Logs of Real Parallel Workloads from Production Systems. https://www.cse.huji.ac.il/labs/parallel/workload/logs.html.

[13] Juan J. Galvez, Nikhil Jain, and Laxmikant V. Kale. 2017. Automatic Topology Mapping of Diverse Large-Scale Parallel Applications. In *International Conference on Supercomputing*.

[14] Taylor Groves, Yizi Gu, and Nicholas J. Wright. 2017. Understanding Performance Variability on the Aries Dragonfly Network. In *International Conference on Cluster Computing*.

[15] Philip Hall. 1935. On Representatives of Subsets. *Journal of the London Mathematical Society* s1-10, 1 (1935), 26–30.

[16] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2008. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In *International Conference on Cluster Computing*.

[17] Torsten Hoefler and Marc Snir. 2011. Generic topology mapping strategies for large-scale parallel architectures. In *International Conference on Supercomputing*.

[18] Nikhil Jain et al. 2017. Predicting the Performance Impact of Different Fat-Tree Configurations. In *Supercomputing*.

[19] Nikhil Jain, Abhinav Bhatele, Xiang Ni, Todd Gamblin, and Laxmikant V. Kale. 2017. Partitioning Low-diameter Networks to Eliminate Inter-job Interference. In *International Parallel and Distributed Processing Symposium*.

[20] Andrzej Jajszczyk. 2003. Nonblocking, repackable, and rearrangeable Clos networks: fifty years of the theory evolution. *IEEE Communications Magazine* 41, 10 (2003), 28–33.

[21] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. 2008. Technology-Driven, Highly-Scalable Dragonfly Topology. *SIGARCH Comput. Archit. News* 36 (June 2008), 77–88. Issue 3.

[22] Jason Lee, Zhou Tong, Karthik Achalkar, Xin Yuan, and Michael Lang. 2016. Enhancing InfiniBand with OpenFlow-Style SDN Capability. In *Supercomputing*.

[23] C.E. Leiserson. 1985. Fat-trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers* 34, 10 (October 1985).

[24] Hans Meuer, Erich Strohmaier, Jack Dongarra, and Horst Simon. 2009. “Top500 Supercomputer Sites”. http://www.top500.org.

[25] P. Patarasuk and X. Yuan. 2007. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *Workshop on High-level Parallel Programming Models and Supportive Environments*.

[26] Samuel A. Pollard, Nikhil Jain, Stephen Herbein, and Abhinav Bhatele. 2018. Evaluation of an Interference-Free Node Allocation Policy on Fat-Tree Clusters. In *Supercomputing*.

[27] Daniele De Sensi, Salvatore Di Girolamo, and Torsten Hoefler. 2019. Mitigating Network Noise on Dragonfly Networks through Application-Aware Routing. In *Supercomputing*.

[28] Arjun Singh. 2005. *Load-Balanced Routing in Interconnection Networks*. Ph.D. Dissertation. Dept. of Electrical Engineering, Stanford University.

[29] Joseph Skovira, Waiman Chan, Honbo Zhou, and David A. Lifka. 1996. The EASY-LoadLeveler API Project. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*.

[30] Staci A. Smith, Clara Cromey, David K. Lowenthal, Jens Domke, Nikhil Jain, Jayaraman J. Thiagarajan, and Abhinav Bhatele. 2018. Mitigating Inter-Job Interference Using Adaptive Flow-Aware Routing. In *Supercomputing*.

[31] Staci A. Smith and David K. Lowenthal. 2021. Jigsaw: A High-Utilization, Interference-Free Job Scheduler for Fat-Tree Clusters (Extended Version). http://dkl.cs.arizona.edu/publications/papers/hpdc21-extended.pdf.

[32] Staci A. Smith and David K. Lowenthal. 2021. Jigsaw HPDC 2021 Submisson Repository. https://osf.io/vxk7m/?view_only=None.

[33] Flux Framework Team. 2020. 2014 Cab Supercomputer Job Scheduling Traces. https://doi.org/10.5281/zenodo.3908771

[34] Sudharshan S. Vazhkudai et al. 2018. The Design, Deployment, and Evaluation of the CORAL Pre-Exascale Systems. In *Supercomputing*.

[35] Xu Yang, John Jenkins, Misbah Mubarak, Robert B. Ross, and Zhiling Lan. 2016. Watch Out for the Bully! Job Interference Study on Dragonfly Network. In *Supercomputing*.

[36] Eitan Zahavi. 2010. *D-Mod-K routing providing non-blocking traffic for shift permutations on real life fat trees*. CCIT Report #776. Israel Institute of Technology.

[37] Eitan Zahavi, Alexander Shpiner, Ori Rottenstreich, Avinoam Kolodny, and Isaac Keslassy. 2016. Links as a service (LaaS): Guaranteed tenant isolation in the shared cloud. In *Symposium on Architectures for Networking and Communications Systems*.