CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

The Case For Colocation of HPC Workloads

Alex D. Breslow[1*], Leo Porter[2], Ananta Tiwari[3], Michael Laurenzano[3], Laura Carrington[3], Dean M. Tullsen[1], Allan E. Snavely[3,4]

[1]*University of California, San Diego*
[2]*Skidmore College*
[3]*San Diego Supercomputer Center*
[4]*Lawrence Livermore National Laboratory*

SUMMARY

The current state of practice in supercomputer resource allocation places jobs from different users on disjoint nodes both in terms of time and space. While this approach largely guarantees that jobs from different users do not degrade one another's performance, it does so at high cost to system throughput and energy efficiency. This focused study presents job striping, a technique that significantly increases performance over the current allocation mechanism by colocating pairs of jobs from different users on a shared set of nodes. To evaluate the potential of job striping in large scale environments, the experiments are run at the scale of 128 nodes on the state-of-the-art Gordon supercomputer. Across all pairings of 1024 process NAS parallel benchmarks, job striping increases mean throughput by 26% and mean energy efficiency by 22%. On pairings of the real applications GTC, LAMMPS, and MILC at equal scale, job striping improves average throughput by 12% and mean energy efficiency by 11%. In addition, the study provides a simple set of heuristics for avoiding low performing application pairs.

Received ...

KEY WORDS: Supercomputer Performance, HPC Application Symbiosis, Energy-Aware Optimization

1. INTRODUCTION

Recent studies into the feasibility of exascale computing have shown that exascale will present a series of unique challenges [1–4]. To conquer these challenges, a reexamination of every level of the supercomputing infrastructure will be necessary, from the facility, to hardware, and up through all levels of the software stack. In this study, we show that the de facto supercomputing scheduling mechanism can be inefficient, and that there is a significant performance and energy efficiency opportunity with an alternative approach.

Most data center supercomputers are a collection of individual nodes or server blades that are connected to one another over a high speed network. Nodes are parallel computers in their own right, often containing multiple sockets and dozens of cores. Such supercomputers are inherently hierarchical and exhibit multiple levels of shared resources including on-chip buses and cache structures, on-node shared memory bandwidth and network interfaces, and the system-wide network interconnect. While shared structures help facilitate the function of a massively parallel machine, their presence necessitates conscientious orchestration of the sharing of these resources.

The prevailing approach to this coordination of resource sharing is simple: schedule jobs from different users on disjoint sets of compute nodes. While this limits the potential chaos of an

[*]Correspondence to: Computer Science and Engineering Department, University of California, San Diego, 9500 Gilman Dr., La Jolla, CA 92093, USA.

ungoverned scheduling environment where any user can monopolize the system, it also eliminates the advantages that cooperative sharing can offer. In particular, prior work on multiprogrammed workloads shows that system performance is significantly increased when heterogeneous threads are intelligently coscheduled on the same computational resources [5–10]. Across a suite of different architectures, these studies find benefit in enforcing policies that facilitate fairer sharing of resources, which avoids thread starvation and accordingly increases throughput.

The reason colocating heterogeneous threads usually improves system performance is because each thread executes distinct code and thus taxes the system in largely different ways. In contrast, homogeneous threads execute near identical code and consequently often make very similar demands on the system. As such, they are more likely to impede one another's progress by competing over shared resources such as last level caches (LLC) and memory bandwidth. Avoiding these collisions on shared resources is key to realizing high aggregate system performance.

For the purposes of this study, we examine the performance opportunity provided by harnessing HPC workload heterogeneity at scale. Most large distributed scientific applications fall under the Single Program Multiple Data (SPMD) programming model. SPMD parallel programs are comprised of multiple identical or nearly identical tasks that operate on distinct data. Since they are similar, the colocation of these tasks on the same computing resources produces contention and hurts performance. Unfortunately, this is exactly what happens on present day supercomputing systems. Each application is given a private set of compute nodes. While this approach makes it easier to guarantee quality of service, it causes homogeneous tasks to be placed together, consequently increasing contention and accordingly decreasing throughput and energy efficiency.

For these reasons, we propose job striping, a process-to-core mapping technique for supercomputing clusters. Job striping reduces contention by capitalizing on the heterogeneity in workloads found on today's supercomputers. On a job-striped set of compute nodes, half the cores of each processor run one parallel application and the other half run another. By interleaving two distinct jobs with one another, system contention is reduced.

To validate the efficacy of job striping, we conduct large-scale experiments on two complete racks (2048 cores) of the state-of-the-art Gordon supercomputer at SDSC. For the 1024 process NAS parallel benchmarks [11], job striping improves single application throughput by as much as 81%, and more than 26% on average. In addition, energy efficiency increases by as much as 52%, and 22% on average.

We also evaluate job striping on three very common scientific applications, GTC [12], LAMMPS [13], and MILC [14] at 1024 processes. On these applications, job striping increases mean throughput by 12% and energy efficiency by 11%. MILC improves the most from striping, with throughput improving by 23% and 32% when it is paired with GTC and LAMMPS, respectively.

This paper also identifies the critical resources that benefit most from heterogeneity – the memory subsystem and the communication network. All applications whose performance improves from striping exhibit reduced contention for one or both of these resources when the workload is heterogeneous.

The primary contribution of this paper is to demonstrate that colocating large scale HPC applications yields a performance benefit. Prior work on heterogeneous scheduling focuses almost exclusively on uni- and multi-threaded benchmarks or synthetic workloads run on a single server. In addition, it largely glosses over energy efficiency and network contention while using benchmark suites that are not representative of HPC workloads. This study, in contrast, demonstrates both the necessity for and the mechanisms to enable heterogeneous workload scheduling for real-world HPC environments.

Additional contributions of this paper are as follows:
(1) We motivate job striping for HPC and describe a particular software solution.
(2) We present a new energy efficiency metric *Scaled Energy Efficiency* (SEE), which approximates the ratio of before and after power-delay products.
(3) We demonstrate that job striping significantly increases energy efficiency and throughput on real applications.

(4) We show that job striping achieves its performance gains by reducing contention on shared resources, through analysis of application profiles.
(5) We provide a simple mechanism to predict low performing application pairs.
(6) We show that pairing GTC with MILC, two applications commonly run at NERSC [15], improves energy efficiency by 13%.

2. MOTIVATION

In this paper, we motivate job striping by exploring an application's behavior when mapped in two commonly found processor affinities, one which we refer to as *compact* and the other which we refer to as *spread*. These terms are consistent with what is found in the literature on NUMA affinity optimizations [16] and in software and system user guides [17, 18]. The best way to explain these configurations is by an illustrative example.

Suppose we are given a distributed application A that is comprised of p single-threaded processes and a set of chip multiprocessors (CMPs) with c cores each. The *compact* configuration would assign CMPs to A such that each process is mapped to a single core with no cores left unassigned. In contrast, the *spread* configuration instead opts to use twice as many CMPs but to leave each socket half full. Thus *spread* uses $\frac{c}{2}$ cores per CMP versus the c that the compact configuration uses.

The spread affinity specification often offers a significant performance advantage over compact because each process experiences less cache contention from its neighbors. In the compact configuration, the shared last level cache (LLC) and memory bandwidth are split among c processes. However, in the spread configuration they are divided between $\frac{c}{2}$ processes. This means that the mean available memory bandwidth and LLC space per core are 2x higher in the spread configuration. Since both of these resources are critical for achieving high performance, the spread configuration almost always outperforms compact. While this is likely a win for raw performance, job spreading has its limitations.

In real supercomputing systems, users are charged for the combined number of CPU hours that their application uses. The price of execution is derived by a simple formula where cost is the product of the number of nodes assigned to a job (N), the number of cores per node (P), the total time the job ran (T), and a rate constant (k) (shown below).

$$C = k * P * N * T \tag{1}$$

Consequently, if a user spreads their job, they are charged for the cores that they leave unassigned. While this would not be a problem if spread offered a 2x increase in performance, in many cases it does not. As such, users have a monetary incentive to maximize their application's performance per CPU hour, not per core. Thus users commonly request only enough nodes such that every process has a core to run on (compact). While this makes sense from the user's economic perspective, this may not yield the best overall system performance. For many scientific computing applications, the memory bandwidth requirements per core are quite high, and when the processor is fully occupied with multiple identical threads or processes, memory bandwidth is insufficient for delivering maximal throughput per process.

In fact, recent studies that examine single-process multi-threaded and distributed programs demonstrate that the optimal number of threads per CMP is highly application specific and rarely equal to the number of cores on the processor. In such cases, the optimal number of threads is either moderately lower or much higher than the number of cores available on a CMP [19, 20].

In such an environment, what we desire is a way to schedule that takes the best features of both *compact* and *spread*. We want the full occupancy that *compact* provides, but we also need the reduction in resource contention that *spread* offers. To do this, we develop and investigate *job striping*. Job striping takes two spread jobs and interleaves them such that every core is occupied by a uniquely identified single-threaded process. In the *striped* configuration, even cores get assigned one application and the odd cores another. This process is repeated across all processor sockets. Figure 1 illustrates the *compact*, *spread*, and *striped* configurations.

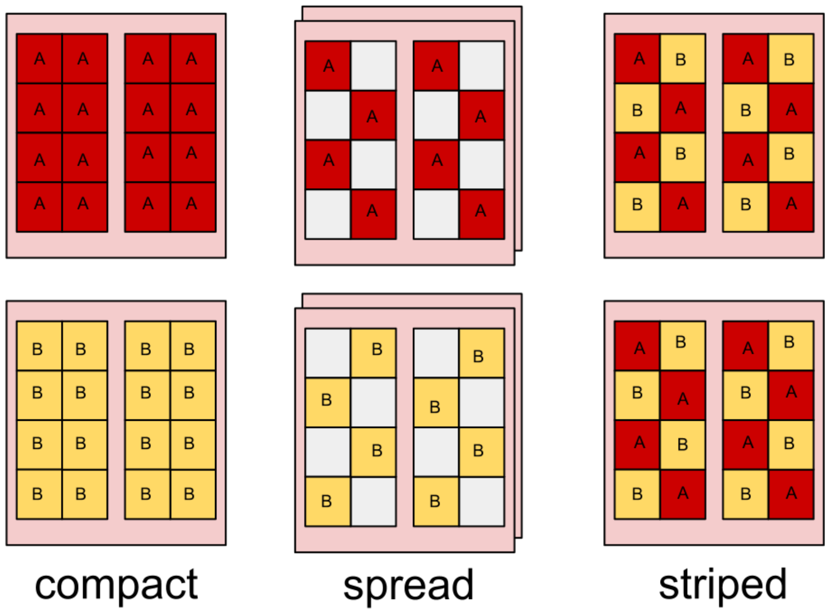

Figure 1: Methods for scheduling two 16 process distributed applications A and B on a supercomputer with two 8-core processor sockets (as in Gordon). In *compact*, A runs on one node and B on another. With *spread*, A and B are still isolated but run on two compute nodes each, with half the cores empty per CMP. Lastly in *striped*, A and B share all sockets of two compute nodes.

Our intuition for why this can be effective is simple: distributed applications using MPI, or another mechanism such as UPC or SHMEM, often execute near identical tasks. As such, their constituent processes make very similar demands on the system. Suppose we are given a parallel program named A that largely follows the SPMD model. If each process comprising A requires that 6MB of its working set remain in L3 cache for high performance, then if one places 8 of these processes on the same 8-core processor, then at least 48MB of shared L3 cache will be required to avoid a reduction in throughput. However, on today's HPC systems, most L3 caches top out at 32MB. The requirement for a larger combined working set than the L3 offers will undoubtedly cause an increase in cache misses and, subsequently, a drop in performance.

However, job striping could remedy this problem. If A is striped with another job B whose processes each require 1MB in L3 cache, then the net performance is likely to be much better. While A scheduled as *compact* will almost certainly bottleneck on L3 capacity, when paired as *striped* with B, the sum of their working sets in L3 will be 28MB, small enough to fit within the 32MB L3 cache.

Communication over the inter-node network can also benefit from job striping. Many distributed scientific applications spend up to 35% of their total execution time sending and receiving messages [21]. Depending on the nature of this communication, similar processes can heavily contend for shared hardware such as the network interface card and switches. Suppose A exhibits short but intense bursts of communication. During these periods, it is quite likely that there will be a backlog of communication requests. If computation cannot be done to hide the latency of this communication, then performance will suffer. However, if there are half as many copies of A per node and B does comparatively little communication, then A is likely to benefit from less waiting on the network and B will continue to make progress. In addition, even if A and B both are communication-heavy, their communication patterns are likely to be more out of phase than those of single parallel application [22].

In particular, a parallel program's tasks are limited in the distance they can run ahead of one another by the time length between global synchronization points. For highly synchronous codes, this means that all threads or processes execute the same set of instructions in relative lock step. For a communicating application A, this means that all of its processes very well might attempt to send messages over the network at roughly the same time. However, if A is paired with another code B, then the temporal communication patterns of A and B's processes can fall largely out of sync. This fact is corroborated by a closely related work by Koop et al. [22] where they show that pairs of symbiotic NAS parallel benchmarks often communicate over largely non-overlapping time intervals. For programs that make heavy use of synchronous communication such as MPI_Send and MPI_Receive, striping two applications in this way can significantly speed up program execution.

3. METHODOLOGY

3.1. Performance Metrics

To evaluate effective coschedules, we utilize a number of key metrics. To quantify system throughput, we use a variant of the weighted speedup metric from Snavely and Tullsen's work on symbiotic coscheduling for an SMT processor [5]. We call this metric scaled throughput (STP). STP is shown in Equation 2.

$$STP = \frac{1}{n}\sum_{i=1}^{n}\frac{S_i}{M_i} \tag{2}$$

In this equation, n is the number of parallel programs that are coscheduled together, S_i is the runtime of a program i when run alone in the compact configuration and M_i is the runtime of the program i when either the spread or striped affinity is applied to it. This variant of STP is the mean weighted speedup of the applications in a coschedule. For instance, a coschedule with an STP of 1.2 would exhibit a 20% average speedup.

For the purposes of determining energy efficiency, we have developed a new metric—Scaled Energy Efficiency (SEE). SEE is shown in Equation 3.

$$SEE = \frac{STP}{\overline{P_s}}\sum_{i=1}^{n}\overline{P_{c,i}} \tag{3}$$

$\bar{P_s}$ is the mean power of the striped or spread job schedule. $\bar{P_{c,i}}$ is the mean power of job i when run in the compact configuration. SEE is the product of scaled throughput and scaled power consumption. SEE is a reasonable energy efficiency metric because energy can be defined as the product of average power and time. Consequently, the metric approximates the ratio of the before (compact) and after (striped) power-delay products.

The $\bar{P_{c,i}}$'s terms are additive because the energy used by two jobs running isolated in the compact configuration is the sum of their individual energies. We can divide this by $\bar{P_s}$ because the number of compute nodes occupied by two jobs A and B scheduled compactly is the same as when the two are striped together. The $\bar{P_s}$ term is necessary because it is currently not possible to measure the contribution to net power from each job when they are striped. Only a combined measurement is realistically feasible.

3.2. Gordon Supercomputer

All of our experiments are conducted on San Diego Supercomputer Center's (SDSC) new Gordon Supercomputer [23]. Gordon is a particularly innovative machine. It is the first supercomputer to incorporate Intel's Sandy Bridge processors and has state-of-the-art Intel NAND Flash solid state drives on every node. The system features a 3D torus network topology with dual-rail QDR infiniband with 8GB/s of bandwidth in each direction between nodes. Each compute node is dual-socketed, each socket being occupied by an 8-core 2.6 GHz Xeon processor with 32KB of L1 and 256KB of L2 private cache per core and 20MB of shared L3 per socket for a total 40MB. Every compute node has 64GB of main memory with 32GB per NUMA node. We obtain this information by using the `lstopo` command from HWLOC [24] and from the Gordon user guide [18].

3.3. NAS Parallel Benchmarks

For one part of our study, we use the message passing interface (MPI) version of the NAS Parallel Benchmarks (NPBs) [25, 26]. As of version 3.3, the NPBs consist of 9 parallel benchmarks with input sizes ranging from W,S,A,B,C,D,E, and F where jumps between classes roughly scale runtime by 4 to 16x. NAS binaries are named as <benchmark name>.<class size>.<number of compiled processes>. For our experiments, we chose to use bt.D.1024, cg.E.1024, ep.E.1024, ft.D.1024, lu.E.1024, mg.E.1024, sp.D.1024. These input sizes ensure that each application runs for at least 20 seconds when scheduled in the compact mode.

The NAS Parallel Benchmarks are important for our study because they represent common computational kernels or motifs that comprise more complex real-world applications. In addition, they are open source, well studied, and exhibit strict versioning. Thus our experiments are repeatable.

3.4. Real Scientific Applications

We also conducted a study using three ubiquitous scientific applications: the 3D Gyrokinetic Toroidal Code (GTC) [12], LAMMPS [13], and the MIMD Lattice Computation (MILC) [14]. GTC is used for modeling microturbulence in plasma and is very heavily used at NERSC at the Lawrence Berkeley National Laboratory (LBL) as well as other Department of Energy (DOE) sites. LAMMPS is a classical molecular dynamics code that has been identified as one of the Department of Defense's key applications. In addition, it is currently being used by Lawrence Livermore National Laboratory (LLNL) as one of the applications to benchmark the new IBM BlueGene/Q Sequoia supercomputer [27]. Our third application MILC is a quantum chromodynamics program that enables the study of subatomic particle interactions. GTC and MILC were selected along with four other scientific applications to benchmark the petascale Hopper Supercomputer at LBL [15].

For GTC, we use an input file that is a modification of one provided by one of its lead developers. With LAMMPS, we use the embedded atom model (EAM) potential input provided with the Sequoia Benchmarks and scale the lattice by a factor of 32 in each of the x, y, and z directions. With MILC, we use the input provided in the NERSC-6 benchmarks for 1024 processes.

3.5. Compilation

We compiled both the NAS parallel benchmarks and the scientific codes using the Intel compilers (version 12.1). For our MPI library, we use MVAPICH2 (version 1.8) because it is optimized for multirail QDR infiniband. For GTC, we largely left the makefile untouched. LAMMPS requires an FFT library – we use the FFTW3 module (version 3.5) that defaults on Gordon when the Intel compilers and MVAPICH2 modules are loaded. In addition, we compile MILC with NETCDF (version 4.0.1).

3.6. Experimental Setup

For each application, we spawn 1024 processes using mpirun_rsh. We run all of the experiments on two full racks of Gordon, totaling 128 compute nodes and 2048 processor cores. We evaluate NPBs and real applications separately. For each job, we run it in the compact, spread, and striped configurations. During trials using the compact scheduling method, we execute one copy of the application on one rack and another copy on the other. In the spread configuration, we bind four processes to each processor socket by setting MV2_CPU_BIND=0:2:4:6:8:10:12:14 (see [17]). We do this to enforce job spacing in the hope that the absence of processes on adjacent cores will improve energy efficiency [28]. In both the compact and spread schemes, we restart a job as soon as it finishes.

For striped jobs, we set the affinity mask of one job to MV2_CPU_BIND=0:2:4:6:8:10:12:14 and the other to MV2_CPU_BIND=1:3:5:7:9:11:13:15. We start both jobs at the same time, and as soon as one job finishes, we catch the signal and restart it. Each striped coschedule is run for a fixed time interval, and the final trial from each job is discarded. For a visual representation, see Figure 2. This

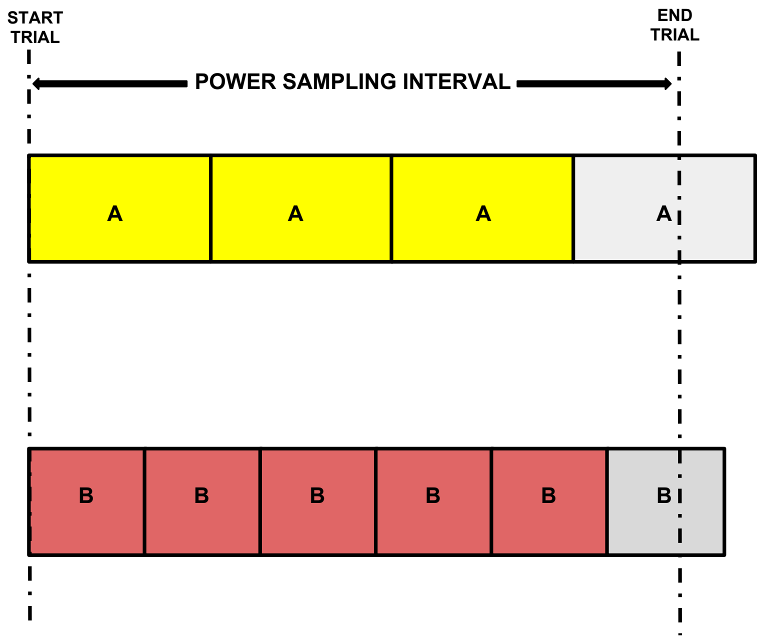

Figure 2: This figure represents a trial of running two striped applications A and B.

methodology allows us to (1) ignore unwanted tail effects and (2) sample the jobs co-executing at a variety of time staggers.

For the NPBs, we run each job in the compact and striped configurations for 12 minutes. After that, we run each possible pairing of NPBs for 12 minutes as well. This is adequate time so that each job completes execution at least three times. Between jobs, we *sleep* for 90 seconds to allow the system to go back down to steady state. For our real applications, we repeat the same steps but increase runtime per experiment to 30 minutes. With this time allowance, each application runs for at least 4 iterations per experimental trial. After running all trials, we repeated the same experiments several days later.

3.7. Power Measurement

We measure power consumption using a key feature of the Gordon supercomputer. For each rack, there are two power sources that provide the energy necessary for operation. On each of these sources is an embedded power monitoring unit (PMU). Each PMU makes a power measurement approximately every 3 to 5 seconds, however not in sync. The Performance Monitoring and Characterization (PMaC) laboratory at SDSC has developed a system whereby these data are sent over the network to a remote host and the readings are compiled into logs. To find average power, we take the discretized integral over the power readings of each PMU device during the runtime interval. As previously mentioned, this interval is 12 minutes for the NPBs and 30 minutes for the real applications. Once we have these values, we sum them together and divide by the runtime (12 or 30 minutes) to get average power. We do this rather than just taking the average over each power measurement because the time intervals between such measurements are not uniform.

3.8. Performance Correlation

The first time we run the experiments, we dynamically instrument each executable at runtime with the IPM [29] profiler. When we rerun the experiments, we instead instrument with the profiling capabilities of PSiNS [30]. During each run of every experiment, we collect the L1, L2, and L3 cache misses, the total dynamic instructions, and the number of cycles without instruction issue

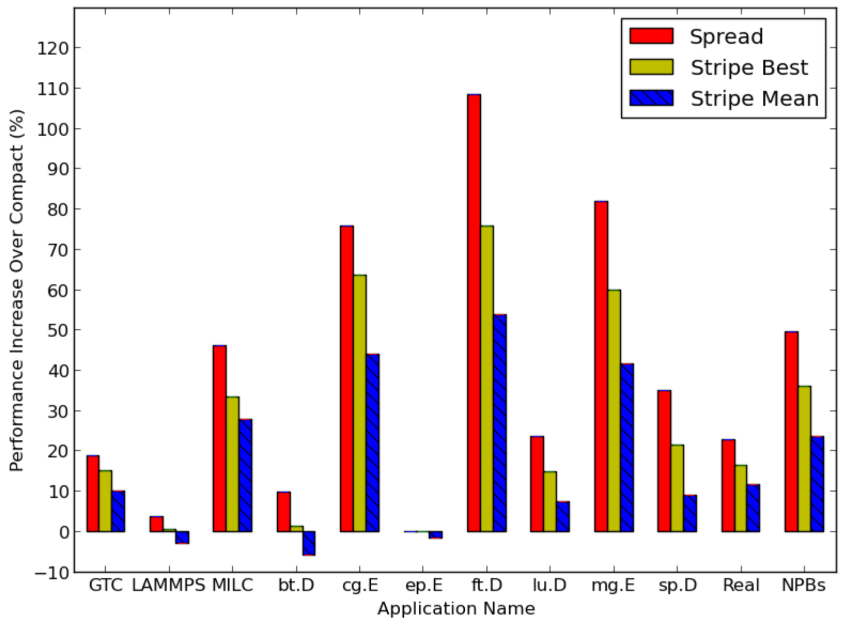

Figure 3: Increase in system throughput (STP) over compact when applying job spreading and striping to the NAS parallel benchmarks and GTC, LAMMPS and MILC

for each MPI task. These data are aggregated at the process level through the use of PAPI [31] performance counters. In addition, both tools report the percent of time spent in each MPI routine for every process. The total penalty of instrumenting each application is at most 10%. This instrumentation is present in every run of the compact, spread and striped experiments. We do not believe that this instrumentation fundamentally alters application behavior. To make sure this is the case, we have repeated a large subset of our experiments without instrumentation and have observed no significant change in coschedule outcomes.

This instrumentation allows us to categorize the behavior of an application through the analysis of the traits of each of the parallel tasks that comprise it. From these data, we draw correlations between increase in striped performance and a reduction in system contention.

4. PERFORMANCE RESULTS

4.1. Compact Versus Spreading Versus Striping

Figure 3 shows the performance results for the first set of experiments with the NAS parallel benchmarks and the second set of experiments with GTC, LAMMPS and MILC. For the NAS parallel benchmarks, the mean performance increase from job spreading is 50%. If one examines striped coschedules of non-identical NPBs, the average performance increase is 26%. If one selects the best running mate other than embarrassingly parallel (EP) for each benchmark, then the average increase in performance is 36%. We choose to exclude EP because EP is minimally contentious. Each EP task's working set fits entirely in the private levels of cache, and EP spends very little time in active communication. Because of these traits, EP universally causes every application that it stripes with to achieve its best striped performance. Thus for the sake of fairness and realism, we exclude these results from the "Best" average.

For the NAS parallel benchmarks, random striping yields about 50% of the performance benefit of job spreading and striping each job with its best running mate provides 70% of the performance benefit of spreading. This trend continues for real applications as well. For the GTC, LAMMPS and

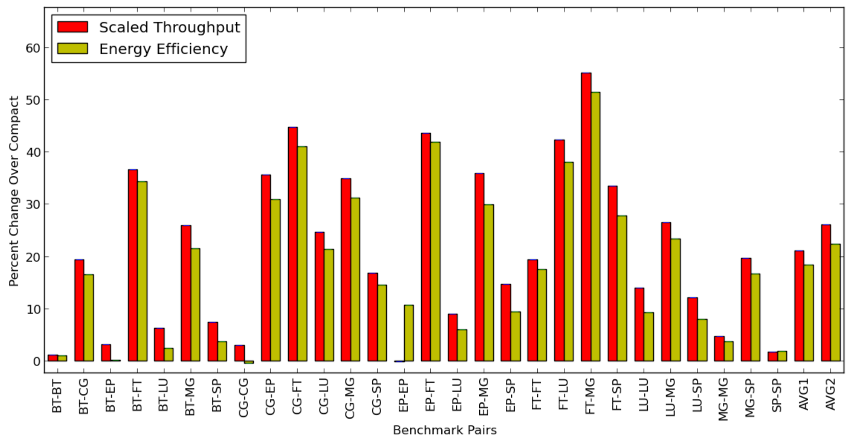

Figure 4: Increase in system throughput and energy efficiency when applying job striping to pairs of 1024 process NAS parallel benchmarks

MILC, job spreading increases throughput by 23%, and the mean heterogeneous striping and the mean best striping improve performance by 12% and 16% respectively.

4.2. NAS Parallel Benchmarks

In this section, we examine the increase in collective throughput and energy efficiency for pairs of striped NAS Parallel Benchmarks. The results are presented in Figure 4.

For completeness, we run all pairwise combinations. This includes both heterogeneous pairings (e.g., CG striped with LU) and homogenous pairings (e.g., CG striped with CG). However, homogeneous pairings typically combine the worst of *compact* (maximized pressure on bottleneck resource) and the worst of *spread* (some increased communication as communicating threads are farther apart). Thus, AVG1 refers to the average of all possible pairings and AVG2 to the mean performance of only heterogeneous pairs. It is reasonable to assume that a job striping runtime system would avoid pairing identical applications together.

Overall, striping provides improved performance. Even in the worst case where we are limited to pairs of identical benchmarks, job striping in aggregate still increases both STP and SEE by 6%. Although the kernels are identical, the two instances do not synchronize with one another, and as a result they compete less for the inter-node network. FT-FT is the best homogeneous pairing and exhibits a 19% increase in performance and a 17% increase in energy efficiency. For additional discussion of the FT-FT pairing, see [22].

Heterogeneous striping improves system throughput by 26% and energy efficiency by 22%. With every benchmark, striping improves scaled throughput (except EP striped with EP -0.2%) and energy efficiency (except for CG striped with CG -0.5%). The best coschedule is the FT-MG pairing. FT speeds up by 51% and MG by 60% for a total STP increase of 55%. Energy for the FT-MG pairing improves by 52%. The next best coschedule in terms of throughput and energy efficiency is CG paired with FT. CG speeds up by 51% and FT by 39% for a total combined performance improvement of 45%. The next two best pairings are EP-FT and FT-LU.

The EP-FT pairing is particularly interesting because it is one of the instances where one application benefits at the expense of another. If we saw no benefit from heterogeneous scheduling, we would expect this to be the common case – that job striping would sacrifice one job for the benefit of another (e.g., a cache-intensive job would accelerate, but the co-scheduled job would suffer from the increased cache pressure). In fact, in over half the cases, job striping improves the throughput of *both* applications. However, EP-FT is one of the exceptions. EP suffers a 4.6% loss in performance, whereas FT's throughput increases by 92%. There are 13 such cases. In these pairings, the average

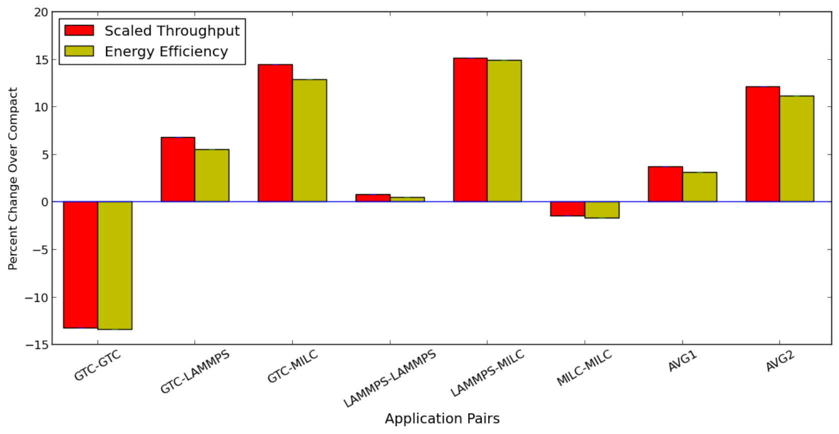

Figure 5: Increase in system throughput and energy efficiency when applying job striping to pairs of 1024 process scientific codes; AVG1 is the mean over all coschedules, and AVG2 is the mean over heterogeneous ones.

performance decrease to the victim application is 4.1%, whereas the average performance increase of the accelerated application is 48.9%. BT and EP are the most commonly victimized kernels, with BT suffering a performance hit from five out of seven pairings, and EP suffering a throughput hit in all seven. Besides these, only CG-SP and CG-LU exhibit victimization (SP and LU victims).

We can partition the pairings into the set of jobs where both improve, and those where one is slowed. Interestingly, the overall system performance gain of the two sets is similar. The former sees 19.4% gain from striping, while the latter gains 19.1%. The gap is only a bit larger if we examine energy efficiency. The former group gains an SEE improvement of 19.3%, while the latter gains 16.4%. These results do raise questions about fairness for the users who initiated the striped jobs. These will be addressed further in Section 6.

4.3. *Real Scientific Applications*

For the production scientific codes, job striping again improves throughput and energy efficiency. Figure 5 illustrates the scaled throughput and energy efficiency per striping. As with the NAS parallel benchmarks, we observe that some applications benefit more than others from striping. The application that benefits most from striping is MILC (at 1024 processes on the NERSC input). When it is paired with GTC and LAMMPS, its throughput increases by 23% and 34% respectively. GTC also exhibits performance gains when striped. Striping GTC with LAMMPS and MILC improves GTC's throughput by 12% and 6%.

LAMMPS is the only application that does not benefit from heterogeneous striping and the only one whose performance improves from homogeneous striping. It suffers a mean 1.7% decrease in throughput when striped with GTC and 3.4% performance loss when paired with MILC.

In addition to improving application throughput, job striping also raises energy efficiency. Similar to the NAS parallel benchmarks, there is little change in power consumption relative to the increase in useful execution. For heterogeneous coschedules, mean energy efficiency increases by 11%, almost as much as the 12% increase in scaled throughput.

4.3.1. Stability of Results Table I reports the minimum and maximum observed runtime per real application over eight or more trials. From the table, we observe that the results from the previous section are quite stable and reliable. Over the multiple runs of each application, the maximum variance in runtime for all applications in each of the three heterogeneous pairings is less than 2% and is no larger than running applications compactly. Consequently, the behavior of a striped coschedule is highly consistent between runs.

Table I: Minimum and maximum runtime in seconds for each of the three applications scheduled compactly, spread, or striped.

		Compact	Spread	GTC	LAMMPS	MILC
GTC	MIN	198.9	167.6	227.9	172.7	187.9
	MAX	202.2	170.2	231.7	174.3	191.5
LAMMPS	MIN	289.9	280.0	295.2	288.3	300.8
	MAX	292.0	282.9	296.3	290.2	302.9
MILC	MIN	511.4	350.2	416.7	383.5	515.7
	MAX	514.7	354.7	424.6	388.3	527.4

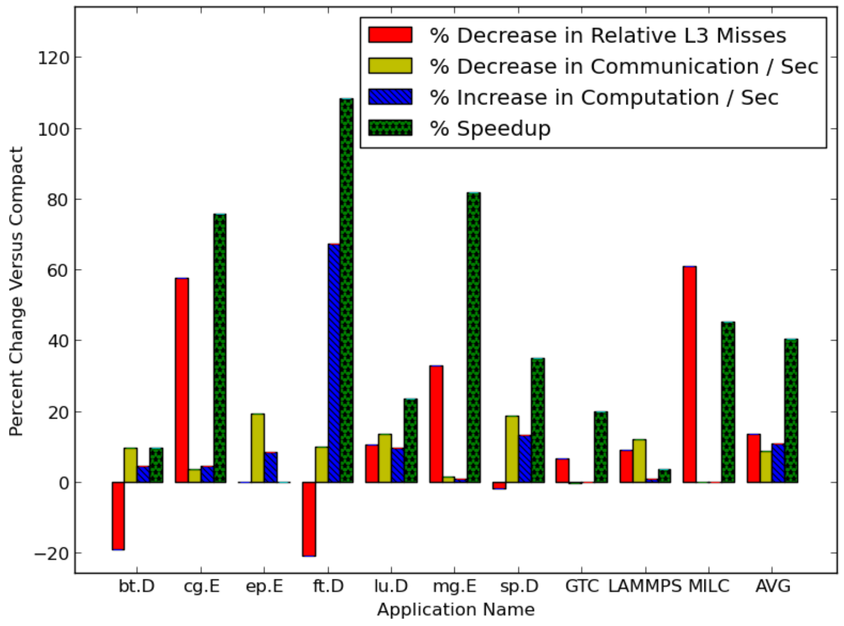

Figure 6: Shown are the change in application performance and performance counters when jobs are spread versus compact: L3 change is scaled relative to the mean L3 misses per 1000 instructions of all ten codes when individually scheduled (compact) and communication and computation changes are scaled by percentage of total runtime. The increase in computation is relative to the total percentage of time spent in computation versus communication.

5. INTERACTION BETWEEN STRIPED PAIRS

Thus far we have only focused on the performance benefit of job striping. In this section, we investigate how it is achieved. From our point of view, the best way to predict how a job's performance will change when striped is to look at the change in its relative L3 misses and communication when spread. Since striping is the interleaving of two spread jobs, a job's change in behavior when spread should largely dictate how and when it will benefit from striping. Figure 6 presents the change in relative L3 misses, the time spent in computation and communication, and the increase in performance.

5.1. Spread Performance Relative to Compact

We examine L3 cache misses because previous studies have demonstrated that applications with high L3 cache miss ratios often have lower instruction throughput than ones with fewer L3 misses.

In addition, the literature shows that pairing heterogeneous uniprocess single and multithreaded programs together can often reduce L3 misses, and boost performance [7]. On real machines, an L3 cache miss incurs a latency penalty that is often an order of magnitude greater than misses to the higher level private caches. On Intel Sandy Bridge processors, the quoted L3 access time is 26 cycles. However, the latency to access main memory is much greater. We use lmbench to measure the relative latencies and find the L3 access time to be 42 cycles and the time to access main memory to be 220 cycles [32,33]. Thus, for each miss to the L3 cache, we incur an additional latency penalty of 180 cycles for that cache line. If this miss occurs on a read, it is unlikely that the processor can completely hide this latency. These latencies are of particular concern for CG and MG which exhibit 5.7 and 2.7 L3 misses per 1000 instructions.

In Figure 6, we observe that certain benchmarks show a significant reduction in L3 cache misses when spread. CG, MG and MILC realize substantial benefit when spread. This is not surprising: CG, MG and MILC have the largest working sets, each allocating 873, 499, 716 MB per active process, and also the highest L3 miss ratios of their peers. Thus the spread configuration greatly reduces cache and memory pressure.

However, certain programs such as BT, FT, and SP actually incur more misses as a result. While an increase in performance despite more L3 misses may seem counter intuitive, if we examine the increase in useful computation and the decrease in communication, these numbers make sense. All three benchmarks exhibit substantial increases in computation per unit time. Thus, even though they incur penalty from more cache misses, this can be offset by an increase in useful work and less waiting on the network.

Most applications exhibit both a decrease in L3 misses and decrease in communication per unit time. To weigh the importance of these changes, one must first look at the applications' characteristics. EP for instance has almost no L3 cache misses per 1000 instructions, thus pairing it with other applications can only hurt. In addition, EP performs much less communication than the other applications. Thus when it is heterogeneously paired, its performance universally suffers. BT is also interesting because spreading significantly increases its relative L3 misses. Thus BT can only benefit from striping if it reduces BT's communication contention. If we look at BT's best pairings, they all occur when it is paired with applications that do as much or less communication than it does (itself and EP).

If we examine the highest throughput coschedules, almost all include FT. FT's L3 misses go up when it is paired with other applications, but this is expected given that spreading alone increases its relative L3 misses. This fact is however offset by the savings due to communication. In particular, FT.D.1024 spends more time in MPI routines than any other program, and it has the second lowest relative L3 cache misses of any program (0.48 per thousand instructions). In addition, reducing the time it spends in communication sharply increases its actual computation per unit time. Thus, it can afford additional cache misses.

When FT is paired with CG and MG its communication per unit time decreases by 8.0% and 4.5% respectively. While this may not seem like much, remember that FT originally spends over 87% of its time communicating. Thus decreasing communication per cycle by 8.0% means FT spends 20% of its time executing meaningful computation instead of 13%. Thus, it executes 54% more computation per unit time. This figure almost exactly matches the 55% increase in throughput for FT, when it is paired with MG. The same is also true for FT paired with CG. FT now spends 30% more time doing computation. This number roughly matches the 39% increase in performance that we observe when it is paired with CG. In addition, we note that CG and MG also spend a large percentage of their total time in communication, 57% and 37% respectively. However, their communication patterns are distinct from FT's. FT spends approximately 80% of its total communication time executing MPI_Alltoall, and thus each process, and hence every compute node sends every other compute node its results.

CG on the other hand spends 53% of its total computation time in synchronous point to point MPI_Send communication and 43% waiting. We believe FT benefits from this pairing because CG's communication pattern is distinct and CG is largely inactive in MPI_Wait for large periods of time. Thus FT benefits from CG waiting on communication and also by the fact that the nature and phases of CG's communication are largely orthogonal to its own.

MG's behavior is also highly distinct from FT's. MPI_Wait and MPI_Init together account for 73% of its total time in MPI routines. Thus, FT can largely run without interference, as MG executes a brief burst of initial communication and then largely tapers off. For this reason, MG is particularly symbiotic with other benchmarks. Rather than having 16 MG processes all competing for network access during initialization, there are only eight. After initialization, it does very little active communication, and its partners profit.

For CG, we believe that the majority of its performance improvement when striped is from reduced L3 cache misses. Only when CG is paired with MG does the percentage of execution spent in communication sharply drop (from 57% to 40%). However, even though the percentage of time CG spends communicating in most coschedules does not change, the communication composition does. Pairing CG with FT causes the time spent in MPI_Wait to decrease by 20 percent. This result is particularly interesting because it reiterates an important point about job striping: job striping can improve performance by reducing LLC contention, communication interference, or a combination thereof. When job striping pairs an application with a large working set and a high LLC cache miss ratio with another application with a small working set and low LLC cache miss ratio, the first application benefits because more of its working set now fits in cache. The second application consequently then suffers more cache misses because less of its working set now fits in cache. However, we observe that many applications with large working sets often spend more time in computation, and applications with smaller working sets usually spend more time doing communication. Applications that exhibit both high communication and high cache contention usually do not scale. Thus, even though the cache-benign application now has a higher LLC miss rate, it is now paired with an application that spends either less time in communication or with a communication pattern distinct from its own.

5.2. *Real Applications*

For real applications, we observe behavior that is similar to that observed with NPBs. MILC's L3 cache misses per 1000 instructions drop significantly when paired with both LAMMPS and MILC. Scheduled compactly, MILC has 1.7 misses per 1000 instructions. When scheduled with GTC or LAMMPS, this figure respectively drops to 1.03 and 0.78. Scheduling using the spreading method decreases this number further to 0.58. If we correlate these numbers to what we see with performance, a trend becomes clear. MILC runs for 513 seconds in compact, 413 seconds paired with GTC, 382 seconds when paired with LAMMPS, and 353 seconds when run in spread. If we fit a linear $y = mx + b$ function to these data (runtime vs. L3 misses per 1000 instructions), we get a coefficient of determination of $r^2 = 0.998$. A coefficient of determination of 1 corresponds to perfectly collinear data, and so the linear model is highly representative of the actual trend in the data. Thus, if we pair MILC on the NERSC input with another application and measure MILC's L3 misses per 1000 instructions, we expect to be able to accurately predict MILC's run time by using our linear model.

The number of cycles without instruction issue also drops when MILC is striped. In the compact configuration, MILC has 22 cycles without instruction issue for every 1000 instructions. Spreading MILC reduces this number to 10, and pairing MILC with GTC and LAMMPS improves this figure to 17 and 13 respectively over the compact baseline. When plotting runtime versus cycles without instruction issue per instruction, $r^2 = 0.964$. While this correlation is slightly weaker, this is to be expected. Depending on where an L3 miss occurs in the code, it may or may not cause the pipeline to stall. When we conduct a linear correlation on L3 misses and cycles without instruction issue per 1000 instructions, $r^2 = 0.976$. Thus we conclude that MILC's speedup when striped is almost entirely tied to a decrease in contention over last level cache. These findings are similar to those found in the SPEC2006 MILC benchmark [7].

GTC benefits the second most from job striping. Like MILC, spreading significantly improves its L3 miss rate per 1000 instructions, while communication as a percentage of runtime remains essentially constant. We believe GTC runs well with MILC because their communication patterns and cache accesses are largely synergistic. GTC spends approximately equal time in MPI_Allreduce,

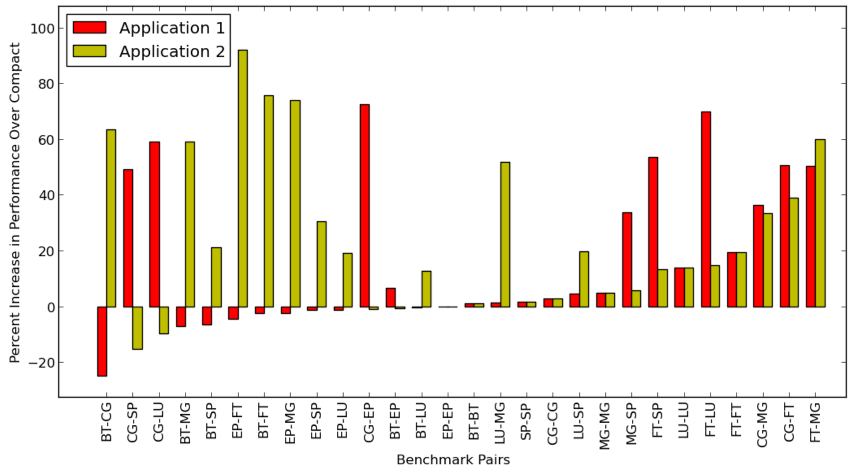

Figure 7: Improvement to scaled throughput per NAS benchmark in each pairing. Pairings are sorted by the worse performer of the pair.

MPI_Sendrecv, and MPI_Barrier. MILC on the other hand spends 66% of its communication time in MPI_Wait and 24% in MPI_Allreduce. Thus for both codes, the majority of communication time is spent waiting for the data to propagate. This explains why spreading does not reduce the communication time percentage for either application, as waiting is not a contentious activity. In terms of cache behavior, GTC makes very few L3 cache accesses. Thus even though MILC's working set in L3 is large, GTC does not suffer.

LAMMPS exhibits very little change in performance when striped with itself and other applications. This is reasonable given that spreading LAMMPS only increases performance by 3%. While we expected the increase in LAMMPS' performance to be more significant due to the marked decrease in L3 misses for LAMMPS when spread, we conclude that these misses must be less detrimental than those incurred by GTC or MILC. In addition to little benefit from reduced cache contention, LAMMPS cannot profit much from improved communication either. In total, LAMMPS spends (7%) of total execution in MPI calls, so that even though pairing LAMMPS with GTC decreases its communication time by 15%, this has a largely inconsequential impact on runtime. We attribute minor benefit to the composition of LAMMPS's communication. LAMMPS spends over 65% of its communication time in bursts of synchronous MPI_Send. By having fewer processes executing MPI_Send at any one time, performance is improved.

Pairing LAMMPS with itself very slightly improves performance, and we attribute this to the communication of each LAMMPS parallel job being out of synchronization with one another. This is consistent with the fact that our communication-intensive or highly synchronous NAS parallel benchmarks also sped up when homogeneously striped.

5.3. Per Process Improvement from Striping

In the previous section, the overall benefit to the system from job striping was demonstrated in terms of both scaled throughput and energy. Figure 7 provides the improvement to runtime per process in each pairing of the NAS benchmarks and Figure 8 provides the improvement per process for our real benchmarks. Although most of these pairings are an overall benefit to the system (see prior section), individual jobs may suffer from the pairing. Approximately half (16/34) of the striping pairs achieve symbiosis—each of the jobs in the pair benefit. 12 out of 34 of the striped pairs experience one of the jobs suffering less than a 5% degradation in performance. Lastly, in 6 out of the 34 pairs, one of the benchmarks suffers by greater than 5%.

Table II groups benchmarks by their gains from being run spread over compact. This produces

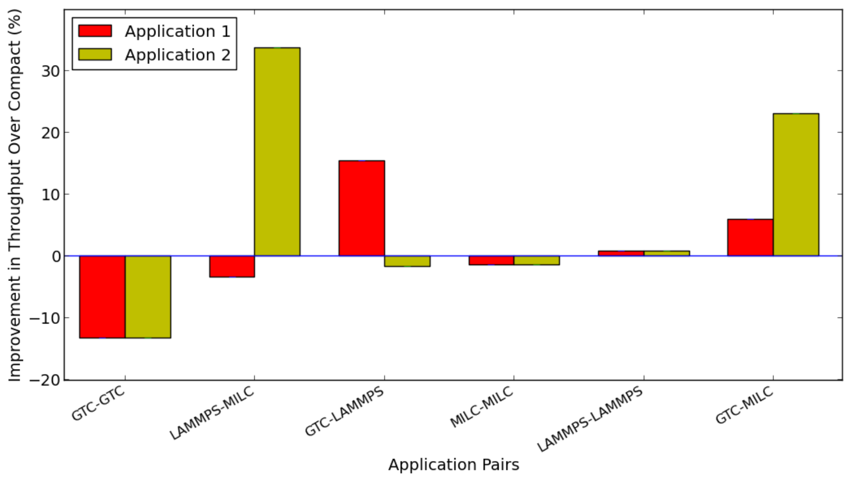

Figure 8: Improvement to scaled throughput per Real benchmark in each pairing. Pairings are sorted by the worse performer of the pair.

Table II: Benchmarks categorized by the reduction in execution time from running spread over running compact.

	Improvement of Spread over Compact		
	High $\geq 40\%$	Medium $\geq 20\%$ and $< 40\%$	Mild $< 20\%$
NAS	CG, FT, MG	LU, SP	BT, EP
Real	MILC		GTC, LAMMPS

Table III: Number of pairs which fall under the outcome of Symbiotic, Minor Interference (one benchmark loss of $< 5\%$), or Non Symbiotic (one benchmark loss of $\geq 5\%$) when pairing benchmarks from different categories of spread benefit (High, Medium, and Low).

Pairing	Symbiotic	Minor Interference ($< 5\%$ Loss)	Moderate Inter. ($\geq 5\%$ Loss)
High-High	6	1	0
High-Medium	4	0	2
High-Low	1	5	2
Medium-Medium	3	0	0
Medium-Low	0	3	1
Low-Low	2	3	1

groups of “High,” “Medium,” and “Low” spread performers. In general, striping with another job will provide worse execution time than if run alone spread. This allows us to use the spread behavior to infer the behavior of running striped. Table III is a combination of Figures 7 and 8 with Table II to investigate the impact of pairing across different groups of benchmarks.

When scheduling a High spread performer for striping, that benchmark always benefits from the striping with the one exception of MILC striped with MILC (where each suffers a minor slowdown). This is fairly intuitive: if the benchmark gains 40% or more running spread alone, it is unlikely (but possible) the interference from the coscheduled job will counteract the large spread gains.

Striping with Medium spread performers is more problematic as they always do well when scheduled with other Medium spread performers, but can suffer when scheduled with High spread performers. Four out of six such pairings of High-Medium are Symbiotic, but two of those six exhibit Moderate Interference. As examples, CG-LU is a High-Medium pair where CG does very well at the expense of LU and the FT-LU pair is a High-Medium pair where both benchmarks benefit from the striping. Medium paired with Low always benefits the Medium spread performer. Although only one such pairing is non-Symbiotic, in three of the four cases the Medium benchmark benefits while the Low benchmark experiences a minor loss.

Striping of Low with Medium or High always benefits the Medium or High spread performer more than the Low. Although in many of these cases (8/11) the Low spread performer only suffers a minor loss in performance, in some cases (3/11) the Low spread performer suffers more substantially. Scheduling pairs of Low spread performers is generally of dubious benefit.

Although the interactions across the different categories of High, Medium, and Low spread performers can cause some complications, a simple trend remains true. When scheduling a High with Medium or Low, High ends up getting the better benefit. When scheduling Medium with Low, Medium receives the majority of the benefit of striping. We will address how one might use this information to address fairness concerns in the following section.

6. FAIRNESS

As we have previously identified, striping can lead to colocations where one application speeds up relative to compact while the other slows down. Different systems have different models of charging users for use and each of these different models may choose to handle fairness concerns related to job striping differently. For example, in highly collaborative systems where all users value overall system throughput over individual job latencies, job striping could be enacted uniformly with everyone recognizing the value. In contrast, in systems where users are charged by the CPU hour, a more fair mechanism may be necessary. We offer the following suggestions which may be used individually or in concert with one another:

- Separate job queues could be provided. Users interested in running striped could submit to that queue.
- Submission to the striped job queue could be incentivized by the system administration in the form of additional running hours or reduced cost. In this study, reducing the cost to users by 23% would be enough to guarantee that no user paid more in the striped queue versus running their job in a traditional queue. This cost reduction would be equivalent to passing on the efficiency savings from running NAS-like codes on to the user.
- Users informed of the implications of spread performance on striped performance could use that information to aid in fairness decisions. For example, high spread-benefit jobs might be charged extra for submission to the striped job queue and low spread-benefit jobs might be charged less. When those jobs are paired, each user is charged a more "fair" value and the overall system benefits.
- Future work may develop striped performance prediction which could be used to inform the accounting mechanism. A generalization of techniques such as those used in [6,7,34] is likely to be effective.

Fairness concerns vary by system and may conflict with concerns for overall system performance. Each system may need to re-address these concerns in light of job striping. This work does not aim to provide a panacea for these potential concerns but to instead inform the reader of the value of job striping and of potential mechanisms to aid in fairness decisions.

7. FUTURE WORK: A JOB STRIPING COSCHEDULER

This work exposes the performance opportunity afforded by large scale job colocation. As a logical next step, we are investigating the two primary challenges to making colocation feasible: a precise mechanism for deciding colocations and an accounting system that is fair in the presence of colocation. Once we have developed adequate solutions, we will consider modifying an existing scheduler to support job striping. It is likely that this modified scheduler, as part of the GreenQueue project at San Diego Supercomputer Center, will leverage machine learning, application idiom classification [35], and large scale dynamic voltage frequency scaling to deliver high throughput and energy efficiency [36].

8. RELATED WORK

This study is the only one to our knowledge that combines use of real scientific applications, power measurement and analysis, and coscheduling at thousands of cores into a single work. Previous studies on CMP-based systems have largely focused on single server coscheduling with multithreaded benchmarks or commercial applications.

The study that most resembles our own is by Koop et al [22]. They show that by colocating pairs of NAS parallel benchmarks across several machines, performance can be improved by reducing communication contention. They present data that shows that symbiotic applications communicate during largely disjoint time intervals.

Snavely and Tullsen proposed symbiotic scheduling for SMT processors [5]. Weinberg and Snavely evaluate the potential of symbiotic workload space sharing on an HPC platform [37] and the users' ability to accurately determine resource bottlenecks on that platform [38]. Paired gang scheduling proposes pairing I/O bound jobs with compute intensive jobs for better overall throughput in HPC [39].

Shared levels of cache have long been recognized as a point of contention. Yan and Zhang aim to predict worst case run-times by using profiled control flow information to predict i-cache contention between two threads [40]. Anderson et al. evaluated grouping processes by L2 miss ratios to avoid coscheduling those with high miss ratios, for real-time tasks on multicores [41]. Fedorova et al. recognize the potential of using heuristics for better scheduling on SMT processors. They use predicted L2 miss ratios based on reuse distance [42] to inform their scheduler on in-order, simulated, SMT processors [43]. StatCC was developed to predict shared cache miss ratios and subsequently coscheduled thread CPI [44]. StatCC evaluated 2-thread coschedule prediction against results from an in-order processor simulator. Blagodurov et al. recently evaluated various classification metrics for thread coscheduling [8–10]. Based on their evaluation, they used performance counters to derive metrics to aid in scheduling as to reduce LLC cache contention [7].

Cache partitioning for Quality of Service and fairness has been evaluated for SMT processors [45] as well as CMPs [46–48]. Hardware support for QoS or Fairness has limitations—namely higher expense, less flexibility, and longer time to market, and proposed software solutions often require dedicated time slicing to ensure disadvantaged threads make fair progress on existing systems [49].

More recent work by Iancu et al. in [19] looked at the benefit of over-subscription of multicore processors for various implementations of the NAS parallel benchmarks. They too found heterogeneity to be beneficial.

In the commercial domain, there has been some recent work on coscheduling. Tang et al. investigate how the performance of Google's workloads changes when using different thread-to-core mappings [50]. Mars et al. present Bubble-Up, a mechanism that can very precisely trade a small decrease in the QoS of certain applications for significant gains in data center utilization [34]. In addition to these works, the multi-tiered Déjà Vu system provides a practical solution for intelligent colocation of virtual machines by combining a lookup table of virtual machine signatures, a clustering algorithm, and a small testbed of machines for periodic profiling when the runtime system's assumptions are violated [51].

9. CONCLUSION

This work has shown that there is tangible benefit in placing heterogeneous distributed applications from different users on the same set of shared resources. We validated the job striping approach at large scales on NAS parallel benchmarks and on three real applications and realized performance benefit both in terms of throughput and energy efficiency.

Random heterogeneous coschedules of 1024 process NAS parallel benchmarks improved throughput by 26% and energy efficiency by 22%. If we could select the best coschedule for each benchmark, throughput and energy efficiency improved further to 31% and 26% over the baseline.

For our real applications, we have shown that GTC and MILC benefited significantly from heterogeneous coscheduling and that LAMMPS boosts the throughput of any coschedule. MILC's throughput when striped with different applications increases by an average of 26%; in addition, we have illustrated that MILC's improvement from job striping is highly correlated to reductions in L3 misses and cycle stalls.

Furthermore, we have characterized how job striping works. Job striping improves the throughput of some applications by reducing communication contention (BT and FT). On others such as CG, MG and MILC, it reduces LLC misses. A combination of communication and LLC contention is also observed for some applications (LU and SP).

Job striping's performance can be predicted by examining how an application's traits change when spread. Applications that exhibit marginal benefit from spreading do not exhibit performance benefit from striping.

We have also shown that job striping yields reliable performance on real applications. Striped jobs that run over similar length time intervals exhibit performance variation that is no higher than compactly scheduling jobs in the traditional manner.

Going forward, we propose extensive reexamination, validation, and testing of the existing software infrastructure. Our work shows that a simple change in the scheduling framework can produce a significant benefit. We are certain that further examination of other established practices will reveal additional opportunities for optimization.

10. ACKNOWLEDGEMENTS

The authors dedicate this work to the loving memory of Dr. Allan Snavely. In addition, the authors thank the anonymous reviewers for their constructive feedback as well as the following outstanding individuals at SDSC and Calit2: Jeff Bennett, Pietro Cicotti, Jerry Greenberg, Adam Jundt, Mitesh Meswani, Wayne Pfeiffer, Bob Sinkovits, Shawn Strande, Mahidhar Tatineni, Nicole Wolter and Kenneth Yoshimoto. Your advice and support has been invaluable.

This research has been supported in part by the DOE Office of Science through the Advanced Scientific Computing Research (ASCR) award titled "Thrifty: An Exascale Architecture for Energy-Proportional Computing," National Science Foundation grant: OCI-0910847, Gordon: A Data Intensive Supercomputer, NSF Grant CCF-0702349, NSF grant CCF-1018356, the UCSD Computer Science and Engineering Department, the Reserve Officers Association Henry J. Reilly Memorial Scholarship, and Semiconductor Research Corporation Grant 2005-HJ-1313. The authors also acknowledge the support of the Multiscale Systems Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation program.

REFERENCES

1. Bergman K, Borkar S, Campbell D, Carlson W, Dally W, Denneau M, Franzon P, Harrod W, Hiller J, Karp S, *et al.*. Exascale computing study: Technology challenges in achieving exascale systems wwwcsendedu/Reports/2008TR-2008-13pdf, 2008.
2. Beckman P. Looking toward exascale computing. *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, IEEE, 2008; 3–3.
3. Cappello F. Fault tolerance in petascale/exascale systems: Current knowledge, challenges and research opportunities. *International Journal of High Performance Computing Applications* 2009; **23**(3):212–226.
4. Shalf J, Dosanjh S, Morrison J. Exascale computing technology challenges. *High Performance Computing for Computational Science–VECPAR 2010* 2011; :1–25.
5. Snavely A, Tullsen DM. Symbiotic jobscheduling for a simultaneous multithreaded processor. *9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000.
6. Porter L. Single threaded performance in the multi-core era. PhD Thesis, University of California, San Diego 2011.
7. Blagodurov S, Zhuravlev S, Fedorova A. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems* 2010; **28**.
8. Chandra D, Guo F, Kim S, Solihin Y. Predicting inter-thread cache contention on a chip multi-processor architecture. *11th International Symposium on High-Performance Computer Architecture*, 2005.

9. Knauerhase R, Brett P, Hohlt B, Li T, Hahn S. Using OS observations to improve performance in multicore systems. *IEEE Micro 28, 3* 2008; .
10. Xie Y, Loh GH. Dynamic classification of program memory behaviors in cmps. *CMP-MSI*, 2005.
11. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, Fatoohi RA, Frederickson PO, Lasinski TA, Schreiber RS, *et al.*. The nas parallel benchmarkssummary and preliminary results. *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, ACM: New York, NY, USA, 1991; 158–165.
12. Lin Z, Rewoldt G, Ethier S, Hahm TS, Lee WW, Lewandowski JLV, Nishimura Y, Wang WX. Particle-in-cell simulations of electron transport from plasma turbulence: recent progress in gyrokinetic particle simulations of turbulent plasmas. *Journal of Physics: Conference Series* 2005; **16**(1):16.
13. Large-scale Atomic/Molecular Massively Parallel Simulator. `http://lammps.sandia.gov/`.
14. MIMD Lattice Computation (MILC) Collaboration. `http://www.physics.indiana.edu/~sg/milc.html`.
15. Antypas K, Shalf J, Wasserman H. Nersc-6 workload analysis and benchmark selection process. *Technical Report*, Lawrence Berkeley National Laboratory 2008. URL `http://escholarship.org/uc/item/7qd192rr`.
16. Broquedis F, Furmento N, Goglin B, Wacrenier PA, Namyst R. Forestgomp: An efficient openmp environment for numa architectures. *International Journal of Parallel Programming* 2010; **38**:418–439.
17. Team M. Mvapich2 1.8 user guide 2012. URL `http://mvapich.cse.ohio-state.edu/support/mvapich2-1.8_user_guide.pdf`.
18. Gordon user guide 2012. URL `http://www.sdsc.edu/us/resources/gordon/`.
19. Iancu C, Hofmeyr S, Blagojevic F, Zheng Y. Oversubscription on multicore processors. *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, 2010; 1 –11.
20. Pusukuri K, Gupta R, Bhuyan L. Thread reinforcer: Dynamically determining number of threads via os level monitoring. *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, 2011; 116 –125.
21. Sayeed M, Bae H, Zheng Y, Armstrong B, Eigenmann R, Saied F. Measuring high-performance computing with real applications. *Computing in Science and Engg.* Jul 2008; **10**(4):60–70.
22. Koop M, Luo M, Panda D. Reducing network contention with mixed workloads on modern multicore, clusters. *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, IEEE, 2009; 1–10.
23. Norman M, Snavely A. Accelerating data-intensive science with Gordon and Dash. *2010 TeraGrid Conference*, 2010.
24. Broquedis F, Clet-Ortega J, Moreaud S, Furmento N, Goglin B, Mercier G, Thibault S, Namyst R. hwloc: a generic framework for managing hardware affinities in hpc applications. *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, IEEE, 2010; 180–186.
25. NAS. Nas parallel benchmarks website http://wwwnasnasagov/Resources/Software/npbhtml.
26. Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Fatoohi RA, Frederickson PO, Lasinski TA, Simon HD, Venkatakrishnan V, *et al.*. The nas parallel benchmarks. *Technical Report*, The International Journal of Supercomputer Applications 1991.
27. Asc sequoia benchmark codes. URL `https://asc.llnl.gov/sequoia/benchmarks/`.
28. Coskun AK, Strong R, Tullsen DM, Simunic Rosing T. Evaluating the impact of job scheduling and power management on processor lifetime for chip multiprocessors. *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS '09, ACM: New York, NY, USA, 2009; 169–180.
29. Wright P, Snavely. Characterizing parallel scaling of scientific applications using ipm. *The 10th LCI International Conference on High-Performance Clustered Computing*, 2009.
30. Tikir M, Laurenzano M, Carrington L, Snavely A. Psins: An open source event tracer and execution simulator. *DoD High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2009*, 2009; 444 –449.
31. Browne S, Dongarra J, Garner N, London K, Mucci P. A scalable cross-platform infrastructure for application performance tuning using hardware counters. *Supercomputing, ACM/IEEE 2000 Conference*, 2000; 42.
32. McVoy L, Staelin C. lmbench: portable tools for performance analysis. *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, USENIX Association: Berkeley, CA, USA, 1996; 23–23.
33. Ruggiero J. Measuring cache and memory latency and cpu to memory bandwidth. Intel Whitepaper 2008. URL `http://download.intel.com/design/intarch/papers/321074.pdf`.
34. Mars J, Tang L, Hundt R, Skadron K, Soffa ML. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. *MICRO '11: Proceedings of The 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ACM: New York, NY, USA, 2011.
35. Olschanowsky C, Snavely A, Meswani MR, Carrington L. Pir: Pmac's idiom recognizer. *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops*, ICPPW '10, IEEE Computer Society: Washington, DC, USA, 2010; 189–196.
36. Tiwari A, Laurenzano M, Peraza J, Carrington L, Snavely A. Green queue: Customized large-scale clock frequency scaling. *In the proceedings of the International Conference on Cloud and Green Computing (CGC 2012)*, 2012.
37. Weinberg J, Snavely A. Symbiotic space-sharing on sdsc's datastar system. *12th Workshop on Job Scheduling Strategies for Parallel Processing*, 2006.
38. Weinberg J, Snavely A. User-guided symbiotic space-sharing of real workloads. *20th ACM International Conference on Supercomputing*, 2006.
39. Wiseman Y, Feitelson D. Paired gang scheduling. *Parallel and Distributed Systems, IEEE Transactions on* June 2003; **14**(6).
40. Yan J, Zhang W. WCET analysis for multi-core processors with shared l2 instruction caches. *IEEE Real-Time and Embedded Technology and Applications Symposium* 2008; .
41. Anderson J, Calandrino J, Devi U. Real-time scheduling on multicore platforms. *12th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2006.
42. Berg E, Hagersten E. StatCache: a probabilistic approach to efficient and accurate data locality analysis.

International Symposium on Performance Analysis of Systems and Software, 2004.
43. Fedorova A, Seltzer M, Small C, Nussbaum D. Performance of multithreaded chip multiprocessors and implications for operating system design. *USENIX Annual Technical Conference*, 2005.
44. Eklov D, Black-Schaffer D, Hagersten E. Fast modeling of shared caches in multicore systems. *6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011.
45. Cazorla FJ, Knijnenburg PM, Sakellariou R, Fernández E, Ramirez A, Valero M. Predictable performance in SMT processors. *1st Conference on Computing Frontiers*, 2004.
46. Guan N, Stigge M, Yi W, Yu G. Cache-aware scheduling and analysis for multicores. *7th ACM International Conference on Embedded software*, 2009.
47. Kim S, Chandra D, Solihin Y. Fair cache sharing and partitioning in a chip multiprocessor architecture. *13th International Conference on Parallel Architecture and Compilation Techniques*, 2004.
48. Lin J, Lu Q, Ding X, Zhang Z, Zhang X, Sadayappan P. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. *14th International Symposium on High Performance Computer Architecture*, 2008.
49. Fedorova A, Seltzer M, Smith MD. Improving performance isolation on chip multiprocessors via an operating system scheduler. *16th International Conference on Parallel Architecture and Compilation Techniques*, 2007.
50. Tang L, Mars J, Vachharajani N, Hundt R, Soffa ML. The impact of memory subsystem resource sharing on datacenter applications. *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, ACM: New York, NY, USA, 2011; 283–294.
51. Vasić N, Novaković D, Miučin S, Kostić D, Bianchini R. Dejavu: accelerating resource allocation in virtualized environments. *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ACM, 2012; 423–436.