Oversubscription on Multicore Processors

Costin Iancu, Steven Hofmeyr, Filip Blagojević, Yili Zheng
Lawrence Berkeley National Laboratory
Berkeley, USA
{cciancu,shofmeyr,fblagojevic,yzheng}@lbl.gov

Abstract—**Existing multicore systems already provide deep levels of thread parallelism; hybrid programming models and composability of parallel libraries are very active areas of research within the scientific programming community. As more applications and libraries become parallel, scenarios where multiple threads compete for a core are unavoidable. In this paper we evaluate the impact of task oversubscription on the performance of MPI, OpenMP and UPC implementations of the NAS Parallel Benchmarks on UMA and NUMA multi-socket architectures. We evaluate explicit thread affinity management against the default Linux load balancing and discuss sharing and partitioning system management techniques. Our results indicate that oversubscription provides beneficial effects for applications running in competitive environments. Sharing all the available cores between applications provides better throughput than explicit partitioning. Modest levels of oversubscription improve system throughput by 27% and provide better performance isolation of applications from their co-runners: best overall throughput is always observed when applications share cores and each is executed with multiple threads per core. Rather than "resource" symbiosis, our results indicate that the determining behavioral factor when applications share a system is the granularity of the synchronization operations.**

I. Introduction

The pervasiveness of multicore processors in contemporary computing systems will increase the demand for techniques to adapt application level parallelism to the available hardware parallelism. Modern systems are increasingly asymmetric in terms of both architecture (e.g. Intel Larrabee or GPUs) and performance (e.g. Intel Nehalem Turbo Boost). Parallel applications often have restrictions on the degree of parallelism and threads might not be evenly distributed across cores, e.g. square number of threads. Furthermore, applications using hybrid programming models and concurrent execution of parallel libraries are likely to become more prevalent. Recent results illustrate the benefits of hybrid MPI+OpenMP [1], [2] or MPI+UPC [3] programming for scientific applications in multicore environments. In this setting MPI processes share cores with other threads. Calls to parallel scientific libraries are also likely to occur [4] in consumer applications that are executed in competitive environments on devices where tasks are forced to share cores. As more and more applications and libraries become parallel, they will have to execute in asymmetric (either hardware or load imbalanced) competitive environments and one question is how well existing programming models and OS support behave in these situations. Existing runtimes for parallel scientific computing are designed based on the implicit assumption that applications run with one Operating System level task per core in a dedicated execution environment and this setting will provide best performance.

In this paper we evaluate the impact of oversubscription on end-to-end application performance, i.e. running an application with a number of OS level tasks larger than the number of available cores. We explore single and multi-application usage scenarios with system partitioning or sharing for several implementations of the NAS Parallel Benchmarks [5] (UPC, MPI+Fortran and OpenMP+Fortran) on multi-socket NUMA (AMD Barcelona and Intel Nehalem) and UMA (Intel Tigerton) multicore systems running Linux. To our knowledge, this is the first evaluation of oversubscription, sharing and partitioning on non-simulated hardware at the core concurrency levels available today. Our results indicate that while task oversubscription sometimes affects performance in dedicated environments, it is beneficial for the performance of parallel workloads in competitive environments, irrespective of the programming model. Modest oversubscription (2,4,8 tasks per core) improves system throughput by 27% when compared to running applications in isolation and in some cases end-to-end performance (up to 46%). For the multi-socket systems examined, partitioning sockets between applications in a combined workload almost halves the system throughput, regardless of the programming paradigm. Partitioning [6]–[8] has been increasingly advocated recently and this particular result provides a cautionary tale when considering scientific workloads. Oversubscription provides an easy way to reduce the performance impact of co-runners and it is also a feasible technique to allocate resources to parallel applications: there is correlation between the number of threads present in the system and the observed overall performance.

When examining an isolated application, our results indicate that the average inter-barrier interval is a good predictor of its behavior with oversubscription. Fine grained applications (few ms inter-barrier intervals) are likely to see performance degradation, while coarser grained applications speed-up or are not affected. The observed behavior is architecture or programming model independent. When examining co-scheduling of applications, again the synchronization granularity is an accurate predictor of their behavior. The results presented in this study suggest that symbiotic scheduling of parallel scientific applications on multicore systems depends on the synchronization behavior, rather than on cache or CPU usage.

One of the more surprising results of this study is that, irrespective of the programming model, best overall system throughput is obtained when each application is allowed to run with multiple threads per core in a non-dedicated environment.

The throughput improvements provided by oversubscription in a "symbiosis" agnostic manner match or exceed the improvements reported by existing studies of co-scheduling on multicore or shared memory systems. As discussed in Section VIII, our evaluation of OpenMP oversubscription exhibits different trends than previously reported.

Our results are obtained only when enforcing an even thread distribution across cores at program startup; oversubscription degrades performance in all cases where the default Linux load balancing is used. For the applications considered, an even initial task distribution seems to be the only requirement for good performance, rather than careful thread to core mappings. The performance results for experiments where threads are randomly pinned to cores are statistically indistinguishable.

II. Oversubscription and Performance

When threads share cores several factors impact the end-to-end application performance: 1) context switching; 2) load balancing; 3) inter-thread synchronization overhead and 4) system partitioning. The impact of context switching is determined by the direct OS overhead of scheduling and performing the task switching and by the indirect overhead of lost hardware state (locality): caches and TLBs. Li et al [9] present microbenchmark results to quantify the impact of task migration on modern multicore processors. Their results indicate overheads with granularities ranging from few microseconds for CPU intensive tasks to few milliseconds for memory intensive tasks; for reference, the scheduling time quanta is around $100ms$. Our application results also indicate that context switching and loss of hardware state do not significantly impact performance for the observed applications: therefore in the rest of this paper we do not quantify these metrics.

Load balancing determines the "spatial" distribution of tasks on available cores and sometimes tries to address locality concerns. Recent studies advocate explicit thread affinity management [10] using the `sched_setaffinity` system call and better system load balancing [11], [12] mechanisms. Explicitly managing thread affinity can lead to non-portable implementations since it cannot accommodate un-even task distributions. In [12] we present a user level load balancer able to provide scalable performance for arbitrary task distributions: those results also substantiate the fact that the impact of "hardware" locality can be ignored. For conciseness reasons, in this paper we present results for experiments performed with even task distributions across cores and concentrate the exposition on the impact of synchronization behavior and system partitioning.

Parallel applications have data and control dependencies and threads will "stall" waiting for other threads. In these cases oversubscription has the potential to improve application performance with better load balancing and CPU utilization. On the other hand, increasing the number of threads has the potential to increase the duration of synchronization operations due to greater operational complexity and hardware contention. The implementation of these operations often interacts with the per-core task scheduler through the `sched_yield` system call which influences the temporal ordering of task execution. In order to increase CPU utilization and system responsiveness, most implementations use a combination of polling and task yielding rather than blocking operations. Similar techniques are used in communication libraries when checking for completion of outstanding operations.

System partitioning answers the question of how should multicore systems be managed: threads either competitively share a set of cores or can be isolated into their own runtime partitions. Several [6]–[8], [13] research efforts advocate for system partitioning as a way of improving both performance and isolation.

We evaluate the performance in the presence of oversubscription of three programming models: MPI, UPC and OpenMP. MPI is the dominant programming paradigm for parallel scientific applications and UPC [14] belongs to the emerging family of Partitioned Global Address Space languages. MPI uses OS processes while UPC uses `pthreads` and exhibits a lower context switch cost. Both UPC and MPI programs perform inter-task synchronization in barrier operations while MPI programs might also synchronize in Send/Rcv pairs. OpenMP runs with `pthreads` and performs barrier synchronization when exiting parallel regions and might synchronize inside parallel regions. The three runtimes considered yield inside synchronization operations in the presence of oversubscription. Hybrid implementations using combinations of these programming models are already encountered in practice [1]–[3].

III. Experimental Setup

We experimented on the multicore architectures shown in Table I. The *Tigerton* system is a UMA quad-socket, quad-core Intel Xeon where each pair of cores shares an L2 cache and each socket shares a front-side bus. The *Barcelona* system is a NUMA quad-socket, quad-core AMD Opteron where cores within a socket share an L3 cache. The *Nehalem* system is a dual-socket quad-core Intel Nehalem system with hyperthreading; each core supports two hardware execution contexts. All systems run recent Linux kernels (2.6.28 on the *Tigerton*, 2.6.27 on the *Barcelona* and 2.6.30 on the *Nehalem*).

We use implementations of the NAS Parallel Benchmarks (NPB), classes S, A, B and C: version 2.4 for UPC [15] and 3.3 for OpenMP and MPI [5]. All programs (CG, MG, IS, FT, EP, SP) have been compiled with the Intel 10.1 compilers (`icc, ifort`) which provide good serial performance on all architectures. The UPC benchmarks are compiled with the Berkeley UPC 2.8.0 compiler which uses `-O3` for the `icc` back-end compiler, while the Fortran benchmarks were compiled with `-fast`, which includes `-O3`. Unless specified otherwise, OpenMP is compiled and executed with `static` scheduling. We have used MPICH 2 on all architectures. Due to space restrictions we will not discuss the details of the NAS benchmarks (for a detailed discussion kindly see [5]).

Asanović et al [4] examined six different promising domains for commercial parallel applications and report that a surprisingly large fraction of them use methods encountered in the scientific domain. In particular, all methods used in the NAS benchmarks appear in at least one commercial domain.

	Processor	Clock GHz	Cores	L1 data/instr	L2 cache	L3 cache	Memory/core	NUMA
Tigerton	Intel Xeon E7310	1.6	16 (4x4)	32K/32K	4M / 2 cores	none	2GB	no
Barcelona	AMD Opteron 8350	2	16 (4x4)	64K/64K	512K / core	2M / socket	4GB	socket
Nehalem	Intel Xeon E5530	2.4	16 (2x4x2)	32K/32K	256K / core	8M / socket	1.5G / core	socket

TABLE I
Test systems.

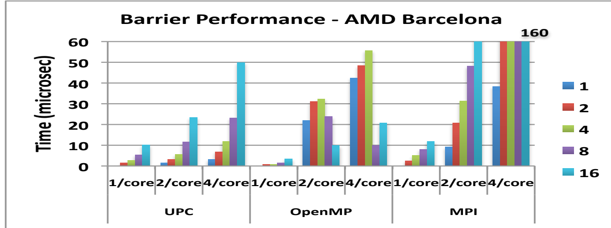


Fig. 1. *Barrier performance with oversubscription at different core counts (legend) on AMD Barcelona. Similar results are observed on all systems.*

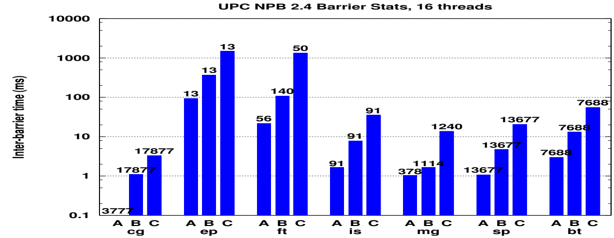


Fig. 2. *Average time between two barriers and barrier count for the UPC benchmarks on* Nehalem. *Similar trends are observed on all systems and implementations.*

The execution time across all benchmarks ranges from a few seconds to hundreds of seconds while the memory footprints range from few MB to GB. For example, the data domain in FT class C is a grid of size 512*x*512*x*512 of complex data points: this amounts to more than 2GB of data. Thus, we have a reasonable sample of short and long lived applications and a sample of small and large memory footprints. For all benchmarks, we compare executions using the default Linux load balancing with explicit thread affinity management, referred to as PIN. For PIN we generate random initial pinnings to capture different initial conditions and thread interactions and pin tasks to cores using the `sched_setaffinity` system call at the beginning of program execution. To capture variation in execution times each experiment has been repeated five or more times; most experiments have at least 15 settings for the initial task layout. As a performance metric we use the average benchmark running time across all repetitions.

IV. Benchmark Characteristics

Figure 1 presents the behavior of barrier implementations for the three programming models in the presence of oversubscription: increasing the number of threads per core increases the barrier latency from few μs to tens of μs. The MPI oversubscribed barrier latency is greater than UPC and OpenMP due to the more expensive process context switch. Note that these microbenchmark results provide a lower bound for the barrier latency when used in application settings. UPC and MPI call `sched_yield` inside barriers when oversubscribed. The Intel OpenMP runtime provides a tunable implementation controlled by the KMP_BLOCKTIME environment variable. Unless specified otherwise, all results use the default behavior of polling for 200 ms before threads `sleep`. The other relevant settings are KMP_BLOCKTIME=0 where threads sleep immediately and is designed for sharing the system with other applications and KMP_BLOCKTIME=*infinite* where threads never sleep and is designed for dedicated system use. The barrier results with the default setting are representative for the other two settings.

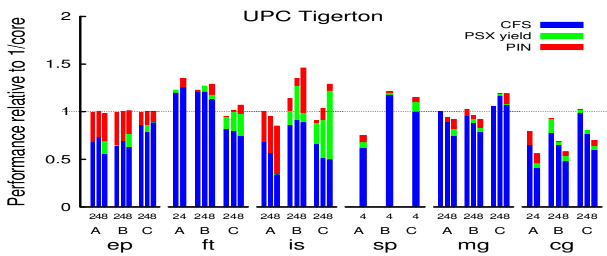
Figure 2 presents the synchronization behavior of the UPC implementations on *Nehalem* : the height of the bars indicates the average time between two barriers in ms, while the labels show the number of executed barriers. The OpenMP and MPI results are very similar and are omitted for brevity. Profiling data shows that most benchmarks exhibit a multi-modal inter-barrier interval distribution, however, as our results indicate, the average interval is a robust predictor for the behavior with oversubscription. We also omit the data about application memory footprints (see [5] for more details).

All implementations exhibit a good load balance: the UPC and MPI implementations have been developed for clusters and have an even domain decomposition, the OpenMP implementations distribute loops evenly across threads.

V. Scalability and Oversubscription

In this section we discuss the effects of oversubscription (up to eight tasks per core) on end-to-end benchmark performance in a *dedicated* environment: each benchmark is run by itself. All benchmarks in the workload (class A,B,C) scale up to 16 cores on all systems. Figures 3, 4, 7 and 8 present selected results for all workloads. For each benchmark we present performance normalized to the performance of the experiment with one thread per core: values greater than one indicate performance improvements. The total height of the bars indicates the behavior when tasks are evenly distributed and explicitly pinned to cores at program startup.

The UPC workload is not affected by oversubscription. The average performance of the whole workload decreases or increases by -2% and 2% respectively, depending on the number of threads per core. We observe several types of behavior when considering individual benchmarks. EP, which is computationally intensive and very coarse grained, is oblivious to oversubscription. This also indicates that the OS overhead for managing the increased thread parallelism is not prohibitive. Oversubscription improves performance for FT and IS with a maximum improvement of 46%. The performance improvements increase with the degree of oversubscription. As the problem size increases, the synchronization granularity of SP



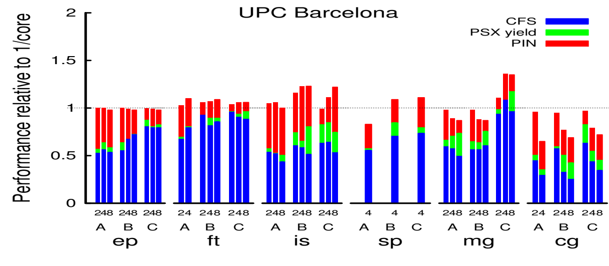
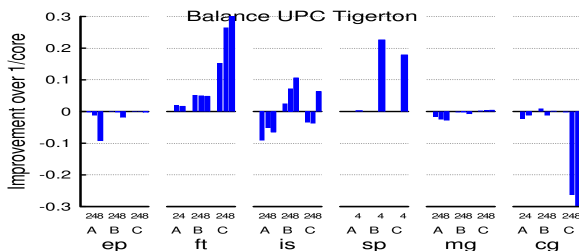


Fig. 3. *UMA oversubscription UPC. Performance is normalized to that of experiments with 1 task per core. Number of tasks per core can be 2, 4 or 8. SP requires a square number of threads. Overall workload performance varies from -2% to 2%.*

Fig. 4. *NUMA oversubscription UPC. Performance is normalized to that of experiments with 1 task per core. Number of tasks per core can be 2, 4 or 8. SP requires a square number of threads. Overall workload performance varies from -2% to 2%.*



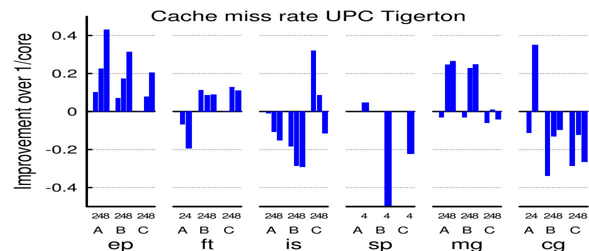
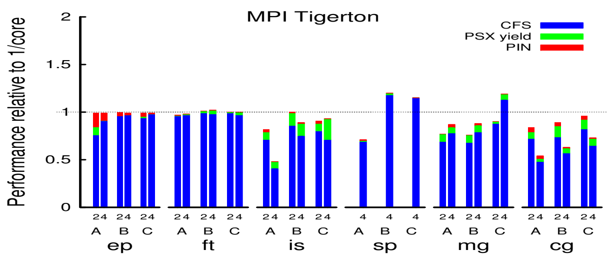


Fig. 5. *Changes in balance on UMA, reported as the ratio between the lowest and highest user time across all cores compared to the 1/core setting.*

Fig. 6. *Changes in the total number of cache misses per 1000 instructions, across all cores compared to 1/core. The EP miss rate is very low.*



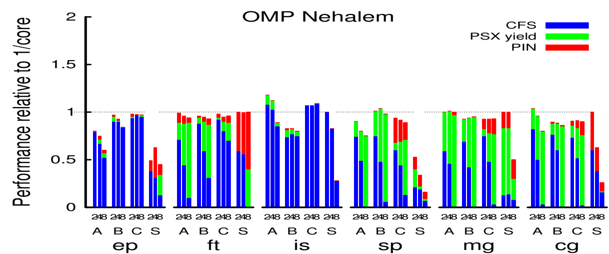
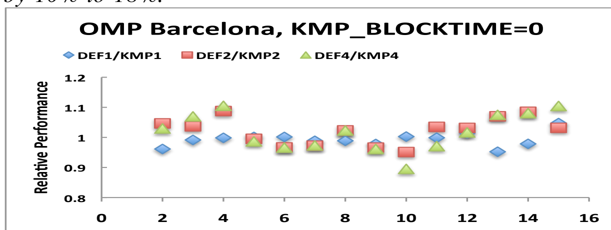


Fig. 7. *UMA oversubscription MPI. Performance is normalized to that of experiments with 1 task per core. Number of tasks per core can be 2 or 4. Overall workload performance decreases by 10% to 18%.*

Fig. 8. *NUMA oversubscription OpenMP. Performance is normalized to that of experiments with 1 task per core. Number of tasks per core can be 2, 4 or 8. Workload performance decreases by 6% to 14%.*



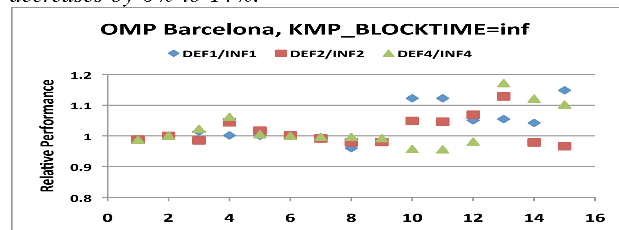


Fig. 9. *OpenMP performance with cooperative synchronization on* Barcelona. *DEF1/KMP1 stands for the default/kmp=0 value with one thread per core. Values greater than 1 indicate performance improvement.*

Fig. 10. *OpenMP performance with "non-cooperative" synchronization on* Barcelona. *DEF1/INF1 stands for the default/kmp=inf value with one thread per core. Values greater than 1 indicate performance improvement.*

and MG also increases and oversubscription is able to provide better performance; increasing the degree of oversubscription enhances the respective trend. CG performance proportionally decreases with the degree of oversubscription, with a maximum slowdown of 44%. Note that all benchmarks where performance degrades with oversubscription are characterized by a short inter-barrier interval (e.g. 1-5 ms on *Nehalem*) and a large number of barrier operations.

The MPI workload is affected more by oversubscription and overall workload performance decreases by 10% when applications are run with two threads per core. Again, most benchmarks are oblivious to oversubscription and performance degradation is observed only for the very fine grained benchmarks. The UPC and MPI implementations use similar domain decomposition for each benchmark and some of the differences from the UPC workload could be attributed to both higher task context switch cost (MPI uses processes, while UPC uses `pthreads`) and to the higher overhead of the MPI barriers with oversubscription. We isolate the impact of context switching by running the UPC workload with processes and shared memory inter-process communication mechanisms: this behavior remains almost identical when running with processes, therefore we attribute any different trends in the MPI behavior to the barrier performance.

The OpenMP behavior on *Nehalem* is presented in Figure 8, similar behavior is observed on the other architectures. Oversubscription slightly decreases overall (classes A,B,C) throughput, again due to the decrease in performance for the fine grained benchmarks. For reference, we also present the behavior of class S. For this problem size the base implementations scale poorly with the increase in cores even when executing with one thread per core. The results presented are with the default setting of KMP_BLOCKTIME and OMP_STATIC. The OpenMP runtime concurrency and scheduling can be changed using the environment variables OMP_DYNAMIC and OMP_GUIDED. We have experimented with these settings, but best performance for this implementation of the NAS benchmarks is obtained OMP_STATIC. Liao et al [16] also report better performance when running OpenMP with a `static` number of threads, evenly distributed. Enabling dynamic adjustment of the number of threads does not affect the overall trends when oversubscribing.

The synchronization behavior of OpenMP can be adjusted using the variable KMP_BLOCKTIME. A setting of KMP_BLOCKTIME=0 forces threads to sleep immediately inside synchronization operations for a more cooperative behavior. This setting determines a slight decrease in performance when running with one thread per core, but it improves performance when oversubscribing. Figure 9 presents the performance compared to the default setting. As illustrated, the performance of the coarse grained benchmarks is not affected, while for the fine grained benchmarks we observe improvements as high as 10%. Figure 10 presents the performance with the setting KMP_BLOCKTIME=*infinite*, where threads are less cooperative and never sleep. This setting provides best overall performance for the OpenMP implementations.

All of our results and analysis indicate that the best predictor of application behavior when oversubscribing is the average inter-barrier interval. Applications with barriers executed every few ms are affected, while coarser grained applications are oblivious or their performance improves.

In order to gain more insight about application behavior in the presence of oversubscription we have also collected data from hardware performance counters (cache miss, total cycles, TLB) and detailed scheduling information from the system logs (user, system time, number of thread migrations, virtual memory behavior). Some of these metrics directly capture the benefits of oversubscription and we illustrate the observed UPC behavior on *Tigerton*. Figure 5 presents the ratio between the lowest and the highest amount of observed user time across all cores normalized to the ratio for the execution with one thread per core. This measure captures the variation of core utilization with oversubscription. Oversubscription improves CPU utilization for FT (all classes), IS-A and CG (A and C). Figure 6 presents the relative behavior of the L2 cache, which is shared on *Tigerton*. We report the normalized total number of cache misses per 1000 instructions across all cores. The results are normalized to the execution with one thread per core. Oversubscription improves memory behavior for the behavior of IS-C and MG (A and B). The behavior of the remaining benchmarks could be explained by a combination of these two metrics.

The UPC performance trends capture another potential benefit of oversubscription: it decreases resource contention and serialization of operations. The benchmarks where performance improves (FT, IS, SP) are characterized by resource contention. They all contain a hand coded communication phase where each thread transfers large amounts of data from the memory space of all (FT,IS) or many (SP) other threads. This portion of the code is written in such a manner that all transfers start from the same thread and proceed in sequence (0, 1, 2...). Oversubscription decreases the granularity of these contending transfers and allows for less serialization. In all three benchmarks, most of the performance improvements occur in these particular parts of the code: this also accounts for the better CPU utilization. Note that this behavior also explains the better improvements of UPC on *Tigerton* when compared to *Barcelona*: *Tigerton* has a much lower memory bandwidth and the front-side bus is a source of contention. The MPI benchmarks use the same domain decomposition as the UPC implementations, but call into tuned implementations of collective and scatter-gather communication. This explains the lower benefits of oversubscription for the MPI implementations.

A. Interaction With Per-Core Scheduling

Figures 3,4,7, and 8 also illustrate the impact of scheduling decisions inside the operating system. The new Linux distributions allow two different behaviors for the `sched_yield` system call. The default behavior (referred to as CFS) does not suspend the calling thread if its quantum has not expired, while the Posix conforming implementation un-schedules the caller. The former is designed to improve interactive behavior in desktop and server workloads, while the latter is the behavior assumed by the implementations of synchronization operations

in runtimes for scientific programming. In the figures, the bars labeled PSX show the additional performance improvements when replacing[1] the default Linux `sched_yield` implementation with the Posix conforming implementation. The results show that Posix yield performs better than CFS yield and usually the performance differences increase with the degree of oversubscription. The impact on a benchmark is relatively independent on the problem class and it is an indication of the frequency of the synchronization performed by the benchmark.

The UPC workload is less affected by the `sched_yield` implementation than the OpenMP workload. This behavior is explained by the finer granularity of parallelism in the OpenMP implementations. The performance of OpenMP runs with eight threads per core is completely dominated by the impact of more cooperative yielding. The impact on the MPI workload is small.

For all implementations, runs where threads are explicitly managed are improved less than runs subject to the default Linux load balancing. The detailed comparison of CFS-Load with CFS-PIN is omitted for brevity. For example, the average improvements in the UPC workload performance are 6% for pinned, 9% for load balancing, with ranges [-7% , 35%] and [-10% , 40%] respectively. As explained in the next section, this behavior is caused by the inability of the Linux load balancing to migrate threads after startup and initial memory allocation. Based on these observations all other experimental results presented are with Posix yield.

B. Thread Affinity Management

Figures 3,4,7, and 8 also present the impact of thread affinity management on application performance. The bars labeled PIN show the average performance improvements when threads are evenly distributed across the available cores when compared to the default Linux load balancing. As expected, the impact of affinity management is higher for the NUMA architectures, as illustrated for UPC by Figures 3 and 4. UPC and OpenMP are sensitive to thread affinity management regardless of the degree of oversubscription. For example, on *Barcelona* UPC runtime performance improves by as much as 57%, while OpenMP performance improves by 31%. MPI performance is less affected by affinity management in the presence of oversubscription. Explicit thread affinity management also increases performance reproducibility: runs with the default Linux load balancing exhibit a variation as high as 120%, while runs with pinned threads vary by at most 10%.

The performance differences are explained by a combination of load balancing behavior, NUMA memory affinity and runtime implementation. With the default load balancing, threads are started on few cores and later migrated. With first-touch memory affinity management, pages are bound to a memory controller before threads have a chance to migrate to an available core. OpenMP performs an implicit barrier after spawning threads; threads might sleep inside the barrier which determines the Linux load balancing to migrate threads. UPC, which is the most sensitive to affinity management, allocates the shared heap at program startup and each thread touches its memory pages. However, this first touch happens before threads have a chance to migrate to an idle core.

For the benchmark implementations examined in this paper, the performance impact of thread affinity management is an artifact of the characteristics of Linux thread startup and load balancing, rather than of the application itself. Ensuring that threads are directly started on the idle cores eliminates most of the effects of explicit affinity management. Our experiments with random thread to core mappings show indistinguishable performance between different pinnings. The differences between the average workload performance with any two pinnings are within few percent (well within the variation for a given pinning), with a maximum for a particular (SP) benchmark of 12%.

In [12] we present a user level load balancer that enforces an even initial thread distribution and constrains threads to a NUMA domain, rather than a particular core. Results for the same workload presented here indicate that threads can freely migrate inside a NUMA domain without experiencing performance degradation. Also note that new implementations of job spawners on large scale systems enforce an even thread distribution at program startup. We therefore expect explicit thread affinity management to play a smaller role in code optimization of scientific applications.

VI. Competitive Environments

We explore two alternatives for system management in competitive environments: 1) sharing (best effort) and 2) partitioning (managed). For the sharing experiments, each application is allowed to run on all the cores, while for the partitioned experiments each application is given an equal number of distinct cores. We consider only fully isolated partitions, i.e. applications do not share sockets. The application combinations we present (EP, CG, FT and MG) have been chosen to contain co-runners with relatively equal durations[2]and different behavior. For lack of space we do not present a full set of experiments and concentrate mostly on UPC and OpenMP behavior. EP is a CPU intensive application while CG, MG and FT are memory intensive. FT performance improves (UPC) or it is not affected by oversubscription (OpenMP), while MG performance slightly degrades with oversubscription. EP performance is not affected by oversubscription, while CG performs fine grained synchronization and its performance degrades. We do not present data for IS, which runs for at most $2s$ and for SP which requires a square number of threads in the UPC implementation.

Figure 11 presents the comparative performance of combined workloads when sharing or partitioning the system. We plot the speedup of each application in a pair (*x-axis*) when sharing the system compared to the its performance with partitioning. In the partitioning experiment, each application is run on eight cores, with one thread per core. In the sharing experiments, each application receives 16 cores and the number of threads per core indicated on the *x-axis*. The

[1] Behavior is controlled by writing 1 in /proc/sys/kernel/sched_compat_yield.

[2] On *Barcelona*: ep-C=21.11s, cg-B=24.426s, ft-B=9.5s and mg-C=27.64s for OpenMP.

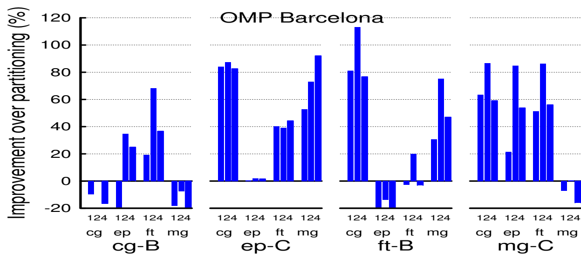


Fig. 11. *Performance for OMP benchmarks when sharing the system compared to partitioning.*

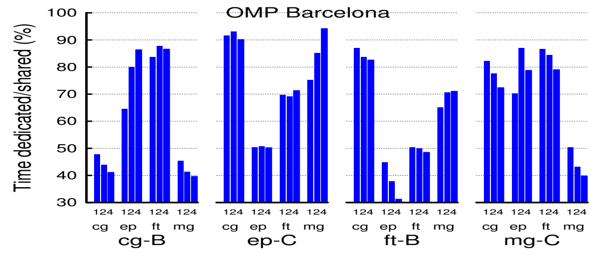


Fig. 12. *Percentage of performance sharing compared with dedicated one per core. The benchmarks are sharing the whole system.*

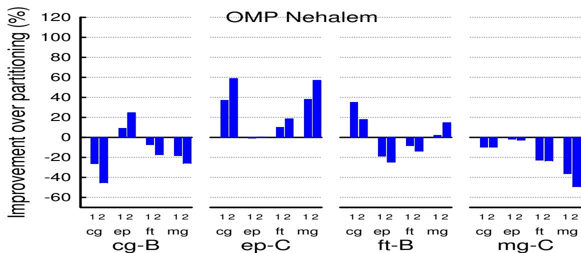


Fig. 13. *Performance for OMP benchmarks when sharing the system compared to partitioning.*

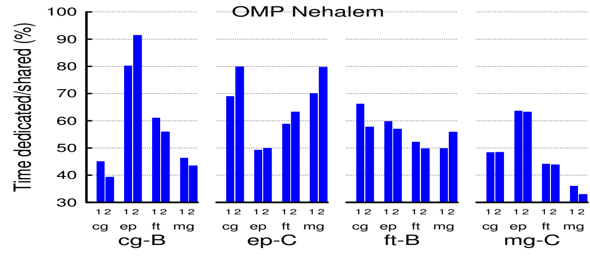


Fig. 14. *Percentage of performance sharing compared with dedicated one per core. The benchmarks are sharing the whole system.*

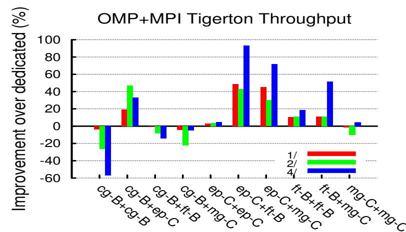


Fig. 15. *Throughput for OMP/MPI sharing.*

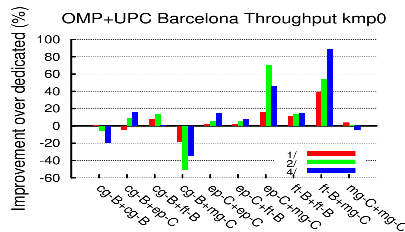


Fig. 16. *Throughput for OMP/UPC sharing. KMP_BLOCKTIME=0.*

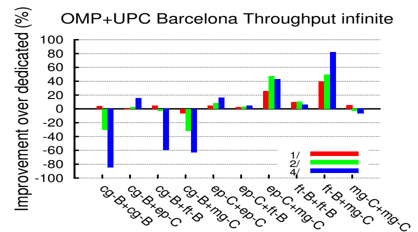


Fig. 17. *Throughput for OMP/UPC sharing. KMP_BLOCKTIME=inf.*

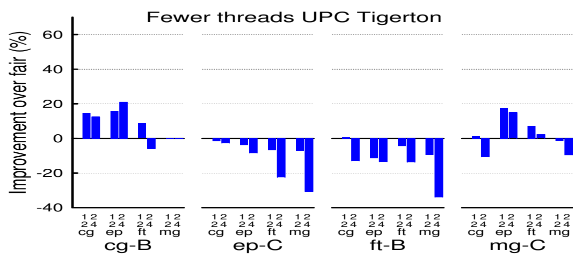


Fig. 18. *Relative differences in performance when the application is given fewer threads. x-axis presents the number of application and co-runner threads. (***1/2 and 2/4***).*

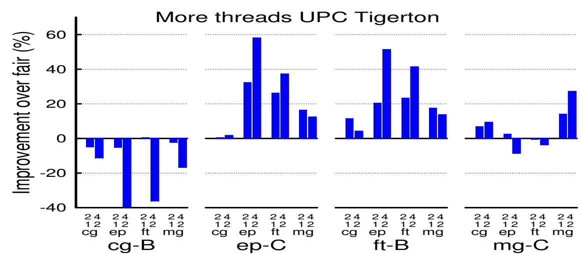


Fig. 19. *Relative differences in performance when the application is given more threads. x-axis presents the number of application and co-runner threads. (***2/1 and 4/2***).*

results indicate that partitioning cores between applications decreases performance and system throughput. For example, a *cg-B*+*cg-B* OpenMP experiment on *Barcelona* observes a 20% slowdown, while in *mg-C*+*cg-B* we observe $\approx 70\%$ improvement for *mg-C* and a 20% slowdown for *cg-B*. A *cg-B*+*cg-B* UPC experiment on *Barcelona* observes a 10% slowdown when sharing cores, while in the *mg-C*+*cg-B* experiment *mg-C* observes a 90% speedup and *cg-B* observes a 30% speedup. Overall, when each application is executed with one thread per core, sharing the system shows a 33% and 23% improvement for the UPC and OpenMP workloads respectively.

Oversubscription increases the benefits of sharing the system for the CMP architectures: the UPC improvements are 38% and 45%, while the OpenMP improvements are 46% and 28% when running with two and four threads per core respectively. The best overall throughout is obtained when sharing the system and allowing each application to run with multiple (2,4) threads per core. The results on the *Nehalem* SMT architecture are presented in Figure 13. *On this SMT architecture, sharing the system provides throughput identical to the partitioned runs.*

Figure 12 provides details about the impact of sharing the *Barcelona* system. For each benchmark, the figure plots the fraction of its performance when compared to the dedicated run with one thread per core. When sharing we expect each benchmark to run close to 50% of its original speed. Any fraction larger than 50% indicates a potential for increased throughput. These results also give an indication of application symbiosis. For example, in the *ep-C*+*ep-C* combination each instance runs at 50% of the original speed. The *cg-B*+*ep-C* contains a memory intensive benchmark and a CPU intensive benchmark: *cg-B* runs at 60% of its original speed, while *ep-C* runs at 90% of its original speed. For combinations of memory intensive (*mg-C*+*cg-B*) applications, *mg-C* runs at 85% of its original speed, while cg-B runs at 45%. Figure 14 shows the behavior on *Nehalem* where we observe a 7% overall throughput improvement.

The UPC programs respond better to oversubscription than OpenMP and therefore respond better to sharing the system. The overall throughput of the UPC+UPC workload is improved by 20% and 27% when applications are executed with one and two threads per core respectively. The OpenMP+OpenMP workload throughput is improved by 9% and 24% respectively. These improvements are relative to the performance observed when applications are run on a dedicated system with one thread per core. For reference, Figure 15 shows the throughput improvements when the system is shared between MPI and OpenMP implementations. We plot the speedup of an application combination when sharing compared to executing each application in sequence: $max(T(A), T(B))$ at the given concurrency compared with $T(A) + T(B)$ when executed with one thread per core. Any combination of programming models exhibits similar trends on all architectures. Figures 16 and 17 show the behavior of a UPC+OpenMP workload for different settings of KMP_BLOCKTIME. All results indicate that best throughput is obtained when sharing with each application running with multiple threads per core.

Figures 11 and 12 present results for experiments (PIN) where thread affinity is explicitly managed. With the default Linux load balancing, results are very noisy and throughput is lowered when sharing cores. In particular, *when using the default Linux load balancing the total execution time in a shared environment is greater than the time of executing the benchmarks in sequence*. Runs with explicitly pinned threads not only outperform the runs with the default Linux load balancing, but also exhibit very low performance variability, less than 10%, compared to the high variability of the later, which is up to 100%. For any given benchmark combination, sharing or partitioning, the performance variability of the total duration of the run $(A\|B)$ is small, usually less than 10%. Partitioning provides performance reproducibility: for every benchmark pair, the performance variation of any benchmark in the pair is less than 10%, irrespective of co-runners. When sharing the system, the variability of each benchmark across individual runs is higher, up to 81% across all combinations.

There is a direct correlation between the oversubscription trends presented in Section V for a dedicated environment and application symbiosis when sharing the system. The applications with coarse grained synchronization share the system very well, while the applications with fine grained synchronization might observe performance degradation. For any given pair of benchmarks, if one benchmark in the pair is not affected by oversubscription, the overall throughput improves irrespective of its co-runner behavior and the number of threads per core in each application: there is a direct correlation between behavior with oversubscription and behavior when sharing. This also indicates that synchronization granularity is perhaps the most important symbiosis metric and it suggests that oversubscription could potentially diminish any advantages of symbiotic scheduling of parallel applications. Increasing the number of threads per core in each application improves overall throughput. Intuitively, oversubscription increases diversity in the system and decreases the potential for resource conflicts.

Oversubscription also changes the relative ordering of the performance of implementations. In a dedicated environment, the NAS OpenMP implementations have a performance advantage over the UPC and MPI implementations ($\approx 10\% - 30\%$). Sharing reverses the relative performance trends observed in dedicated environments, and the shared UPC workloads provide the shortest time to solution ($\approx 10\%$ faster).

A. Imbalanced Sharing

The per-core scheduling mechanism (*Completely Fair Scheduler*) in Linux attempts to provide a fair repartition of execution time to the tasks sharing the core and oversubscription might provide a mechanism to proportionally allocate system resources to applications.

Figures 18 and 19 present results for sharing the system when one application is given preference and it is allowed to run with a larger number of threads. Figure 18 presents the impact on the application that receives the smaller number of threads, while Figure 19 presents the impact on the application with the larger number of threads. Both figures present the performance normalized to the performance observed when both

applications run concurrently with one thread per core. CG performance degrades with oversubscription in dedicated environments and, for these experiments, allocating more threads to CG than to co-runners does not improve its performance. EP is compute bound and allocating more threads determines a throughput increase proportional to the thread ratio with respect to co-runners. FT which benefits from oversubscription observes good throughput increases when given preference; MG also observes throughput increases, albeit smaller.

The results show a strong correlation between the application behavior with oversubscription in dedicated environments and the observed results in these scenarios. Overall, the performance of the applications that receive the smaller number of threads does not degrade and for the considered benchmarks we observe little (1% and -7% when receiving 33% and 20% per core share respectively) throughput changes when compared to balanced sharing. The performance of applications that receive a larger number of threads improves and we observe an overall improvement in throughput. The improvement in throughput is correlated to the task share received by the application, e.g. we observe 10% overall throughput improvement for two threads and 8% for four threads. These results are heavily skewed by the behavior of CG. Without CG, the improvements are 12% and 20% respectively. Our experiments indicate that imbalanced sharing should not be considered for OpenMP. We plan to examine in future work the impact of gang scheduling on the OpenMP performance in this scenario.

These trends indicate that, besides priorities and partitioning, controlling the number of threads an application receives is worth exploring as a way of controlling its system share. The magnitude of the performance differences indicates that for modest sharing (two, three applications) and modest oversubscription (two, four eight threads) gang scheduling techniques might not provide large additional performance improvements for the SPMD programming models (UPC and MPI) evaluated.

VII. Discussion

All implementations examined in this study have even per-task domain decompositions and are well balanced. We expect the benefits of oversubscription to be even more pronounced for load imbalanced irregular applications.

There are several implications from our evaluation of sharing and partitioning. Partitioning gives each application a share of dedicated resources at the expense of lower parallelism. Sharing time-slices resources across applications. Performance is determined by a combination of load balancing, degree of parallelism and contention with respect to resource usage: CPU, caches and memory bandwidth. The fact that partitioning produces lower performance than sharing indicates that for our workloads parallelism and load balance concerns still trump contention concerns.

We examine reference implementations compiled with commercially available compilers. Better optimizations or applying autotuning techniques to these applications might produce code (better cache locality, increased memory bandwidth requirements) that requires contention awareness and system partitioning. However, current autotuning techniques improve short term locality and reuse; the granularity of the OS time quantum is likely to remain larger than the duration of the computation blocks affected by autotuning. We believe that better code generation techniques will not affect the trends reported in this study for the current system sizes: parallelism and load balancing concerns will continue to be the determining performance factors. Determining the core count at which contention and scheduling interactions require careful partitioning is an open research question.

While we clearly advocate running in competitive environments and sharing cores for improved throughput, the question of the proper environment for code tuning remains open. The results with system partitioning indicate a good performance isolation between competing applications and seems to be the preferred alternative.

We believe that further performance improvements can be achieved for the fine grained applications by re-examining the implementations of the collective and barrier synchronization operations. The current implementations are heavily optimized for execution with one thread per core in dedicated environments. In [17] we present kernel level extensions for cooperative scheduling on the CellBE in the presence of oversubscription. In that particular environment, oversubscription was required for good performance and we have extended Linux with a new system call `sched_yield_to`. Similar support and a re-thinking of the barrier implementations might be able to improve the performance of oversubscribed fine grained applications. As future work we also plan to examine the interaction between oversubscription and networking behavior on large scale clusters.

The reversal of performance trends between UPC and OpenMP in the presence of competition and oversubscription, indicates that these factors might be valuable when evaluating parallel programming models in desktop and shared servers competitive environments.

VIII. Related Work

Charm [18] and AMPI [19] are research projects that advocate oversubscription as an efficient way of hiding communication latency and improving application level load balancing for MPI programs running on clusters. They use thread virtualization and provide a runtime scheduler to multiplex tasks waiting for communication. Cilk and X10 provide work stealing runtimes for load balancing. All these approaches provide their implementation specific load balancing mechanism and assume execution in dedicated environments with one "meta-thread" per core. The behavior of these models in competitive environments has not been well studied. We believe that oversubscription can provide an orthogonal mechanism to increase performance robustness in competitive environments.

Oversubscription for OpenMP programs is discussed by Curtis-Maury [20] et al for SMT and CMP (simulated) processors. For the NAS benchmarks they report a much higher impact of oversubscription than the impact we observe in this study on existing architectures and mostly advocate against it. For SMT, they also indicate that symbiosis with the hardware

context co-runner is very important. Our results using a similar workload on Nehalem processors indicate that the determining performance factor is the granularity of the synchronization operations. The differences in reported trends might be caused by the fact that we evaluate systems with a larger number of cores. Liao et al [16] discuss OpenMP performance on CMP and SMT architectures running in dedicated mode and consider both `static` and `dynamic` schedules: their results indicate little variation between different schedules.

A large body of research in multiprocessor scheduling can be loosely classified as symbiotic scheduling: threads are scheduled according to resource usage patterns. Surprisingly, there is little information available about the impact of sharing or partitioning multicore processors for fully parallel workloads. Most available studies consider either symbiotic scheduling of parallel workloads on clusters (e.g. [21] computation + I/O) or symbiotic scheduling of multiprogrammed workloads on multicore systems. Snavely and Tullsen [22] introduce symbiotic co-scheduling on SMT processors for a workload heavily biased towards multiprogrammed single-threaded jobs. Their approach samples different possible schedules and assigns threads to SMT hardware contexts. Fedorova et al [23] present an OS scheduler able to improve the cache symbiosis of multiprogrammed workloads. Our results indicate that for parallel scientific workloads in competitive environments oversubscription is a robust way to increase symbiosis without any specialized support.

Boneti et al [24] present a scheduler for balancing high performance computing MPI applications on the POWER5 processor. Their implementation targets SMT processors and uses hardware support to manipulate instruction priorities within one core. Li et al [11] present an implementation of Linux load balancing for performance asymmetric multicore architectures and discuss performance under parallel and server workloads. They modify the Linux load balance to use the notions of scaled core speed and scaled load but balancing is performed based on run queue length.

Job scheduling for parallel systems has an active research community. Feitelson [25] maintains the Parallel Workload Archive that contains standardized job traces from many large scale installations. Feitelson and Rudolph [26] provide an insightful discussion about the impact of gang scheduling on the performance of fine grained applications. Their study was conducted in 1992 on the Makbilan processor. Zhang et al [27] provide a comparison of existing techniques and discuss migration based gang scheduling for large scale systems. Gang scheduling is increasingly mentioned in multicore research studies, e.g. [6], but without a practical, working implementation. Gang scheduling has been also shown to reduce OS jitter on large scale systems. The new operating systems for the IBM BG/Q and Cray XT6 systems have been announced to support dedicated OS service cores in an attempt to minimize jitter at very large scale. Our results provide encouraging evidence that oversubscription might provide an alternate way for reducing the impact of OS jitter on the performance of scientific applications and alleviating some of the need for gang scheduling.

IX. Conclusion

In this paper we evaluate the impact of executing MPI, UPC and OpenMP applications with task oversubscription. We use implementations of the NAS Parallel Benchmarks on multi-socket multicore systems using both UMA and NUMA memory systems. In addition to the default Linux load balancing we evaluate the impact of explicit thread affinity management. We also evaluate the impact of sharing or partitioning the cores within a system on the throughput of parallel workloads.

Our results indicate that oversubscription with proper support should be given real consideration when running parallel applications. For competitive environments, oversubscription decreases the impact of co-runners and the performance variability. In these environments, oversubscription also improves system throughput by up to 27%. For the CMP systems evaluated, partitioning in competitive environments reduces throughput by up to 46%. On the *Nehalem* SMT architecture partitioning the cores between applications results in almost identical throughput to shared usage. On all architectures evaluated, best throughput is obtained when applications share all the cores and are executed with multiple threads per core. The granularity of the synchronization present in an application in a dedicated environment is perhaps the best measure of its degree of symbiosis with other applications.

References

[1] G. Krawezik, “Performance Comparison of MPI And Three OpenMP Programming Styles On Shared Memory Multiprocessors,” in *SPAA ’03: Proceedings Of The Fifteenth Annual ACM Symposium On Parallel Algorithms And Architectures*, 2003.

[2] F. Cappello and D. Etiemble, “MPI Versus MPI+OpenMP on IBM SP for the NAS Benchmarks,” in *Supercomputing ’00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, 2000.

[3] P. Balaji, R. Thakur, E. Lusk, and J. Dinan, “Hybrid Parallel Programming with MPI and PGAS (UPC),” Available at http://meetings.mpi-forum.org/secretary/2009/07/slides/2009-07-27-mpi-upc.pdf, 2009.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The Landscape of Parallel Computing Research: A View from Berkeley,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[5] “The NAS Parallel Benchmarks,” Available at http://www.nas.nasa.gov/Software/NPB.

[6] R. Liu, K. Klues, S. Bird, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz, “Tessellation: Space-Time Partitioning in a Manycore Client OS,” *Proceedings of the First Usenix Workshop on Hot Topics in Parallelism*, 2009.

[7] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith, “Multicore Resource Management,” *IEEE Micro*, vol. 28, no. 3, 2008.

[8] L. Xue, O. Ozturk, F. Li, M. Kandemir, and I. Kolcu, “Dynamic Partitioning Of Processing And Memory Resources In Embedded Mpsoc Architectures,” in *DATE ’06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 690–695.

[9] T. Li, D. Baumberger, and S. Hahn, “Efficient And Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin,” in *PPoPP ’09: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. New York, NY, USA: ACM, 2009, pp. 65–74.

[10] A. Mandal, A. Porterfield, R. J. Fowler, and M. Y. Lim, “Performance Consistency on Multi-Socket AMD Opteron Systems.” RENCI, Tech. Rep. TR-08-07, 2008. [Online]. Available: http://www.renci.org/publications/techreports/TR-08-07.pdf

[11] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn, “Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architectures,” in *SC ’07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, 2007.

[12] S. Hofmeyr, C. Iancu, and F. Blagojevic, “Load Balancing on Speed,” *To appear in Proceedings of Principles and Practice of Parallel Programming (PPoPP’10)*, 2010.

[13] S. B. Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang, “Corey: An Operating System for Many Cores,” in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’08)*, 2008.

[14] “UPC Language Specification, Version 1.0,” Available at http://upc.gwu.edu.

[15] “The GWU NAS Benchmarks,” Available at http://upc.gwu.edu/ download.html.

[16] C. Liao, Z. Liu, L. Huang, and B. Chapman, *Evaluating OpenMP on Chip MultiThreading Platforms*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008.

[17] F. Blagojevic, C. Iancu, K. Yelick, M. Curtis-Maury, D. S. Nikolopoulos, and B. Rose, “Scheduling Dynamic Parallelism On Accelerators,” in *CF ’09: Proceedings of the 6th ACM conference on Computing Frontiers*, 2009.

[18] “CHARM++ project web page,” Available at http://charm.cs.uiuc.edu.

[19] C. Huang, O. Lawlor, and L. V. Kal, “Adaptive MPI,” in *In Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03*, 2003, pp. 306–322.

[20] M. Curtis-Maury, X. Ding, C. Antonopoulos, and D. Nikolopoulos, “An Evaluation of OpenMP on Current and Emerging Multithreaded/Multicore Processors,” in *Proceedings of the 1st International Workshop on OpenMP (IWOMP’05)*.

[21] J. Weinberg and A. Snavely, “User-Guided Symbiotic Space-Sharing Of Real Workloads,” in *ICS ’06: Proceedings of the 20th Annual International Conference on Supercomputing*, 2006.

[22] A. Snavely, “Symbiotic Jobscheduling For A Simultaneous Multithreading Processor,” in *In Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000, pp. 234–244.

[23] A. Fedorova, M. I. Seltzer, and M. D. Smith, “Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler,” in *Proceedings of Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[24] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero, “A Dynamic Scheduler for Balancing HPC Applications,” in *SC ’08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, 2008.

[25] “Parallel Workload Archive,” Available at http:// www.cs.huji.ac.il/labs/parallel/workload/.

[26] D. G. Feitelson and L. Rudolph, “Gang Scheduling Performance Benefits for Fine-Grain Synchronization,” *Journal of Parallel and Distributed Computing*, vol. 16, pp. 306–318, 1992.

[27] Y. Zhang, H. Franke, J. Moreira, and A. Sivasubramaniam, “An Integrated Approach to Parallel Scheduling Using Gang-Scheduling, Backfilling and Migration,” in *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 2003.