Load Balancing on Speed

Steven Hofmeyr, Costin Iancu, Filip Blagojević

Lawrence Berkeley National Laboratory
{shofmeyr, cciancu, fblagojevic}@lbl.gov

Abstract

To fully exploit multicore processors, applications are expected to provide a large degree of thread-level parallelism. While adequate for low core counts and their typical workloads, the current load balancing support in operating systems may not be able to achieve efficient hardware utilization for parallel workloads. Balancing run queue length globally ignores the needs of parallel applications where threads are required to make equal progress. In this paper we present a load balancing technique designed specifically for parallel applications running on multicore systems. Instead of balancing run queue length, our algorithm balances the time a thread has executed on "faster" and "slower" cores. We provide a user level implementation of speed balancing on UMA and NUMA multisocket architectures running Linux and discuss behavior across a variety of workloads, usage scenarios and programming models. Our results indicate that speed balancing when compared to the native Linux load balancing improves performance and provides good performance isolation in all cases considered. Speed balancing is also able to provide comparable or better performance than DWRR, a fair multi-processor scheduling implementation inside the Linux kernel. Furthermore, parallel application performance is often determined by the implementation of synchronization operations and speed balancing alleviates the need for tuning the implementations of such primitives.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: [Design studies, Modeling techniques]; D.2.4 [*Software Engineering*]: Metrics—Performance measures; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3 [*Programming Languages*]: [Parallel, Compilers]; I.6.4 [*Computing Methodologies*]: Simulation and Modeling—Model Validation and Analysis

General Terms Performance, Measurement, Languages, Design

Keywords Parallel Programming, Operating System, Load Balancing, Multicore, Multisocket

1. Introduction

Multi-core processors are prevalent nowadays in systems ranging from embedded devices to large-scale high performance computing systems. Over the next decade the degree of on-chip parallelism will significantly increase and processors will contain tens and even hundreds of cores [6]. The availability of cheap thread level parallelism has spurred a search for novel applications that would add commercial value to computing systems, e.g. speech and image recognition or parallel browsers. These new applications are parallel, often on multiple levels, and are likely [2] to use methods and algorithms encountered in scientific computing. Their emergence will cause the structure of future desktop and even embedded workloads to be fundamentally altered.

Current operating systems are written and optimized mainly for multiprogrammed commercial and end-user workloads and thread independence is a core assumption in the design of their schedulers. In a parallel application there is a higher level of inter-thread interaction; consequently, balancing applications with data and synchronization dependences requires additional scheduler mechanisms [5, 8, 9, 16, 30]. Runtimes that are specifically designed for parallel computing provide paradigm ad hoc work stealing solutions for load balancing (for example, Adaptive MPI [11] and Cilk [14]).

In this paper we propose a generic user level technique to load balance parallel scientific applications written in SPMD style. This technique is designed to perform well on multicore processors when any of the following conditions is met: 1) the number of tasks in an application might not be evenly divisible by the number of available cores; 2) asymmetric systems where cores might run at different speeds and 3) non-dedicated environments where a parallel application's tasks are competing for cores with other tasks.

Our scheduler explicitly manages all the threads within one application and uses migration to ensure that each thread is given a fair chance to run at the fastest speed available system wide. We named our scheduling algorithm *speed balancing* and we present a user level implementation on NUMA (AMD Barcelona) and UMA (Intel Tigerton) multicore systems running the Linux 2.6.28 kernel. We empirically validate its performance under a variety of usage scenarios using combinations of parallel (OpenMP, MPI and UPC) and multiprogrammed workloads. Using speed balancing the performance sometimes doubles when compared to the Linux load balancer, while execution variability decreases from a maximum of 100% to usually less than 5%. Speed balancing is also able to provide better performance than OS-level load balancing mechanisms [15] that are designed to provide global fairness on multicore processors. Furthermore, our results indicate that speed balancing eliminates some of the implementation restrictions for synchronization or collective operations such as barriers when running in non-dedicated or oversubscribed environments.

Research in process scheduling has a long and distinguished history and the importance of implementing effective operating system or paradigm independent load balancing mechanisms for mixed parallel workloads has been recently [5, 16] re-emphasized. To the best of our knowledge, the proposed approach of dynamically balancing the speed of running threads is original. Measuring speed is an elegant and simple way of capturing history and the complex interactions caused by priorities, interactive tasks or

PPoPP'10, January 9–14, 2010, Bangalore, India.

heterogeneous cores. The notion of speed allows us to design a novel user level parallel-application balancer that is architecture independent, works in non-dedicated systems, does not make assumptions about application behavior and performs well with uneven task distributions. As discussed throughout this paper, alternative approaches have limitations in at least one of these areas. Speed balancing can easily co-exist with the default Linux load balance implementation and our Linux-centric discussion is easy to relate to other operating systems. Furthermore, speed balancing has the potential to simplify the design of managed runtimes that have "domain" specific load balancers and opens the door for simpler parallel execution models that rely on oversubscription as a natural way to achieve good utilization and application-level load balancing.

2. Load Balancing

The importance of load balance in parallel systems and applications is widely recognized. Contemporary multiprocessor operating systems, such as Windows Server, Solaris 10, Linux 2.6 and FreeBSD 7.2, use a two-level scheduling approach to enable efficient resource sharing. The first level uses a distributed run queue model with per core queues and fair scheduling policies to manage each core. The second level is load balancing that redistributes tasks across cores. The first level is scheduling in time, the second scheduling in space. The implementations in use share a similar design philosophy: 1) threads are assumed to be independent; 2) load is equated to queue length and 3) locality is important.

The current design of load balancing mechanisms incorporates assumptions about the workload behavior. Interactive workloads are characterized by independent tasks that are quiescent for long periods (relative to cpu-intensive applications) of time. Server workloads contain a large number of threads that are mostly independent and use synchronization for mutual exclusion on small shared data items and not for enforcing data or control dependence. To accommodate these workloads, the load balancing implementations in use do not start threads on new cores based on global system[1] information. Another implicit assumption is that applications are either single threaded or, when multi-threaded, they run in a dedicated environment. The common characteristics of existing load balancing designs can be summarized as follows: 1) they are designed to perform best in the cases where cores are frequently idle; and 2) balancing uses a coarse-grained global optimality criterion (equal queue length using integer arithmetic). These heuristics work relatively well for current commercial and end-user multiprogrammed workloads but are likely to fail for parallelism biased workloads or on asymmetric systems. Development of scheduler support for recognizing proper parallel application characteristics is an essential step towards achieving efficient utilization of highly parallel systems.

In this paper we focus on Linux, which embodies principles mirrored in other OSes such as Windows, Solaris and FreeBSD. In the Linux kernel 2.6 series there are per-core independent run queues containing the currently running tasks[2]. Since version 2.6.23, each queue is managed by the Completely Fair Scheduler (CFS). The *Load* on each core is defined as the number of tasks in the per-core run queue. Linux attempts to balance the load (queue lengths) system wide by periodically invoking the load balancing algorithm on every core to pull tasks from longer to shorter queues, if possible.

[1] For example, at task start-up Linux tries to assign it an idle core, but the idleness information is not updated when multiple tasks start simultaneously.

[2] Linux does not differentiate between threads and processes: these are all tasks.

Linux contains topology awareness, captured by *scheduling domains*. The scheduling domains form a hierarchy that reflects the way hardware resources are shared: SMT hardware context, cache, socket and NUMA domain. Balancing is done progressing up the hierarchy and at each level, the load balancer determines how many tasks need to be moved between two groups to balance the sum of the loads in those groups. If the balance cannot be improved (e.g. one group has 3 tasks and the other 2 tasks) Linux will not migrate any tasks. The frequency with which the load balancer is invoked is dependent on both the scheduling domain and the instantaneous load. It is configured via a set of kernel parameters that can be changed at runtime through the */proc* file system. The default is to perform load balancing on idle cores every 1 to 2 timer ticks (typically 10ms on a server) on UMA and every 64ms on NUMA; on busy cores, every 64 to 128ms for SMT, 64 to 256ms for shared packages, and every 256 to 1024ms for NUMA. Another tunable kernel parameter is the imbalance percentage, a measure of how imbalanced groups must be in order to trigger migrations. This is typically 125% for most scheduling domains, with SMT usually being lower at 110%. With the default parameter settings, the frequency of balancing and the number of migrations decreases as the level in the scheduling domain hierarchy increases.

The load balancer may fail to balance run queues because of constraints on migrating tasks. In particular, the balancer will never migrate the currently running task, and it will resist migrating "cache hot" tasks, where a task is designated as cache-hot if it has executed recently ($\approx 5ms$) on the core (except for migration between hardware contexts on SMT). This is a simple locality heuristic that ignores actual memory usage. If repeated attempts (typically between one and two) to balance tasks across domains fail, the load balancer will migrate cache-hot tasks. If even migrating cache-hot tasks fails to balance groups, the balancer will wake up the kernel-level *migration thread*, which walks the domains from the base of the busiest core up to the highest level, searching for an idle core to push tasks to.

The FreeBSD ULE scheduler [24], available as default in FreeBSD 7.2, uses per core scheduling queues and an event driven approach to managing these queues. FreeBSD uses a combination of pull and push task migration mechanisms for load balancing. Of particular interest to the performance of parallel applications is the push migration mechanism that runs twice a second and moves threads from the highest loaded queue to the lightest loaded queue. In the default configuration the ULE scheduler will not migrate threads when a static balance is not attainable, but theoretically it is possible to change the configuration to allow threads to migrate even when queues are imbalanced by only one thread. According to the ULE description [24], the balancer will migrate tasks even when run queues are short, i.e. three tasks running on two CPUs. In our experiments, we have explored all variations of the `kern.sched` settings, without being able to observe the benefits of this mechanism for parallel application performance.

The recently proposed [15] Distributed Weighted Round-Robin (DWRR) multi-processor fair scheduling provides system-wide fair CPU allocation from the Linux kernel. The implementation schedules based on *rounds*, defined as the shortest time period during which every thread in the system completes at least one of its *round slices* (weighted by priority). DWRR maintains two queues per core, *active* and *expired* and a thread is moved to the expired queue after finishing its round slice. To achieve global fairness, each CPU has a round number and DWRR ensures that during execution this number for each CPU differs by at most one system-wide. When a CPU finishes a round it will perform *round balancing* by stealing threads from the active/expired queues of other CPUs, depending on their round number. When all the threads have run, the round number is advanced. DWRR uses 100 ms for the round slice in the

2.6.22 Linux kernel and 30 ms in the 2.6.24 kernel. DWRR does not maintain a migration history and both the number of threads migrated from each core within one round and the round slice are configurable; it appears that in order to enforce fairness the algorithm might migrate a large number of threads. Being a kernel level mechanism the proposed implementation is not application aware and uniformly balances all the tasks in the system. To our knowledge, DWRR has not been tuned for NUMA. In contrast, our approach maintains a migration history and it is designed to limit the rate of migrations by stealing only one task at one time.

3. Parallel Applications and Load Balancing

The vast majority of existing implementations of parallel scientific applications use the SPMD programming model: there are phases of computation followed by barrier synchronization. The three paradigms explored in this paper, OpenMP, UPC and MPI, all provide SPMD parallelism. The SPMD model contravenes the assumptions made in the design of system level load balancing mechanisms : threads are logically related, have inter-thread data and control dependence and have equally long life-spans.

In SPMD programming, users or libraries often make static assumptions about the number of cores available and assume dedicated environments. Applications are run with static parallelism or plainly have restrictions on the degree of task parallelism due to the difficulty of parallel domain decomposition, for example, many parallel jobs often request an even or perfect square number of processors. Furthermore, asymmetries or uneven thread distributions across cores are likely to be encountered in future systems. The Intel Nehalem processor provides the Turbo Boost mechanism that over-clocks cores until temperature rises and as a result cores might run at different clock speeds. Recently proposed OS designs such as Corey [32] or those under development at Cray and IBM provide for reserving cores to run only OS level services.

Achieving good performance for SPMD applications requires that : 1) all tasks within the application make equal progress and 2) the maximum level of hardware parallelism is exploited. To illustrate the former, consider a two CPU system and an application running with three threads. The default Linux load balancing algorithm will statically assign two threads to one of the cores and the application will perceive the system as running at 50% speed. The impact on performance in this case is illustrated in Section 6.2. The implementations of both the ULE scheduler or fair multiprocessor DWRR scheduling [15] might correct this behavior by repeatedly migrating one thread. In this case, the application perceives the system as running at 66% speed. Our approach also addresses this scenario by explicitly detecting and migrating threads across run queues, even when the imbalance is caused by only one task.

The interaction between an application or programming model and the underlying OS load balancing is largely accomplished through the implementation of synchronization operations: locks, barriers or collectives (e.g. reduction or broadcast). Implementations use either polling or a combination of polling with yielding the processor (`sched_yield`) or sleeping[3] (`sleep`). In dedicated environments where applications run with one task per core, polling implementations provide orders of magnitude performance improvements [21]. In non-dedicated, oversubscribed or cluster environments some form of yielding the processor is required for overall progress. The Intel OpenMP runtime we evaluate calls `sleep`, while UPC and MPI call `sched_yield`. A thread that yields remains on the active run queue and hence the OS level load balancer counts it towards the queue length (or load). By contrast, a thread that sleeps is removed from the active run queue, which enables the OS level load balancer to pull tasks onto the CPUs where threads are sleeping.

Based on our Linux and FreeBSD experiences described in Section 6.2, applications calling `sleep` benefit from better level system load balancing. On the other hand, when load is evenly distributed, implementations that call `sched_yield` can provide better performance due to faster synchronization. One of the implications of our work is that with speed balancing, identical levels of performance can be achieved by calling only `sched_yield`, irrespective of the instantaneous system load.

[3] For brevity any other mechanism that removes threads from the run queue is classfied as sleep.

4. Argument For Speed Balancing

When threads have to synchronize their execution in SPMD applications, the parallel performance is that of the slowest thread and variation in "execution speed" of any thread negatively affects the overall system utilization and performance. A particular thread will run slower than others due to running on the core with the longest queue length, sharing a core with other threads with higher priority or running on a core with lower computational power (slower clock speed). In the rest of this paper and the description of our algorithm we classify cores as slow or fast depending on the "progress" perceived by an application's threads. In the previous example where an application with three threads runs on two cores, the core running two threads will be classified as slow.

Consider N threads in a parallel application running on M homogeneous cores, $N > M$. Let T be the number of threads per core $T = \lfloor \frac{N}{M} \rfloor$. Let FQ denote the number of *fast* cores, each running T threads and SQ the number of *slow* cores with $T + 1$ threads. Assume that threads will execute for the same amount of time S seconds and balancing executes every B seconds. Intuitively, S captures the duration between two program barrier or synchronization points. With Linux load balancing, the total program running time under these assumptions is at most $(T + 1) * S$, the execution time on the slow cores.

We assume that migration cost is negligible, therefore the statements about performance improvements and average speed provide upper bounds. Assuming a small impact of thread migration is reasonable: the cost of manipulating kernel data structures is small compared to a time quantum, cache content is likely lost across context switches when threads share cores and our algorithm does not perform migration across NUMA nodes. Li et al [15] use microbenchmarks to quantify the impact of cache locality loss when migrating tasks and indicate overheads ranging from μseconds (in cache footprint) to 2 milliseconds (larger than cache footprint) on contemporary UMA Intel processors. For reference, a typical scheduling time quantum is 100ms.

Under existing circumstances, with fair per core schedulers the average thread speed is $f * \frac{1}{T} + (1 - f) * \frac{1}{T+1}$, where f represents the fraction of time the thread has spent on a fast core. The Linux queue-length based balancing will not[4] migrate threads so the overall application speed is that of the slowest thread $\frac{1}{T+1}$. Ideally, each thread should spend an equal fraction of time on the fast cores and on the slow cores[5]. The asymptotic average thread speed becomes $\frac{1}{2*T} + \frac{1}{2*(T+1)}$ which amounts to a possible speedup of $\frac{1}{2*T}$.

Instead of targeting perfect fairness we make an argument based on necessary but not sufficient requirements: *In order for speed balancing to perform better than queue-length based balancing each thread has to execute at least once on a fast core.*

[4] If threads do not sleep or block.

[5] This is the ideal for speed balancing to perform better than the default Linux load balancing. Clearly this is not the ideal for optimal balance in every case.

Deriving constraints on the number of balancing operations required to satisfy this condition indicates the threshold at which speed balancing is expected to perform better than Linux-style queue length based balancing. Below this threshold the two algorithms are likely to provide similar performance. Consequently, in the rest of this discussion the negative qualifiers (e.g. non-profitable or worst-case scenario) mean in fact the *same performance* as the Linux default.

Lemma 1. *The number of balancing steps required to satisfy the necessity constraint is bound by* $2 * \lceil \frac{SQ}{FQ} \rceil$.

Proof. We show the proof for the case $FQ < SQ$. In each step of the algorithm we pull one thread from a slow queue to a fast queue. The simple act of migrating a thread flips one whole queue from slow to fast and the other from fast to slow. This means that at each step we give $FQ * T$ threads a chance to run on a fast queue. We continue doing this for $\frac{SQ}{FQ}$ steps and at the end $N - SQ$ threads have once run fast. There are SQ threads left that had no chance at running fast. From all the queues containing these threads we need to pull a different thread onto the available fast queues and run one interval without any migrations. This process takes $\frac{SQ}{FQ}$ steps. Thus the total number of steps is $2 * \frac{SQ}{FQ}$. A similar reasoning shows that for $FQ \geq SQ$ two steps are needed. □

The proof describes an actual algorithm for speed balancing and we can now discuss its expected behavior. Queues grow by at most one thread per step and the rate of change is slow. Since threads should not be migrated unless they have had a chance to run on a queue, balancing could start after $T + 1$ time quanta: increasing load postpones the need for balancing. Lemma 1 provides us with heuristics to dynamically adjust the balancing interval if application behavior knowledge is available.

The total "program" running time is $Total\_time = (T+1) * S$ and according to Lemma 1 the prerequisite for speed balancing to be profitable is:
$Total\_time > 2 * \lceil \frac{SQ}{FQ} \rceil$ or $(T + 1) * S > 2 * \frac{NmodM}{M-NmodM}$.

This reasoning assumes an implementation where balancing is performed synchronously by all cores. Note that our implementation uses a distributed algorithm so task migrations might happen with a higher frequency, proportional to the number of cores.

Figure 1 indicates the scalability of this approach for increasing numbers of threads and cores. It plots the minimum value of S for (N,M,$B = 1$ $time\ unit$) after which speed balancing is expected to perform better than Linux load balancing. This plot captures both the relationship between the duration of inter-thread synchronization periods and frequency of balancing as well as the scalability of the algorithm with the number of cores and threads. In the majority of cases $S \leq 1$, which indicates that the algorithm can balance fine-grained applications where the duration of computation is "equal" to the duration of a synchronization operation. For a fixed number of cores, increasing the number of threads decreases the restrictions on the minimum value of S, whereas increasing the number of cores increases the minimum value for S. The high values for S appearing on the diagonals capture the worst case scenario for speed balancing: few (two) threads per core and a large number of slow cores ($M - 1$, $M - 2$). Coarse grained applications will be effectively balanced because the minimum S will always be high.

The preceding argument made for homogeneous systems can be easily extended to heterogeneous systems where cores have different performance by weighting the number of threads per core with the relative core speed.

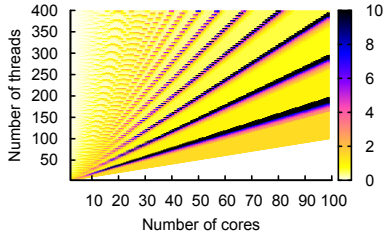


Figure 1. *Relationship between inter-thread synchronization interval (S) and a fixed balancing interval (B=1). The scale of the figure is cut off at 10; the actual data range is [0.015,147].*

5. Speed Balancing

We provide a user level balancer that manages the application threads on user requested cores. Our implementation does not require any application modifications, it is completely transparent to the user and it does not make any assumptions about application implementation details. In particular, it does not make any assumptions about thread "idleness," which is an application-specific notion, or about synchronization mechanisms (busy-wait, spin locks or mutexes). Blocking or sleep operations are captured by the algorithm since they will be reflected by increases in the speed of their co-runners.

For the purpose of this work we define $speed = \frac{t_{exec}}{t_{real}}$, where t_{exec} is the elapsed execution time and t_{real} is the wall clock time. This measure directly captures the share of CPU time received by a thread and can be easily adapted to capture behavior in asymmetric systems. It is simpler than using the inverse of queue length as a speed indicator because that requires weighting threads by priorities, which can have different effects on running time depending on the task mix and the associated scheduling classes. Using the execution time based definition of speed is a more elegant measure than run queue length in that it captures different task priorities and transient task behavior without requiring any special cases. Furthermore, the current definition provides an application and OS independent metric for our algorithm. We discuss in Section 7 other possible measures of speed based on sampling performance counters. Also, sampling performance counters at the user level it is not an entirely portable approach since it will interfere with application tuning efforts or other runtime techniques that utilize the same CPU performance registers.

5.1 Algorithm

Speed balancing uses a balancing thread running on each core (termed the *local* core). We implement a scalable distributed algorithm where each balancing thread or *balancer* operates independently and without any global synchronization. Periodically a *balancer* will wake up, check for imbalances, correct them by pulling threads from a slower core to the local core (if possible) and then sleep again. The period over which the *balancer* sleeps (*balance interval*) determines the frequency of migrations. The impact of this parameter is illustrated in Section 6.1. For all of our experiments we have used a fixed balance interval of 100 ms.

Note that in the following description the notion of a core's speed is an entirely application specific notion. When several applications run concurrently each might perceive the same core differently based on the task mix. When activated, the *balancer* carries out the following steps:

1. For every thread th_i on the local core c_j, it computes the speed s_j^i over the elapsed balance interval.
2. It computes the local core speed s_j over the balance interval as the average of the speeds of all the threads on the local core: $s_j = average(s_j^i)$.

3. It computes the global core speed s_{global} as the average speed over all cores: $s_{global} = average(s_j)$.
4. It attempts to balance if the local core speed is greater than the global core speed: $s_j > s_{global}$.

The only interaction between *balancer* threads in our algorithm is mutual exclusion on the variable s_{global}. The *balancer* attempts to balance (step 4 above) by searching for a suitable remote core c_k to pull threads from. A remote core c_k is suitable if its speed is less than the global speed ($s_k < s_{global}$) and it has not recently been involved in a migration. Since our algorithm does not perform global synchronization, this post-migration block must be at least two balance intervals, sufficient to ensure that the threads on both cores have run for a full balance interval and the core speed values are not stale. This heuristic has the side effect that it allows cache hot threads to run repeatedly on the same core. Once it finds a suitable core c_k, the *balancer* pulls a thread from the remote core c_k to the local core c_j. The *balancer* chooses to pull the thread that has migrated the least in order to avoid creating "hot-potato" tasks that migrate repeatedly.

Without global synchronization, our algorithm cannot guarantee that each migration will be the best possible one. Each *balancer* makes its own decision, independent of the other *balancers*, which can result in a migration from the slowest core to a core that is faster than average, but not actually the fastest core. To help break cycles where tasks move repeatedly between two queues and to distribute migrations across queues more uniformly, we introduce randomness in the balancing interval on each core. Specifically, a random increase in time of up to one balance interval is added to the balancing interval at each wake-up. Consequently the elapsed time since the last migration event varies randomly from one thread to the next, from one core to the next, and from one check to the next. If knowledge about application characteristics is available, the balancing interval can be further tuned according to the principles described in Section 4.

5.2 Implementation

Speed balancing is currently implemented as a stand-alone multi-threaded program, `speedbalancer`, that runs in user space. `speedbalancer` takes as input the parallel application to balance and forks a child which executes the parallel application. `speedbalancer` then inspects the */proc* file system to determine the process identifiers (PIDs) of all[6] the threads in the parallel application. Due to delays in updating the system logs, we employ a user tunable startup delay for the balancer to poll the */proc* file system. Initially, each of the threads gets pinned (using the `sched_setaffinity` system call) to a core in such a way as to distribute the threads in round-robin fashion across the available cores. While speed balancing can attain good dynamic behavior regardless of the initial distribution, on NUMA systems we prevent inter-NUMA-domain migration. The initial round-robin distribution ensures maximum exploitation of hardware parallelism independent of the system architecture.

The `sched_setaffinity` system call is also used to migrate threads when balancing. `sched_setaffinity` forces a task to be moved immediately to another core, without allowing the task to finish the run time remaining in its quantum (called the `vruntime` in Linux/CFS). Any thread migrated using `sched_setaffinity` is fixed to the new core; Linux will not attempt to move it when doing load balancing. Hence, we ensure that any threads moved by `speedbalancer` do not also get moved by the Linux load balancer. This also allows us to apply speed balancing to a particular parallel application without preventing Linux from load balancing any other unrelated tasks.

[6] This implementation can be easily extended to balance applications with dynamic parallelism by polling the */proc* file system to determine task relationships.

	Tigerton	*Barcelona*
Processor	Intel Xeon E7310	AMD Opteron 8350
Clock GHz	1.6	2
Cores	16 (4x4)	16 (4x4)
L1 data/instr	32K/32K	64K/64K
L2 cache	4M per 2 cores	512K per core
L3 cache	none	2M per socket
Memory/core	2GB	4GB
NUMA	no	socket (4 cores)

Table 1. *Test systems.*

To accurately measure thread speed, we need to obtain the elapsed system and user times for every thread being monitored. We found the most efficient approach to be through the *taskstats* netlink-based interface to the Linux kernel. Because of the way task timing is measured, there is a certain amount of noise in the measurements, and since `speedbalancer` pulls threads from slower cores, it needs a mechanism to ensure that the slow cores are accurately classified. Hence `speedbalancer` only pulls from cores that are *s*ufficiently lower than the global average, i.e. it pulls from core c_k when $s_k/s_{global} < T_S$, where T_s is the configurable speed threshold. This also ensures that noise in the measurements does not cause spurious migrations when run queues are perfectly balanced. In our experiments we used $T_s = 0.9$.

Unlike Linux, we do not implement hierarchical scheduling-domain balancing triggered at different frequencies. Instead, different scheduling domains can have different migration intervals. For example, `speedbalancer` can enable migrations to happen twice as often between cores that share a cache as compared to those that do not. The scheduling domains are determined by reading the configuration details from the */sys* file system. Migrations at any scheduling domain level can be blocked altogether. This is particularly useful on NUMA systems, where migrations between nodes can have large performance impacts. In our experiments we allowed migrations across "cache" domains and blocked NUMA migrations: estimating the cost of NUMA migration as presented by Li et al [16] can be easily integrated.

6. Results

We tested speed balancing on two different multicore architectures as shown in Table 1. 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. In the results that follow, we focus on the *Tigerton* until section 6.4, when we report on the effects of NUMA. Both systems were running the Linux 2.6.28.2 kernel (latest version as of January 2009). We report results for experiments with non-essential services disabled and essential services pinned to the 16th core to minimize OS interference. The results with non-pinned services indicate even more erratic behavior for the default Linux load balancing and have no impact on the behavior of speed balancing. We have also run the full set of experiments on a two socket Intel Nehalem system which is a 2x4x(2) NUMA SMT architecture with each core providing two hardware execution contexts. Although our results (omitted for brevity) indicate that speed balancing outperforms load balancing on the Nehalem, speed balancing does not adjust for the the asymmetric nature of the SMT architecture. In future work we intend to weight the speed of a task according to the state of the other hardware context, because a task running on a "core" where both hardware contexts are utilized will run slower than when running on a core by itself.

We use parallel and multiprogrammed workloads and examine their combined behavior. Each experiment has been repeated ten times or more. The parallel workload contains implementations of

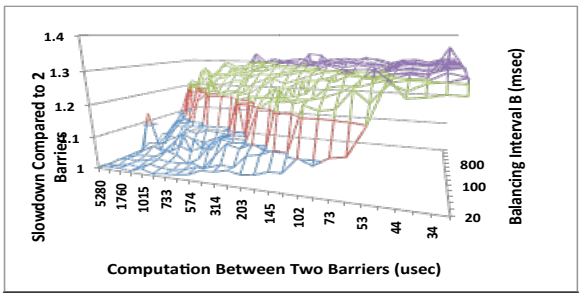


Figure 2. *Three threads on two cores on Intel Tigerton, fixed amount of computation per thread* $\approx$ $27s$*, with barriers at the interval shown on x-axis.*

the NAS [19] Parallel Benchmarks (NPB) in OpenMP, MPI and UPC. These are scientific applications written with SPMD parallelism which capture both CPU and memory intensive (regular and irregular) behaviors. We ran several configurations for each benchmark (classes S, A, B, C) with different memory footprints, resulting in a range of execution times from a few seconds to hundreds of seconds. Thus, we have a reasonable sample of short and long lived applications that perform synchronization operations at granularities ranging from few milliseconds to seconds, as illustrated in Table 2. Asanović et al [2] 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 application domain.

The NAS [19] 3.3 release OpenMP and MPI benchmarks are compiled with the Intel `icc` 11.0 compiler. The NAS [20] 2.4 release UPC benchmarks are compiled with the Berkeley UPC 2.8.0 compiler using `icc` 11.0 as a back-end compiler.

For comparison, on the Intel Tigerton UMA system we have repeated the experiments using both FreeBSD 7.2 and the DWRR implementation in the Linux kernel 2.6.22. These implementations respectively have limited or no NUMA support. DWRR supports both the Linux 2.6.22 kernel which uses the O(1) per core scheduler and the Linux 2.6.24 kernel which uses the CFS per core scheduler (same as 2.6.28). We were not able to boot the 2.6.24 DWRR based implementation and therefore we present results using the 2.6.22 based implementation. Furthermore, due to stability problems with the latter (system hangs or kernel panic) we have only partial results to compare speed balancing against DWRR.

In the rest of this discussion, LOAD refers to default Linux load balancing and SPEED refers to speed balancing (experiments run with `speedbalancer`).

6.1 Impact of Balancing and Synchronization Frequency

We illustrate the relationship between the application level granularity of synchronization S (time between barriers) and the load balancing interval B using a modified version of the NAS EP benchmark. EP is "embarrassingly parallel": it uses negligible memory, no synchronization and it is a good case test for the efficiency of load balancing mechanisms. In EP each task executes the same number of "iterations" and we have modified its inner loop to execute an increasing number of barriers. Results for a configuration with three threads running on two cores on the Intel Tigerton architecture are presented in Figure 2. Increasing the frequency of migrations (balancing events) leads to improved performance. The EP benchmark is CPU intensive and has a very small memory footprint, therefore thread migrations are cheap with a magnitude of several μs, similar to the results reported in [15]. A 20 ms balancing interval produces the best performance for EP. Loss of cache context for memory intensive benchmarks increases the cost of migrations to several ms. Our parameter sweep, using all the NAS benchmarks, for the granularity of the balancing interval indicates that a value of 100 ms works best in practice. This is also the value of the system scheduler time quanta: using a lower value for the balancing interval might produce inaccurate values for thread speeds since the OS may have stale values.

6.2 Dedicated Environment

In this set of experiments we examine the behavior of NPB running in a dedicated environment. Each benchmark is compiled to use 16 threads and we vary the number of cores allocated to the application. We compare SPEED, LOAD and scenarios where the threads are specifically pinned to cores. The effectiveness of SPEED for parallel applications in a dedicated environment is illustrated in Figure 3, which shows the speedup for a UPC implementation of the NAS EP benchmark. EP scales perfectly when compiled and run with one thread per core ("One-per-core"). The dynamic balancing enforced by SPEED achieves near-optimal performance at all core counts, with very little performance variation. By contrast, static application level balancing where threads are pinned to cores ("PINNED"), only achieves optimal speedup when 16 $mod\ N = 0$ (at 2, 4, 8 and 16 cores). LOAD[7] is often worse than static balancing and highly variable (run times can vary by a factor of three), indicating a failure to correct initial imbalances. We explore this issue further in section 6.4.

Performance with DWRR exhibits low variability (a few percent lower than SPEED) and scales as well as with SPEED up to eight cores and much better than with LOAD. On more than eight cores, DWRR performance is worse than SPEED, and at 16 threads on 16 cores, the speedup is only 12. With every other balancing system investigated, speedup at 16 on 16 was always close to 16. Unfortunately, we have only a very limited set of DWRR results due to its stability problems. Performance with the ULE FreeBSD scheduler[8] is very similar to the pinned (statically balanced) case. Although we attempted to tune FreeBSD to migrate tasks in unbalanced queues, this did not appear to happen.

The lines labeled LOAD-SLEEP and LOAD-YIELD in Figure 3 capture the different behavior caused by barrier implementations. The default UPC barrier implementation calls `sched_yield` in oversubscribed runs. For LOAD-SPEED, we have modified the UPC runtime to call `usleep(1)` and, as illustrated, the Linux load balancer is able to provide better scalability.

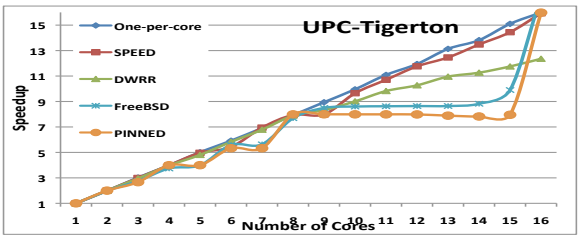
A representative sample of the NPB that covers both small and large memory footprint benchmarks, as well as coarse and fine grained synchronization is presented in Table 2. All benchmarks scale up to 16 cores on both systems and their running time is in the range $[2\ sec, 80\ sec]$. On all benchmarks, the performance with SPEED is better than both static [9] thread distribution (up to 24%) and LOAD (up to 46%) as summarized in Table 3. The improvements are computed as the average of ten runs. Performance with LOAD is erratic, varying up to 67% on average, whereas with SPEED it varies less than 5% on average.

The distribution of performance improvements for each benchmark across various core counts is illustrated in detail in Figure 4, which presents the performance benefits of using SPEED instead of LOAD for the worst (SB_WORST/LB_WORST) and average performance (SB_AVG/LB_AVG) observed over ten runs.

[7] We force Linux to balance over a subset of cores using the `taskset` command. For each core count, we chose a subset that spans the fewest scheduling domains to give Linux the most opportunity for balancing.

[8] Configured with `kern.sched.steal_thresh=1` and `kern.sched.affinity=0`.

[9] We include static balancing to give an indication of the potential cost of migrations; for non-dedicated environments explicitly pinning threads is not a viable option as shown in section 6.3.



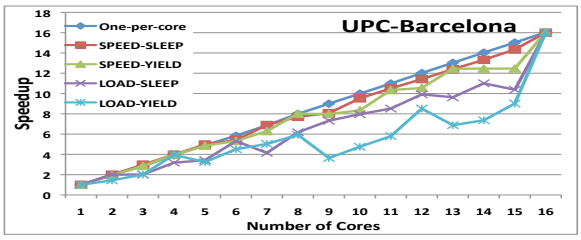


Figure 3. *UPC EP class C speedup on* Tigerton *and* Barcelona*. The benchmark is compiled with 16 threads and run on the number of cores indicated on the x-axis. We report the average speedup over 10 runs.*

Worst case performance significantly improves (up to 70%) and the average performance improves up to 50%. SPEED shows very little performance variation when compared to LOAD, as illustrated by the lines $SB\_VARIATION$ and $LB\_VARIATION$ which are plotted against the right hand $y-axis$. Overall, SPEED shows a 2% performance variance. The results for the MPI workload are similar to the results obtained for the UPC workload in all cases. For brevity, we omit MPI results from the paper.

For the OpenMP workload we examine several scenarios. The behavior of the OpenMP barrier implementation in the Intel runtime can be controlled by setting the environment variable `KMP_BLOCKTIME`. The default implementation is for each thread to wait for 200 ms before going to sleep. A setting of `infinite` forces threads to poll continuously. In Figure 4 the lines labeled DEF use the default barrier implementation, while the lines labeled INF use the polling implementation. For this workload, LOAD with the polling barrier implementation performs better than LOAD with the default barrier: the overall performance with LB_INF is 7% better than the performance with LB_DEF. The improvements for each benchmark are correlated with its synchronization granularity shown in Table 2. For example, cg.B performs barrier synchronization every 4 ms. In this case, polling barriers provide best performance.

The results presented are with the default setting of 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 with OMP_STATIC. Liao et al [17] also report better performance when running OpenMP with a `static` number of threads, evenly distributed. In our experiments, the best performance for the OpenMP workload is obtained when running in polling mode with SPEED, as illustrated by the line SB_INF/LB_INF. In this case SPEED achieves a 11% speedup across the whole workload when compared to LB_INF. Our current implementation of speed balancing does not have mechanisms to handle sleeping processes and SPEED slightly decreases the performance when tasks sleep inside synchronization operations. Comparing SB_DEF with LB_DEF shows an overall performance decrease of 3%.

Removing the need for tasks to sleep in order to benefit from the system level load balancing directly improves the performance of synchronization operations and the improvment is sometimes directly reflected in end-to-end application performance. For example, using the OpenMP `infinite` barriers we obtain an overall 45% performance improvement on *Barcelona* for the class S of the benchmarks when running on 16 cores. The behavior of class S at scale is largely determined by barriers, so this illustrates the effectiveness of polling on synchronization operations.

6.3 Competitive Environment

In this section we consider the effects of the speed balancing in multi-application environments. The limitations of static balancing

BM	Class	RSS (GB)	Speedup on 16 cores		Inter-barrier time (msec)	
			Tigerton	*Barcelona*	OpenMP	UPC
bt	A	0.4	4.6	10.0		
cg	B	0.6	4.2	9.2	4	2
ep	C	0.0	15.6	15.9	2800	15000
ft	B	5.6	5.3	10.5	73	206
is	C	3.1	4.8	8.4	44	63
mg	C	5.6	5	8.8	16	39
sp	A	0.1	7.2	12.4	2	

Table 2. *Selected NAS parallel benchmarks. RSS is the average resident set size per core as measured by Linux during a run.*

	SPEED % improvement			% variation	
	PINNED	LB av	LB worst	SPEED	LOAD
bt A	8	4	20	1	38
cg B	7	7	26	2	40
ep C	24	46	90	2	67
ft B	14	5	14	1	15
is C	23	10	33	4	40
mg C	15	8	22	3	32
sp A	6	10	33	2	41
all	14	13	34	2	39

Table 3. *Summary of performance improvements for the combined UPC workload. We present the improvements with SPEED compared to PINNED and LOAD ("LB") averaged over all core counts. The percentage variation is the ratio of the maximum to minimum run times across 10 runs.*

on shared workloads are illustrated in Figure 5, which shows EP sharing with an unrelated task that is pinned to the first core (0) on the system. The task is a compute-intensive "cpu-hog" that uses no memory. When EP is compiled with one thread per core, each thread pinned to a core ("One-per-core"), the whole parallel application is slowed by 50% because the *cpu-hog* always takes half of core 0, and EP runs at the speed of the slowest thread. In the case where 16 EP threads are pinned to the available cores (less than 16), the results are initially better because EP gets more of a share of core 0, for instance, $8/9$ of core 0 when running on two cores. But as the number of threads per core decreases, so the *cpu-hog* has more impact, until at 16 cores EP is running at half speed.

LOAD is also fundamentally limited in this test case: there is no static balance possible because the total number of tasks (17) is a prime. However, the performance with LOAD is good because LOAD can balance applications that sleep, such as the OpenMP benchmark shown in Figure 5. When OpenMP is configured to use polling only, LOAD is significantly suboptimal as illustrated by the line labeled LOAD_YIELD. Due to the dynamic approach, SPEED attains near-optimal performance at all core counts, with very low performance variation (at most 6% compared with LOAD of up to 20%). Similar trends are observed for all implementations.

SPEED also performs well when the parallel benchmarks considered share the cores with more realistic applications, such as `make`, which uses both memory and I/O and spawns multiple subprocesses. Figure 6 illustrates the relative performance of SPEED over LOAD when NAS benchmarks share the system with `make -j 16`. We time the compilation of the UPC runtime, which is similar to the complexity of a Linux kernel build. The results shown are for each NAS benchmark being managed by SPEED; `make` is managed by LOAD. LOAD can migrate `make` processes, even in the case when SPEED is balancing the NAS benchmark.

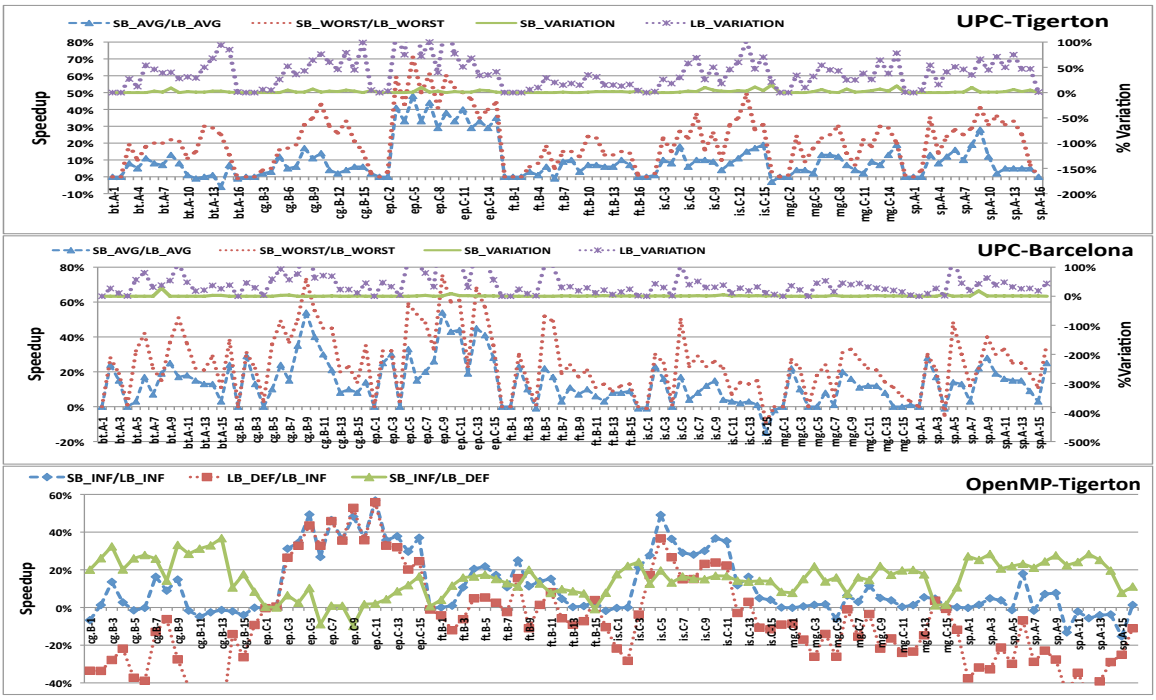


Figure 4. *Performance improvements using SPEED compared to the default Linux load balancing. Each benchmark was executed with 16 threads on the number of cores indicated in the x-axis label, e.g. bt.A-1 executes with 16 threads on one core. The left hand y-axis shows the improvements for SPEED and the right hand y-axis shows the variation of the running time for 10 runs for SPEED and LOAD. AVG refers to the average performance over 10 runs and WORST refers to the worst performance in 10 runs.*

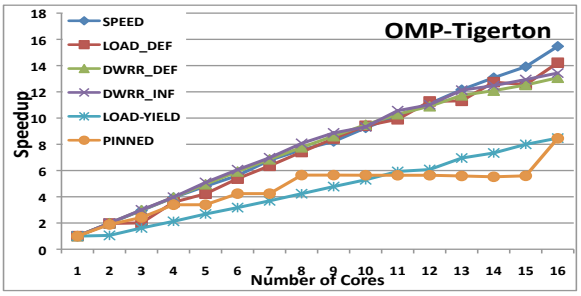


Figure 5. *Speedup for EP, sharing with one external unrelated CPU intensive task. EP is run with 16 threads.*

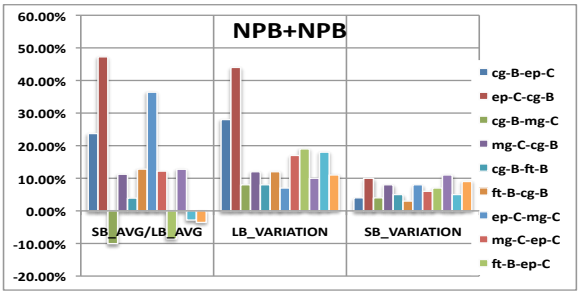


Figure 7. *Balancing applications together. SB_AVG/LB_AVG indicates the performance improvements for the first benchmark in the pair when managed by SPEED. LB_VARIATION and SB_VARIATION show the variation in runtime for each benchmark for the respective balancing method.*

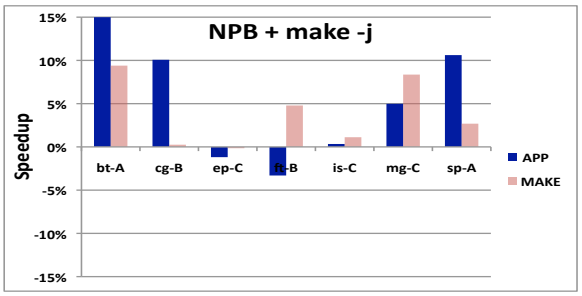


Figure 6. *Relative performance of SPEED over LOAD when running UPC NAS benchmarks together with* `make -j 16`.

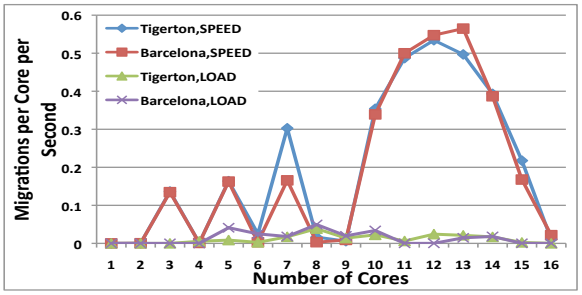


Figure 8. *Migration rates of EP on a dedicated system. All other benchmarks exhibit similar behavior.*

	SPEED % improvement			% variation	
	PINNED	LB av	LB worst	SPEED	LOAD
bt A	0	16	38	2	38
cg B	0	29	66	1	55
ep C	17	43	95	1	86
ft B	5	10	27	1	27
is C	5	7	24	1	30
mg C	2	9	22	1	26
sp A	0	18	35	2	36
all	4	19	44	1	43

Table 4. *Performance improvements of speed balancing over static thread distribution ("PINNED') and load balancing ("LB") for various NPB on the* Barcelona*, averaged over all core counts. The percentage variation is the ratio of the maximum to minimum run times across 10 runs.*

Figure 6 shows that in most cases SPEED improves the performance of both the parallel application and the compilation job: the NAS benchmarks improve by at most 15%, `make` also improves by at most 10%. The experiments were set up so that `make` always runs longer than the NAS benchmark; the relative performance of `make` is reported only for the period during which the NAS benchmark was also running. The presence of an application managed by SPEED also reduces the variability from as high as 23% with LOAD to 11% with SPEED.

SPEED is also effective at balancing multiple parallel applications running simultaneously. Figure 7 shows a representative sample of pairs of NAS benchmarks, compiled with 16 threads and running simultaneously. Each benchmark is managed by its own speedbalancer or un-managed (LOAD). The figure presents the performance improvements for the first benchmark in the pair with SPEED compared to runs where the benchmarks were managed by LOAD. We report the average behavior over 10 runs. SPEED outperforms LOAD by as much as 50% and has very low variability, less than 10%, compared to the variability in LOAD, which is up to 40%.

6.4 NUMA

Speed balancing is clearly effective when the performance impact of migrations is small. On NUMA systems, process migrations are particularly expensive when performed across different memory modules, since the memory pages do not get migrated. Our current strategy for NUMA systems is to block all inter-node migrations. Strictly speaking, this is not necessary when memory access speed is not an issue, e.g. computation bound benchmarks such as EP, or when data fits in the cache. However, the effects of blocking inter-node migrations on EP are not dramatic, as shown in the right-hand plot in Figure 3: the speedup of EP is not far from optimal. By contrast, LOAD is particularly poor on NUMA, even for EP, where non-local memory accesses are not an issue. The slowdown comes from poor initial distribution of threads which LOAD does not correct since they span NUMA nodes.

When inter-node migrations are prevented, the effects of the initial task distribution are particularly important on NUMA: Table 4 shows that SPEED performs similarly to static thread distribution on average, within 4%. This adds support to the idea that the poor performance of LOAD on NUMA is a consequence of initial imbalances that never get properly corrected. Furthermore, the performance for LOAD varies dramatically, (up to 86%), as we would expect if the threads sometimes have poor initial distributions. Also, as expected from the analysis in Section 4, the applications with high synchronization frequency benefit less from speed balancing.

Figure 4 (UPC-Barcelona) shows the performance improvement of SPEED over LOAD at different core counts. Note that SPEED and LOAD consistently perform the same at four cores, in contrast to the *Tigerton*, where SPEED outperforms LOAD at four cores (UPC-Tigerton). Because there are sixteen threads on four cores one would expect there to be no difference between LOAD an SPEED on either architecture. The key difference between the *Barcelona* and the *Tigerton* here is that the four cores are a single scheduling domain (a NUMA node) on the *Barcelona* whereas they comprise two scheduling domains on the *Tigerton* (two die). Thus we conclude that balancing across scheduling domains is responsible for the failure of LOAD to correct an initial imbalance, even without NUMA.

6.5 Migration rates

Speed balancing imposes additional costs because of the higher number of migrating threads. Figure 8 shows that good dynamic balance can be achieved with low migration frequencies (less than one per core per second). Furthermore, the migration frequency drops down to near zero when static balance is attainable and migrations cannot add any performance benefits. Thus SPEED gets the best of both worlds. Blocking inter-node migrations in NUMA makes little difference to the migration frequency. From Figure 8 it is clear than LOAD does not try to attain a dynamic balance; the few migrations seen occur when starting up.

7. Related Work

The mechanisms employed in operating systems schedulers address both temporal and spatial concerns. Traditional temporal scheduling is used to manage the time allocation for one processor and has been the subject of intensive research [13, 23]. Squillante and Lazowska [26] examine scheduling for shared memory UMA systems under time-sharing policies and conclude that locality, as captured by the cache-hot measure in Linux, is important. They advocate for affinity (locality) based scheduling. However, for space-sharing policies on UMA systems, Vaswani and Zahorjan [31] conclude that locality can be safely ignored. Our results for parallel applications running oversubscribed or in competitive environments also indicate that balance trumps locality concerns.

A large body of research in multiprocessor scheduling can be loosely classified as symbiotic scheduling: threads are scheduled according to their resource usage patterns. Snavely and Tullsen [25] present simulation results for symbiotic coscheduling on SMT processors for a workload heavily biased towards multiprogrammed single-threaded jobs. Their approach samples different possible schedules based on performance counters and assigns threads to SMT hardware contexts. They discuss different estimators and in particular point out that either balancing IPC estimators or composite estimators (IPC+cache) perform best in practice. Banikazemi et al [3] present the design of a user space meta-scheduler for optimizing power, performance and energy. They sample performance counters and introduce an algebraic model for estimating thread performance, footprint and miss rate when sharing caches. Their approach has been evaluated only with multiprogrammed workloads. The behavior of their approach for workloads containing parallel applications is not clear.

Tam et al [27] and Thekkath et al [28] examine scheduling techniques to maximize cache sharing and reuse for server workloads and scientific workloads respectively. Tam et al use sampling of the hardware performance counters to cluster threads based on their memory access patterns. Threads clusters are then "evenly" balanced across cores. The focus of their approach is detecting and forming the clusters and not load balancing. Our results indicate that migration rather than "locality" based clustering is beneficial to the performance of parallel scientific applications.

Boneti et al [5] present a dynamic scheduler for balancing high performance scientific MPI applications on the POWER5 processor. Their implementation is architecture dependent and it makes assumptions about implementation behavior. Specifically, they consider a load imbalanced when threads sleep if they are idle waiting for synchronization events. Li et al [16] 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 balancer to use the no-

tions of scaled core speed and scaled load but balancing is performed based on run queue length and they do not discuss migration, competitive environments or uneven task distributions. Their implementation provides heuristics to asses the profitability of migration across NUMA nodes. In particular, their heuristic to migrate across NUMA nodes checks if a task's memory footprint fits in cache. Most of the benchmarks in our workload have per task footprints larger than the caches and therefore will not be migrated. Their heuristics can be trivially added to speed balancing.

Job scheduling for parallel systems has an active research community. Feitelson [1] maintains the Parallel Workload Archive that contains standardized job traces from many large scale installations: in these traces the number of processors required by any job does not exceed the number of processors in the system. Most batch job scheduling approaches require estimates about job length and the impact of wrong estimates [29] is a topic of debate within that community. Ousterhout [22] proposes one of the first implementations of gang scheduling for the Medusa system on the Cm* multiprocessor. Gang scheduling does not require job length estimates and it is effective in reducing wait time for large scale systems, at the expense of increasing the apparent execution time. Zhang et al [33] provide a comparison of existing techniques and discuss combining backfilling with migration and gang scheduling. Gang scheduling is increasingly mentioned in multicore research studies, e.g. [18], but without a practical, working implementation. We believe that by increasing the multiprogramming level (MPL) of the system, the principles behind speed balancing can be easily incorporated within the algorithms to populate the entries in the Ousterhout scheduling matrix. While the job scheduling techniques explored for large scale systems [12] provide potential for improving existing operating system schedulers and load balancers, their behavior for multicore loads is far from understood. One of the open research problems is accommodating the short lived or interactive tasks present in workstation workloads.

Feitelson and Rudolph [8] discuss the interaction between gang scheduling and the application synchronization behavior. They indicate that when the number of tasks in a gang matches the number of processors, gang scheduling combined with busy waiting is able to provide best performance for fine-grained applications. Gupta et al [10] discuss the interaction between the implementation of synchronization operations and operating system scheduling support. They use parallel scientific applications and compare priority based scheduling, affinity and handoff based scheduling, gang scheduling, batch scheduling and two-level scheduling with process control. Their study reports that best performance is obtained by two-level scheduling schemes where the number of an application' tasks is adjusted to match the number of available cores. As our results indicate, speed balancing complements two-level scheduling schemes and it can reduce the performance impact when applications run with more than one task per core.

8. Discussion

This work has been partially motivated by our increased frustration with the level of performance attained by Linux when parallel applications oversubscribe the available cores. This is ongoing research into parallel execution models and a user level implementation allowed us to achieve portability across OS and hardware platforms. Current parallel runtimes are designed for dedicated environments and assume execution with one OS level task per core. Some provide their own thread abstraction and load balancing through work-stealing: Adaptive MPI [11] provides task virtualization for MPI and the SPMD programming model, while Cilk [14] provides for fork-join parallelism. These approaches perform load balancing at the "task" level: stealing occurs when run queue length changes, while our approach balances based on time quanta (or balancing interval). For applications where the task duration is shorter than "our balancing interval", work stealing runtimes are likely to provide better performance. For coarse grained tasks, speed balancing is an orthogonal mechanism that together with time slicing can be added to work-stealing runtimes to improve performance. A generic, runtime independent load balancing implementation that performs well might alleviate the need for complex runtime specific techniques. Furthermore, efficiently mapping multiple threads on each core reduces some of the need for either adjusting the number of tasks at runtime [7] or using profiling and feedback techniques for better performance.

We have started work on exploring the interaction between speed balancing and other programming paradigms: work stealing runtimes (Cilk) and `pthreads` based applications such as the PARSEC [4] benchmarks that use condition variables or `sleep` for synchronization. Preliminary results indicate that Cilk programs that generate a large number of fine grained tasks are oblivious to speed balancing, as are most PARSEC benchmarks, with some notable exceptions, for example, speed balancing improves performance for the *blackscholes* benchmark. Our OpenMP results indicate that best performance is obtained with speed balancing in combination with calling `sched_yield` in synchronization operations. One of the topics of future interest is generalizing speed balancing to handle `sleep` and to answer the question whether implementations that use solely `sched_yield` can provide best performance when combined with proper load balancing such as speed balancing or DWRR.

In our experiments we have obtained better performance using speed balancing for runs longer than a few seconds; this is the price we had to pay for generality. For short lived applications, the behavior is that of the default OS load balancing implementation. Our balancer has a startup delay from polling the */proc* file system to capture the application threads. Packaging the balancer as a library, instead of a standalone application, and integrating with the runtime is likely to improve the results for short lived applications.

Our design provides room for many other functional and algorithmic refinements. Distinguishing between user and system time when measuring speed allows the balancer to classify tasks based on their overall behavior. In this way the task mix can be controlled based on perceived behavior and for example system service queues can be dynamically created and managed. This distinction might also be beneficial for ad hoc variants that try to infer the synchronization behavior of applications. The current implementation provides a slow rate of change in queue membership by moving only one task. For systems where process spawners are not aware of parallelism a simple change to select the number of candidates based on weighted speed difference will hasten reaching dynamic equilibrium.

9. Conclusion

We have shown that specializing the load balance mechanism for parallelism is beneficial to performance and can coexist with the system level load balancers. We presented the design, some theoretical foundations and an user level implementation able to improve the performance of OpenMP, MPI and UPC applications. Traditional load balancing mechanisms attempt to balance run queue length globally and disfavor migrations across nearly balanced queues, a design that ignores the needs of parallel applications where threads should make progress at equal rates. Speed balancing balances the time a thread spends on fast and slow queues, regardless of run queue length. We evaluated it with multiple parallel and multiprogrammed workloads on contemporary UMA and NUMA multicores and observe both increased isolation and performance. Our balancer is able to greatly reduce variation in execution time

and it improves performance over the Linux load balancer by up to a factor of two.

References

[1] Parallel Workload Archive. Available at http://www.cs.huji.ac.il/labs/parallel/workload/.

[2] 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. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] M. Banikazemi, D. E. Poff, and B. Abali. PAM: A Novel Performance/Power Aware Meta-Scheduler for Multi-Core Systems. In *Proceedings of the ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SC'08)*, 2008.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT '08: Proceedings of the 17th International Conference On Parallel Architectures And Compilation Techniques*, pages 72–81, New York, NY, USA, 2008. ACM.

[5] 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.

[6] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski, and J. Rattner. Platform 2015: Intel Processor and Platform Evolution for the Next Decade. White Paper, Intel Corporation, 2005.

[7] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Trans. Parallel Distrib. Syst.*, 19(10):1396–1410, 2008.

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

[9] K. B. Ferreira, P. Bridges, and R. Brightwell. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008.

[10] A. Gupta, A. Tucker, and S. Urushibara. The Impact Of Operating System Scheduling Policies And Synchronization Methods On Performance Of Parallel Applications. *SIGMETRICS Perform. Eval. Rev.*, 19(1), 1991.

[11] 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)*, pages 306–322, 2003.

[12] M. A. Jette. Performance Characteristics Of Gang Scheduling In Multiprogrammed Environments. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM)*, 1997.

[13] L. Kleinrock and R. R. Muntz. Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time Shared System. *J. ACM*, 19(3):464–482, 1972.

[14] Kunal Agrawal and Yuxiong He and Wen Jing Hsu and Charles E. Leiserson. Adaptive Task Scheduling with Parallelism Feedback. In *Proceedings of the Annual ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2006.

[15] 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*, pages 65–74, New York, NY, USA, 2009. ACM.

[16] 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.

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

[18] 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.

[19] The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.

[20] The UPC NAS Parallel Benchmarks. Available at http://upc.gwu.edu/download.html.

[21] R. Nishtala and K. Yelick. Optimizing Collective Communication on Multicores. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, 2009.

[22] J. Ousterhout. Scheduling Techniques for Concurrent Systems. In *In Proceedings of the 3rd International Conference on Distributed Computing Systems (ICDCS)*, 1982.

[23] D. Petrou, J. W. Milford, and G. A. Gibson. Implementing Lottery Scheduling: Matching the Specializations in Traditional Schedulers. In *ATEC '99: Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 1999.

[24] J. Roberson. ULE: A Modern Scheduler for FreeBSD. In *USENIX BSDCon*, pages 17–28, 2003.

[25] A. Snavely. Symbiotic Jobscheduling For A Simultaneous Multithreading Processor. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, 2000.

[26] M. S. Squillante and E. D. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 1993.

[27] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, 2007.

[28] R. Thekkath and S. J. Eggers. Impact of Sharing-Based Thread Placement on Multithreaded Architectures. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1994.

[29] D. Tsafrir, Y. Etsion, and D. G. Feitelson. Backfilling Using System-Generated Predictions Rather Than User Runtime Estimates. *In IEEE TPDS*, 2007.

[30] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications. In *ICS '05: Proceedings of the 19th Annual International Conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM.

[31] R. Vaswani and J. Zahorjan. The Implications Of Cache Affinity On Processor Scheduling For Multiprogrammed, Shared Memory Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 1991.

[32] 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.

[33] 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.