# Adaptive Space-Shared Scheduling for Shared-Memory Parallel Programs

Younghyun Cho, Surim Oh, and Bernhard Egger(✉)

Department of Computer Science and Engineering,
Seoul National University, Seoul, Korea
{younghyun,surim,bernhard}@csap.snu.ac.kr

**Abstract.** Space-sharing is regarded as the proper resource management scheme for many-core OSes. For today's many-core chips and parallel programming models providing no explicit resource requirements, an important research problem is to provide a proper resource allocation to the running applications while considering not only the architectural features but also the characteristics of the parallel applications.

In this paper, we introduce a space-shared scheduling strategy for shared-memory parallel programs. To properly assign the disjoint set of cores to simultaneously running parallel applications, the proposed scheme considers the performance characteristics of the executing (parallel) code section of all running applications. The information about the performance is used to compute a proper core allocation in accordance to the goal of the scheduling policy given by the system manager.

We have first implemented a user-level scheduling framework that runs on Linux-based multi-core chips. A simple performance model based solely on online profile data is used to characterize the performance scalability of applications. The framework is evaluated for two scheduling policies, balancing and maximizing QoS, and on two different many-core platforms, a 64-core AMD Opteron platform and a 36-core Tile-Gx36 processor. Experimental results of various OpenMP benchmarks show that in general our space-shared scheduling outperforms the standard Linux scheduler and meets the goal of the active scheduling policy.

## 1 Introduction

Modern operating systems (OSes) are still based on time-shared scheduling techniques originally developed for single-core machines where – despite local run queues – one system kernel maintains the entire information about the system and manages all running tasks in the same manner. At the same time, current parallel programming models such as OpenMP [11], TBB [23], Cilk [7], or OpenCL [16] assume that each application can utilize all physically available hardware resources without considering the current system workload.

In such a disjoint runtime model, OSes lack good scheduling policies when multiple parallel programs are executed simultaneously. Averse effects of this approach include not only a low cache utilization caused by cache cold misses


N. Desai and W. Cirne (Eds.): JSSPP 2015/2016, LNCS 10353, pp. 158–177, 2017.
DOI: 10.1007/978-3-319-61756-5_9

due to context switches of an oversubscription of threads to single cores but also performance interference caused by scheduling threads in a time-shared manner. For example, the standard Linux scheduler, the Completely Fair Scheduler (CFS), provides the same execution chances for each task [21]. Each task's virtual (execution) time is accumulated in proportion to the task priority. This per-task resource management can cause severe performance interference between applications comprising a large number of threads.

A promising approach to address this issue is space-shared resource allocation [28]. In the OS community, several research groups have introduced experimental OS prototypes [5,18,30]. The presented prototypes commonly follow the principle of space-partitioning for scalable many-core resource management. In this new model, the role of the OS is divided into two parts. The *coarse-grained resource manager* provides space-shared resource multiplexing while the application-specific *fine-grained resource manager* manages an application's resource management such as, for example, assigning the application's tasks to the allocated cores. Although the space-shared OSes introduced new OS design principles, the scheduling and resource allocation schemes have lots of open issues to be solved. For traditional parallel job scheduling, a lot of space-shared scheduling schemes such as First-come First-served or Backfilling [14,17,20] have been proposed. However, the prior art mostly focuses on supercomputers or distributed systems where the resource requirements of parallel jobs are typically explicit. For today's many-core chips and parallel programming models that provide no explicit resource requirements, an important research issue of space-shared scheduling is thus to provide a proper resource allocation to the running applications while considering not only the architectural features but also the characteristics of the parallel applications.

In this paper, we introduce a space-shared scheduling strategy for shared-memory parallel programs. Many parallel applications exhibit varying resource requirements and performance characteristics during the execution lifetime of the application. It is therefore increasingly important to consider an application's dynamically changing workload. The proposed scheduling scheme considers all parallel code section of an application. To properly assign the core resources, the scheduler considers the performance characteristics of the currently executing parallel code sections of all running applications. Based on profiled performance information, the scheduler allocates the core resources in accordance with the active scheduling policy which is given by the system manager.

We have first implemented a user-level scheduling framework that performs the proposed scheduling strategy for OpenMP programs on Linux-based many-core systems. In shared-memory systems, the memory access contention is one of the major limiting factor of performance scalability. By extracting the shared-memory access patterns, a simple performance model is used to characterize the performance scalability of running applications. In this work, we implement two Quality-of-Service (QoS) based scheduling policies: Equalizing QoS, and Maximizing QoS. The scheduling framework is evaluated for various OpenMP benchmark scenarios on two different many-core platforms, a 64-core AMD Opteron

NUMA system and a 36-core Tile-Gx36 processor. The empirical studies in this paper show that our space-sharing scheme outperforms the current Linux standard scheduler and confirm space-sharing has potential as a resource management scheme for current and future many-core systems.

The remainder of this paper is organized as follows. Section 2 discusses related work. In Sect. 3, we introduce our approach to dynamic space-shared scheduling. Section 4 describes our scheduler framework including the performance model and the scheduling algorithm. In Sect. 5, we evaluate our scheduling and show how our performance model and scheduling scheme meet the requirements of the active scheduling policy. Section 6 discusses where we see additional room for further performance improvement and the research direction towards better scheduling. Finally, we conclude this paper in Sect. 7.

## 2 Related Work

For shared-memory many-core systems several empirical scheduling techniques have been proposed that consider applications' performance characteristics.

Moore and Childers [19] introduce a utility model to characterize an application's scalability and performance interference between multi-threaded applications based on offline training. Their model finds the performance plateau of an application and uses the information to choose the thread counts for multi-threaded applications. Grewe *et al.* [15] decide thread counts of OpenMP applications based on prediction. The prediction relies on machine learning for the selected performance features. Both works decide the proper thread count when a parallel program is launched, but they do not consider varying workloads in an application, and their techniques rely on additional efforts for offline training.

For more fine-grained resource management, Raman *et al.* [22] propose Parcae, a compiler-assisted dynamic tuning system. Their compiler generates flexible codes for data- and pipeline-parallelism such that they can change the degree of parallelism at runtime. They find the proper thread count by iteratively trying different thread configurations based on a hill-climbing algorithm. However, the hill-climbing approach often fails to find globally optimal thread counts. Emani *et al.* [13] use a machine learning technique and compiler-assisted information to predict better OpenMP thread counts. On the other hand, Creech *et al.* [10] introduce SCAF that decides the proper thread count of OpenMP parallel applications when it enters a parallel region. The scalability of an OpenMP application is determined by creating and running a serial process concurrently with the parallel section. An online profiler then compares the progress of the serial process to that of the multi-threaded process based on the instructions per cycle count. Since SCAF does not change the parallelism once the number of threads has been determined, their method is useful for iterative programs in that the serial process is executed once when a parallel section is first executed. These works aim at improving system performance by dynamically tuning the parallelism, but the techniques require application runtime support to dynamically manage the degree of parallelism. For HPC (High Performance Computing)

systems, Breitbart *et al.* [8] perform co-scheduling of multiple applications on a multi-core node. To maximize system throughput their scheduler automatically detects an application's main memory bandwidth utilization. On the other hand, we focus on a spatial mapping of parallel programs for managing QoS of applications and do not consider changing active thread counts of applications.

Sasaki *et al.* [24] consider the scalability of multiple multi-threaded applications and perform a spatial partitioning without offline information. The work also focuses on balancing performance of co-located applications via spatial partitioning. To understand the performance scalability of applications, they run the same application on three different core allocations at runtime and measure how the application's IPS (Instructions Per Second) changes for the different resource configurations. In addition, they detect applications' phases to deal with varying workload characteristics. Our scheduling scheme, on the other hand, require a communication API between the space-shared scheduler and the application runtimes. We also aim to characterize the application's performance scalability without runtime reconfiguration and adaptively change the scheduling goals according to the input given by the system manager.

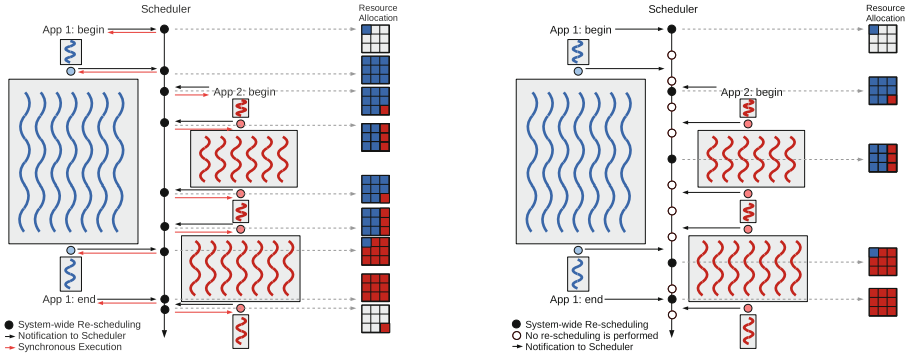
## 3 The Space-Shared Scheduling Strategy

In this section, we introduce a space-shared scheduling strategy for scheduling multiple shared-memory parallel applications. It is obvious that, even when strictly adhering to a space-shared resource allocation scheme, the allocations are of temporal nature and must be re-evaluated in order to efficiently utilize the given hardware and achieve satisfactory performance. In general, applications can start and end at any given time which will require a re-computation of the current resource allocation. More importantly, the resource requirements of a parallel application are likely to vary over the course of its execution. For example, in sequential sections, one core should be sufficient whereas the different parallel regions of the application may each exhibit divergent characteristics and will thus require a different resource allocation. Assigning the correct amount of resources to all applications at any given time while satisfying both the overall system's and each application's performance goals is thus undoubtedly a challenging problem.

### 3.1 Interaction with Application Runtimes

Parallel applications consist of several parallel and sequential regions, and the different parallel applications typically posses divergent characteristics [4]. To consider the varying performance characteristics of parallel applications, we propose an approach in which the space-shared scheduler interacts directly with the parallel programming model runtimes of the applications.

A naïve but easy approach is to use synchronous communication between the scheduler and the application runtimes. Figure 1(a) illustrates the approach. System-wide resource re-allocation is performed whenever an application enters




(a) Synchronous scheduling (b) Asynchronous scheduling

**Fig. 1.** Adaptive space-shared scheduling scheme.

or exits a parallel region. However, this synchronous scheduling model suffers from a number of problems. The synchronous communication model serializes accesses of parallel applications to the scheduler, thereby effectively limiting parallelism. In addition, re-evaluating the resource allocations whenever one of the concurrently executing parallel applications enters or exits a (possibly very short) parallel region leads to an unacceptably high overhead of the scheduler.

A better approach is to perform asynchronous communication and scheduling as illustrated in Fig. 1(b). In this scheme, parallel applications communicate with the scheduling runtime by sending their status (i.e., events about entering and exiting parallel regions) to the scheduler and asynchronously execute their jobs. The scheduler periodically wakes up to re-evaluate the current resource allocation of the simultaneously running application. Since re-scheduling and re-allocating resources to applications are comparatively expensive operations, re-scheduling is not performed at every wake-up but only if the system state is stable for a long enough period of time. Without that condition, an application executing a short parallel regions in a loop might trigger a system-wide re-scheduling of resources. Section 5 discusses the wake-up frequency and defines what we consider to be a stable system state. The proposed scheduling approach does not limit progress of parallel applications thanks to the asynchronous communication model and, at the same time, is able to reduce frequent inefficient resource re-allocations.

We need to define a method to communicate with the applications or the application-specific runtimes. Applications have to notify the global scheduler about their state. A parallel application is either executing a sequential code section or in one of its parallel sections. We do not distinguish between different sequential code sections, however, the scheduler considers the different parallel sections of an application. The *execution state* of an application from the viewpoint of the scheduler is defined as (a) *sequential* or (b) by the *unique identifier of a parallel section.*

The calls to the scheduler can be inserted manually or automatically in to the source code, made by a customized parallel programming runtime library, or by intercepting the application's calls to the runtime through dynamic library interpositioning. In this paper, we have implemented the last approach for OpenMP applications, dynamic library interpositioning, because it is automatic and neither requires access to the application's source code nor modifications to the parallel runtime library. An easy way to uniquely label parallel sections is to use the address of the parallel code section. Section 4.2 elaborates the details of this approach.

### 3.2 Application Runtimes and Programming Malleability

Since no application shares core resources with any other application in a space-shared approach, we first need to reclaim the cores from an application before redistributing them to other applications. This functionality has to be implemented without much runtime overhead in order to be beneficial.

To add or reclaim cores from an application, we need to consider what programming model is used. The most prominent parallel programming models are thread-based fork-join model (e.g., OpenMP), task-parallel models (e.g., Cilk, TBB, or OpenMP 3.0) and data-parallelism (e.g., OpenCL, CUDA).

In the OpenMP fork-join model, it is impossible to reduce or increase the number of parallel threads once the worker threads have been created and assigned with a portion of the workload at the entrance of a parallel region unless additional compiler or runtime support is provided. On the other hand, task-parallel models and data-parallel models present opportunities to reclaim and reassign resources during the execution of a parallel section. For example, task-parallel programming models such as Cilk or TBB have a scheduler which distributes tasks for all cores. Also, data parallel programming models such as OpenCL, a possibly large number of work units (termed *work groups* in the OpenCL model) are distributed to a pool of worker threads by the OpenCL work group scheduler. For those programming models, increasing or decreasing the number of active worker threads is easily achieved.

In this paper, we tackle the OpenMP programming model and thus focus on scheduling a fixed number of threads. Instead of adjusting the number of threads, we employ a thread-migration based approach. We allow different applications to share core resources temporarily during a migration phase until the required migrations have finished. Changing the thread count during execution is outside of the scope of this paper and part of future work.

### 3.3 Performance Model and Scheduling Policies

A space-shared scheduler needs to provide sufficient resources to all running applications while considering each application's characteristics and hardware features at the same time. Also, the resource manager should be able to compute a proper resource allocation according to the specific scheduling policy. In order to do so, the scheduler needs information about the applications' runtime profiles.

There are some important features that affect an application's performance scalability. Memory access patterns, inter-core communication patterns, and the (dynamic) working set input size can all affect the scalability of an application. To understand the performance characteristics of parallel applications, prior works usually employ offline training, [19], machine learning [13,15], extracted information through static program analysis [29], runtime resource reconfiguration [10,22,24], or analytical approaches such as, for example, resource contention modeling [26,27].

In this work, we implement a performance model that solely relies on online profile data. The model assumes that the memory access contention is the major limiting factor of performance scalability. Section 4.3 shows this assumption is valid to capture the trend of an application's scalability. More sophisticated shared resource contention modeling and additional performance information from applications' annotation or static analysis can potentially obtain more accurate and versatile performance models and allow implementation of more sophisticated scheduling policies. Better analysis techniques and policies are part of future work.

## 4 The Scheduling Framework

In this section, we discuss the details of the proposed scheduling framework that performs dynamic space-shared resource allocation as outlined in Sect. 3. The framework is implemented on Linux-based many-core platforms and performs scheduling for GNU-OpenMP applications. Other parallel programming models can be easily supported by defining and implementing the communication interface into the respective parallel programming library.

Figure 2 illustrates the system software stack of the framework. We explain each component in the subsections below.

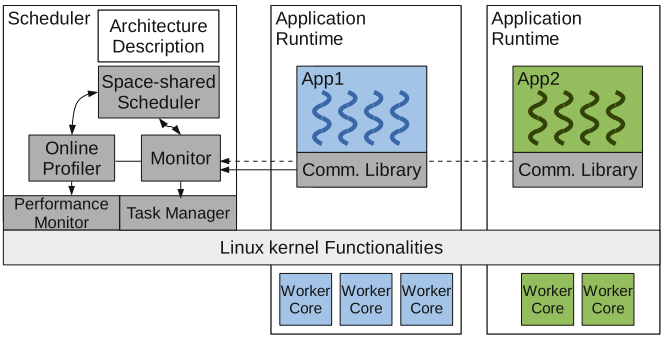



**Fig. 2.** The scheduling framework.

### 4.1 Core Mapping Scheme

In our scheduling framework, the scheduler allocates a `cluster` (a set of computing cores that share a common last-level cache (LLC)) as the default scheduling granularity. Exceptions are allowed in two cases: (1) when a serial section is scheduled or (2) when the number of running applications is bigger than the number of `clusters` in the system. This is sensible because each application can benefit from cluster-level management in terms of LLC sharing.

Furthermore, the core allocator tries to maintain shapes of minimal perimeter inside each `cluster`. This has a positive effect on inter-core communication and also reduces the number of migrations at runtime caused by resource reallocations. For some architectures, especially mesh-style NoCs, the overhead of the cache coherence protocol is high if the communication distance is long.

### 4.2 Communication Library

The runtime environment manager is implemented in a component called *monitor* (Fig. 2). This *monitor* module runs as a daemon and interacts with the application runtimes, the space-shared scheduler and the online profiler. The monitor periodically tries to perform system-wide scheduling if all running applications execute in a specific parallel/serial code region for a sufficiently long enough time. The wake-up frequency of the scheduler is an parameter that depends on the target architecture platform.

Our scheduler needs to keep track of the contexts of all running applications. In other words, the scheduler requires information whether a given application is currently executing in a sequential or a parallel section. Since different parallel sections exhibit different performance characteristics, it also needs to distinguish between the different parallel sections.

In our implementation, the current context of an application is stored to a global memory region that is shared with the scheduler. An OpenMP application calls `GOMP_parallel_start` and `GOMP_parallel_end` when it enters and exits a parallel region, respectively. Our framework intercepts these calls through library interpositioning. The function pointer of the parallel section provided to the `GOMP_parallel_start` upon entering a parallel code section is used to distinguish between different parallel sections.

### 4.3 Online Profiler

A proper performance model is required in order to compute resource allocations to the different parallel applications with respect the current scheduling policy. Our online profiler collects important performance features by monitoring the hardware's performance counters and computes a performance model for each encountered parallel code section of every application.

Our model is based on the idea that shared resource contention among threads is one of important factors that affect applications' performance scalability. Tudor *et al.* [27] introduced an analytical memory contention model for

shared-memory parallel programs. In their work, the authors show that shared-memory resource contention can be modeled using an M/M/1 queue. We apply this idea to efficiently characterize the performance characteristics of the concurrently executing applications.

**Memory Contention Performance Model.** The performance (i.e., speedup) model is organized as follows. The specific amount of work that a parallel section of an application performs is denoted by the *work cycles* $W$. If a parallel section uses $N$ cores then the work cycles are divided by $N$. If the threads are completely independent, i.e., in the absence of inter-core communication or access to shared resources, the application's parallel section speedup becomes $N$. Most applications, however, access shared resources such as memory. The shared-resource contention is denoted by $C(N)$. In our model, we currently consider only memory-level contention. We also don't take into account other features such as load-imbalance and data dependencies because experiments with OpenMP benchmarks (especially in a parallel section) exhibit a much larger sensitivity to the contention in memory accesses $C(N)$.

To estimate the speedup, we measure the per-core last-level cache miss rate of each application, denoted $LLC$. In addition, we compute $LLC\_ALL$, the sum of the total LLC miss rates from all applications running in parallel. If $N$ cores are assigned to an application, the speedup model is given as follows:

$$SpeedUp(N, LLC, LLC\_ALL) = \frac{W + C(1)}{W/N + C(N)} \tag{1}$$

To compute the shared memory contention overhead $C(N)$, we first estimate how many memory accesses happen during the given work cycles ($W/N * LLC$). Then we can compute the total number of cycles required to finish the given work cycles by multiplying the expected memory service time ($T(N)$) (i.e., latency cycles) as shown below:

$$C(N) = \frac{W}{N} * LLC * T(N) \tag{2}$$

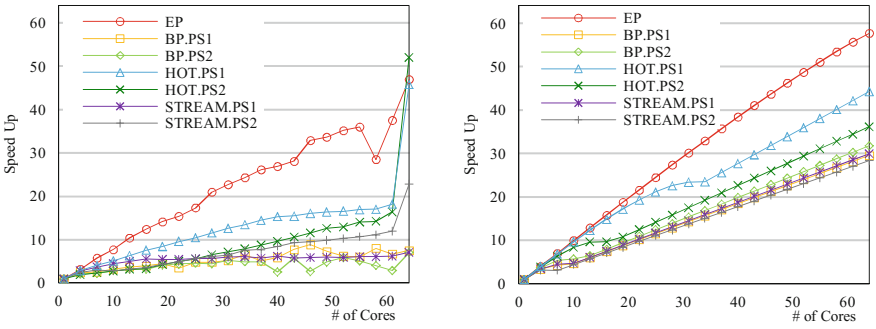
The service time is modeled by an M/M/1 queuing model. In this work, we assume that if the system contains a number of memories, the memory accesses are evenly distributed (interleaved) to each memory. We further assume that the memory service times are the same regardless of the distance between memory controllers and cores in the interleaved allocation scheme. The memory latency cycles without any contention are modeled as $L$ and the number of memories is represented by $M$.

$$T(N) = \frac{1}{service\ rate - request\ rate} = 1/(\frac{1}{L} - \frac{LLC\_ALL}{M}) \tag{3}$$

This model efficiently generates a logarithmic scalability curve by modeling the increased contention overhead. However, this model is not theoretically valid for our problem. Foremost, the model assumes an infinite number of resource

competitors, but we only have a finite number of cores. This possibly incurs negative or impractically large estimated latency cycles because the simple regression ($LLC * N$ in Eq. 3) increases $LLC$ without considering the service response (the response time reduces the LLC miss rate of a queuing competitor). In addition, in modern memory architectures, there are a number of distributed contention points. To overcome this limitation, we use a threshold for the delay. Once the estimated latency reaches the pre-determined threshold in memory-intensive applications, the estimated latency is fixed to the threshold and the scalability curve becomes linear.

**Examples of the Performance Model.** Figure 3 shows some examples of the model. The baseline in Fig. 3(a) is obtained by running the applications with a varying number of cores on the target machine, a 64-core AMD architecture. Each benchmark is then executed standalone on all available cores and the LLC miss-rate is obtained by monitoring the performance counters. The modeled performance scalability is depicted in Fig. 3(b). Linear performance scalability in the graph is caused by the capping the maximum service delay with a threshold as outlined above.




(a) Original performance (b) Performance model

**Fig. 3.** Examples of the performance model. EP is taken from NPB3.3 [25], the others (BP-backpropagation, HOT-hotspot, STREAM-streamcluster) are from Rodinia [9]. We show the results of benchmarks containing one or two parallel sections (PS).

The performance model does not capture various performance features such as data dependencies, inter-core communication or synchronization patterns. However, as long as the relation between the different applications' scalabilities are predicted correctly, the absolute error of the model is not of great importance in order to compute a resource allocation.

**Performance Counter Measurement.** The performance indicators required by the performance model are obtained by monitoring the hardware performance counters. We measure LLC miss events and total cycles.

Our framework is tested on a 64-core AMD Opteron platform and the Tile-Gx36 platform. For the AMD platform [3], the LLC miss event counters are provided on the AMD NorthBridge and we can obtain the count using the "NBPMCx4E1 L3 Cache Misses" [12] as an event descriptor. The total number of cycles is a generalized CPU event which is already defined and measured in the Linux kernel. On the other hand, the Tile-Gx36 does not have a specific last-level cache. Instead, the architecture uses DDC (Dynamic Distributed Cache) techniques [2] in which local cache misses try to fetch their data from distributed caches. For the Tile-Gx36 NoC architecture, we consider local cache misses (from local cache to memory) and remote cache misses (from remote cache to memory) at the same time. $LLC$ is computed as the sum of the two.

### 4.4 Scheduling Algorithm

The main advantage of the performance model is that we can design different scheduling algorithms to accommodate for specific scheduling policies. In this work, our scheduling policies are based on the QoS which we define as the normalized speedup compared to executing the application on fully-available hardware resources without co-located applications. QoS is computed as follows:

$$QoS = \frac{SpeedUp\ On\ Given\ Resources}{SpeedUp\ Executed\ Standalone} \tag{4}$$

To maintain a polynomial-time algorithm, the scheduler implements greedy-pareto policies (e.g., maximize QoS, balancing QoS). The algorithm first reserves at least one allocation unit to each application. Whenever the scheduler reserves a new allocation unit, it takes the best (pareto) solution according to the scheduling policy.

Algorithm 1 decides the proper amount of core resources based on the balancing QoS policy. The computational complexity is $O(N^2M)$ where $N$ is the number of applications and $M$ is the number of allocation units. This is an acceptable overhead because the number of executed applications is usually small. Also, we allocate `cluster` as a default allocation granularity (refer Sect. 4.1) which reduces the complexity as well.

After the core resources for all applications have been reserved, we consider core clustering among applications when more than two applications are packed into one `cluster`. For example, the Tile-Gx36 has 36 tiles in a single chip because the machine has no specific LLCs. In addition, on the tiled architecture, the clustering benefits from dynamic distributed caching technique because it can reduce the cost of maintaining cache coherence.

Another important consideration is to reduce the number of thread migrations (i.e., re-assignment to a different core) caused by the system-wide rescheduling. In this work, we have implemented a rather simplistic approach in which the scheduler always allocates the cluster/core resources to applications in the same order in order to minimize the number of migrations.

**Algorithm 1.** Scheduling Policy: Balancing QoS

```
N = # of applications in the system
M = # of computing cores in the system
CPU[N] = # of reserved cores for each application, initialized to 1
LLC[N] = per-core LLC miss rate of each application

while sum(CPU) < M do
    best_app = −1
    min_variance = ∞
    for i = 0 to N − 1 do
        QoS[N] = estimated QoS for each applications
        LLC_ALL = summation of LLC miss rate from all applications
        for j = 0 to N − 1 do
            if i == j then
                LLC_ALL += LLC[j]*(CPU[j]+1)
            else
                LLC_ALL += LLC[j]*CPU[j]
        for j = 0 to N − 1 do
            if i == j then
                QoS[j] = SpeedUp(CPU[j]+1, LLC[j], LLC_ALL) / SpeedUp(M, LLC[j], LLC[j]*M)
            else
                QoS[j] = SpeedUp(CPU[j], LLC[j], LLC_ALL) / SpeedUp(M, LLC[j], LLC[j]*M)
        if variance(QoS) < min_variance then
            min_variance = variance(QoS)
            best_app = i
    CPU[best_app] += 1
```

### 4.5 Task Manager

Another important consideration is the application of internal task scheduling in the application-specific runtime. To assign an application's tasks to specific cores, the framework comprises a special kernel module. As our main concern is not application-specific resource management but space-shared mapping, we utilize the Linux kernel's processor affinity mask to define the set of cores that can be utilized by the threads of an application. The standard Linux scheduler is responsible for thread allocation to the assigned cores and load-balancing. In this way, we can focus on coarse-grained resource allocation techniques and leave the application-specific fine-grained thread-to-core assignment to the Linux kernel.

## 5 Evaluation

### 5.1 Target Architectures

The scheduling framework has been evaluated on a 64-core AMD Opteron 6380 server platform [3] and the Tile-Gx36 platform [1]. The AMD Opteron server represents a multi-socket multi-core NUMA system, and the Tile-Gx36 platform is a representative of a mesh-style many-core processor. The main features for performance evaluation of the two architectures are shown in Table 1.

### 5.2 Target Applications

For the evaluation, we selected several OpenMP applications which have specific characteristics from known benchmark suites. The three OpenMP applications

**Table 1.** Target Architecture.

| Architecture | AMD64 | Tile-Gx36 |
|---|---|---|
| Processor | Opteron6380 | Tile-Gx8036 |
| Clock frequency | 2.5 GHz | 1.2 GHz |
| Memory size | 128 GB | 32 GB |
| Total cores | 64 | 36 |
| # of cores per processor | 8 | 36 |
| # of NUMA nodes | 8 | 2 |
| Linux kernel | 3.13 | 2.6.40 |
| Scheduling frequency | 33 Hz | 13 Hz |
| Scheduling steady state | 2 periods | 2 periods |

(EP, CG, MG) from SNU-NPB3.3 [25], Freqmine from Parsec 3.0 [6] and Streamcluster from the Rodiana [9] benchmark suite all exhibit different characteristics: EP is CPU intensive, CG issues irregular memory accesses, MG is a memory-intensive benchmark, Freqmine is CPU-intensive with a long sequential part, and Streamcluster is also a memory-intensive benchmark. Execution information about each benchmark is shown in Table 2. The standalone execution and speedup values in the table are obtained by executing the benchmarks standalone on the AMD Opteron platform. For Tile-Gx36, we use the same benchmarks but adjust the working set sizes because the Tile-Gx8036 processor has less processing power than the AMD Opteron6380 processor.

**Table 2.** Target application (A-AMD64, T-Tile-Gx36).

| Application | Description | Serial time | Standalone execution (Speed Up) |
|---|---|---|---|
| EP | Embarrassingly parallel | - | A-21.4s(47.1) T-43.6(34.1) |
| CG | Conjugate gradient | - | A-11.6s(9.3) T-40.1(34.2) |
| MG | Multi-grid | - | A-17.2s(10.7) T-32.2(23.9) |
| F.M (Freqmine) | FP-growth method | A-5.8s T-7.1s | A-25.5s(17.8) T-34.0(4.4) |
| S.C (Streamcluster) | clustering | - | A-15.8s(7.7) T-34.7(15.2) |

One benchmark scenario comprises several parallel application benchmarks executed simultaneously. The 10 benchmark scenarios are composed of different application benchmarks representing different workload patterns in order to show the broad applicability of the proposed method.

There are some considerations of the applications' executions with respect to evaluation of performance. First of all, the working set size (execution time) is an important factor for performance evaluation. For example, if the working set sizes are too different between applications, then the normalized performance may vary too much. Therefore, we manually adjust the working set sizes for the

target applications to have a similar turnaround time when they are executed standalone on the target architecture. We further use the system's memory interleaving option (using *numactl* tool to manage NUMA settings) such that memory allocations are evenly distributed across the available memory controllers.

To evaluate the scheduling performance for simultaneous applications, we measure the performance for each application and compute the summation of the performance metrics (QoS, speedup, and turnaround time) from simultaneous applications. We also compute the standard deviation for each performance metric to quantify how balanced the applications' performances are.

### 5.3 Scheduling Policies

To evaluate the space-shared scheduling policies, we compare the scheduling performance with not only the time-shared scheduler but also other simple space-shared scheduling policies. In these experiments, the different schedulers/scheduling policies are as follows.

- CFS - the Linux standard time-shared scheduler.
- Static Equal Partition - static equal core partitioning for each application. In this scheme, system resource re-allocation is not performed (not even when an application finishes execution).
- Dynamic Equal Partition - dynamic equal core partitioning: perform dynamic scheduling whenever an application starts or finishes execution.
- Dynamic Equal QoS - our space-shared scheduling scheme where we perform fine-grained resource management and strive to balance the QoS among applications whenever a core resource is reserved.
- Dynamic Max QoS - our space-shared scheduling scheme. The scheduling policy is set to maximize the sum of the QoS of all applications.

### 5.4 Scheduling Scenario

We execute every combination of three applications from the five target applications as shown in Table 2 for each scheduling policy. Thus, the overall scheduling runs comprise ten sets of applications. We provide the benchmark results of the scheduling set for five different policies both on AMD64 and the Tile-Gx36 platform. In each scenario, each application creates as many threads as physical cores are available in the system, and the thread count is not changed during an application's lifetime.

**64-Core AMD Opteron Platform.** Figure 4 shows the performance of the framework on the 64-core AMD Opteron platform. The first graph in the figure represents the summation of QoS among applications. The second graph shows its speedup, and the last graph shows the turnaround time among applications.

An important consideration is the variance of the three target applications' performances. We show the standard deviation of the performance of three simultaneous applications on the top of each bar. Longer lines indicate a bigger standard deviation; a good scheduler should provide low standard deviations.

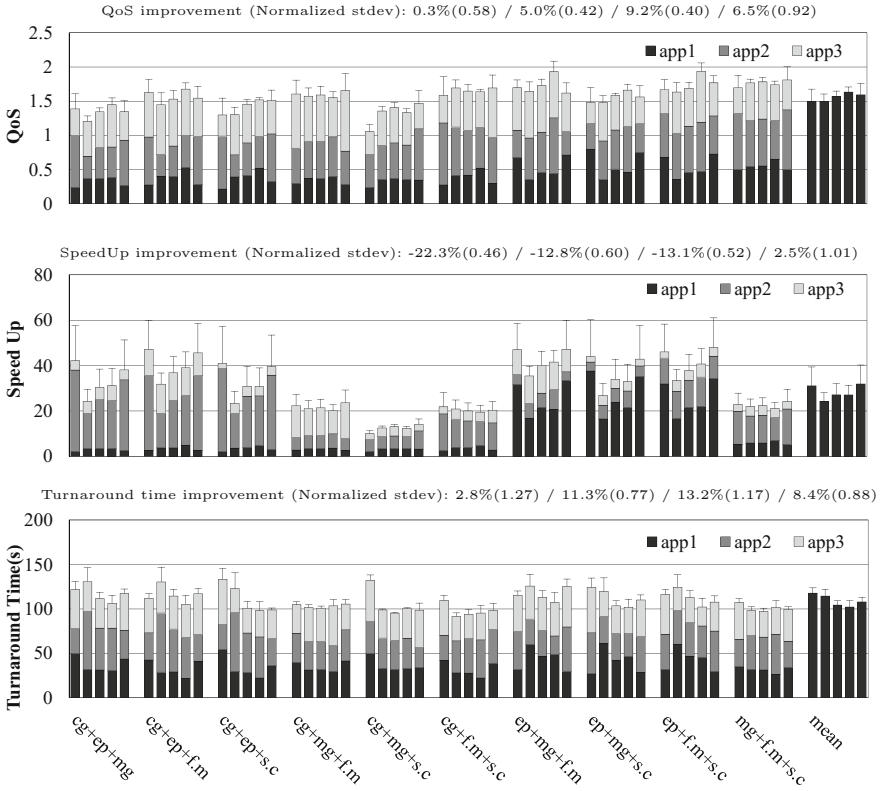



**Fig. 4.** Benchmark result: 64 core AMD Opteron platform. The five bars for each scheduling scenario represent the results in the order CFS, Static Equal Partition, Dynamic Equal Partition, Equal QoS, and Max QoS. The four values on the top of each graph compare the performance (and standard deviation) of each scheduling policy to CFS in terms of the performance metric from left (Static Equal Partition) to right (Max QoS).

For the QoS analysis, higher is better. QoS is an important metric, because if a scheduler only considers maximum speedup then the scheduler may allocate a large number of cores to the application which has the best scalability while starving the others. In such a scenario, only one application would achieve a good scalability and a high QoS. To increase the overall QoS among applications, the scheduler eventually needs to be aware of the application's scalability in advance.

For example, consider the first scenario (app1: CG, app2: EP, app3: MG): In the QoS graph, CFS provides a good QoS for EP, the most CPU-intensive application in our target application set. However, the other applications starve and fail to get a good QoS. This effect is also visible in the second (SpeedUp) graph. Here, EP achieves the biggest speedup, whereas MG and CG do not achieve satisfactory performance. On the other hand, the Equal QoS policy or the Equal Partition policy provide more CPU time (physical cores) for CG and MG,

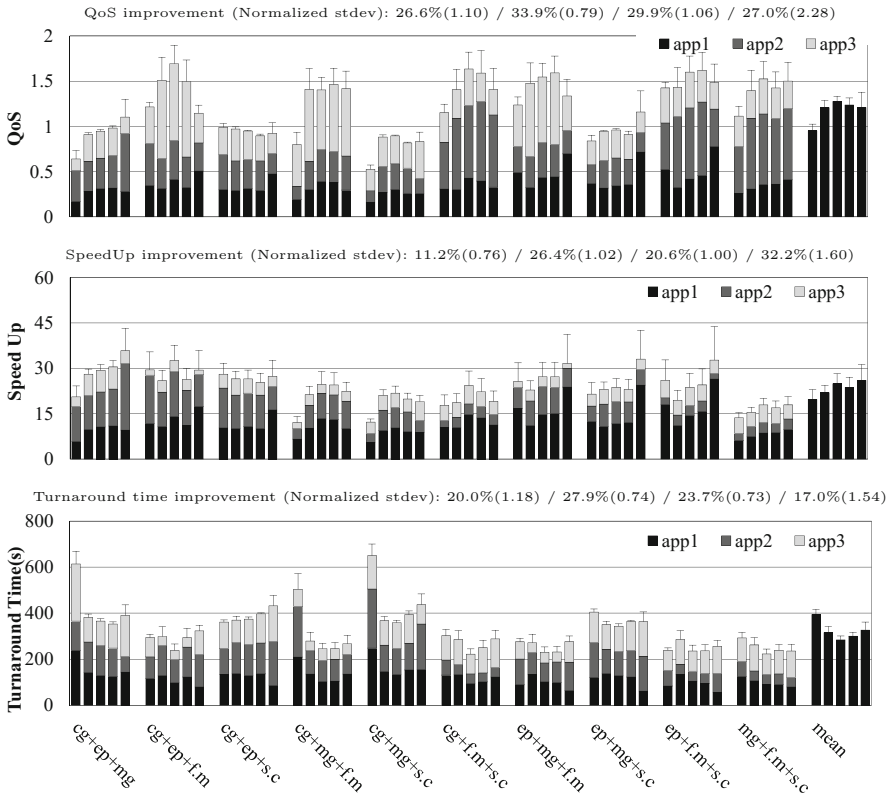



**Fig. 5.** Benchmark result: 36 core Tile-Gx36 platform. The layout of the benchmark results is equal to that in Fig. 4.

and as a consequence, the speedup of the two applications CG and MG increases. Since CG and MG are less scalable than EP, the overall speedup of the Equal-QoS policy is smaller than that of CFS. However, the overall QoS is increased and the QoS becomes more balanced. We observe that the Equal-Partition and the Equal-QoS policies manage to reduce the variance (i.e., standard deviation) of the results.

The static policy provides a static allocation of cores to each application for its entire lifetime and thus achieves a good resource isolation. As a consequence, cores that become available when one of the application finishes early cannot be reallocated to the running ones, which in turn causes a reduced overall performance. The last scheduling policy, Max-QoS shows similar characteristics as the Linux scheduler. The reason is that the Max-QoS policy determines that EP achieves the best increase in QoS.

Across the scheduling scenarios, there is no scheduling policy that always achieves the best performance; each scheduling policy shows a slightly different behavior. However, in the general case, the space-shared scheduling schemes

outperform the Linux scheduler in terms of our performance metrics. The last five bars in each graph show the average (geometric mean for QoS and speedup, arithmetic mean for turnaround time) of the summation of three application performances. We have found that in general the equal QoS policy is well suited to meet the requirements of QoS among parallel applications on our 64-core AMD Opteron machine.

The third graph in Fig. 4 shows the sum of the turnaround times of each application, i.e., lower is better. We observe that all space-shared policies outperform the CFS scheduler with Equal QoS performing best.

**36-Core Tile-Gx36 Platform.** Figure 5 shows the performance of scheduling policies on the Tile-Gx36 platform for the same experimental scenarios.

Overall, all space-shared policies outperform the standard Linux scheduler with the dynamic Equal partition policy performing best. This is in contrast to the AMD64 NUMA platform where our scheduling policies performed better.

The reason is twofold: first, the lower computational power of the Tile-Gx36 platform causes less contention, and the speedups of the different applications become similar. Second, while the overhead caused by the periodic re-allocation of the resources is not an issue on the AMD machine, the effect is noticeable on the slower Tile-Gx36 chip. Dynamic equal partition policy requires recomputations only when an application starts or finishes.

An interesting observation is that Tile-Gx36 benefits by a significantly larger performance improvement from space-partitioning compared to the results from AMD64. The reason is that each core in the AMD system is a highly-efficient super-scalar processor on which multiple threads from multiple applications can be efficiently scheduled by using advanced hardware technologies. However, the trend of many-core architectures suggests that future many-core chips will comprise simpler but many more cores in a single chip. Therefore, we believe that space sharing will be an indispensable scheduling component for future many-core resource management.

## 6 Discussion

Although the evaluations show that our scheduling policies can outperform Linux's time-shared scheduling in terms of QoS, speedup, and turnaround time for simultaneous parallel applications, there are still lots of issues to be solved and room for improvement. Here we discuss our research direction for improved space-shared scheduling based on the experience of this work.

First, for scheduling OpenMP applications, our evaluation is fixed to the default setting where every (OpenMP) parallel application is executed with the same default number of threads (i.e., $\#threads = \#cores$). However, Linux and OpenMP runtime systems may apply different resource management schemes in dependence of the number of an application's internal tasks. We also need to consider other situations where application thread counts are not equal to the

number of physical cores in the system (e.g., when an application requests a bigger number of threads than physically available cores, or vice-versa).

Second, although a space-shared scheduler requires sophisticated performance models to characterize various performance features with high accuracy, the performance model used in this paper is not accurate and captures only the scalability trend. In addition, we used multi-socket multi-core NUMA systems for the evaluation but did not consider NUMA-related performance issues in this work. The advanced performance modeling and understanding how application's performance is varied according to the active NUMA policy are our future work.

Third, an important issue of space-shared resource allocation is how to manage the degree of parallelism (thread counts). Especially for data-parallel programming models (OpenCL, Hadoop) or task-parallel runtimes (Intel TBB, Cilk) we can reclaim and reassign resources in a more flexible way. We expect that by avoiding thread overcommitment, we can achieve additional performance improvements. The mechanism for efficiently changing the parallelism from an application runtime is our research consideration. We also further consider to dynamically manage the parallelism even for the thread-based OpenMP programming model by exploiting runtime and compiler support.

Lastly, our scheduling framework aims at providing (fine-grained) resource allocation while considering a dynamically changing workload. However, known parallel application benchmarks are usually based on a monotonous workload, i.e., their behavior does not change enough for the benefits from a fine-grained resource management scheme to become apparent. We consider to use various real-world applications composed of several phases exhibiting different performance characteristics.

## 7 Conclusion

In this paper, we introduce an adaptive space-shared scheduling strategy for shared-memory parallel applications to efficiently handle dynamically changing workload characteristics. We have implemented a space-shared scheduling framework with several scheduling policies such as achieving balanced or maximal performance when simultaneously executing several OpenMP applications. Based on a simple performance model that uses the last-level cache miss rates as its main metric, our scheduler dynamically recomputes core resource allocations.

The analysis of the results on our implementations for two different many-core platforms, a 64-core AMD architecture and the Tile-Gx36, shows that, in general, space-shared scheduling schemes provide better QoS compared to the standard Linux time-shared scheduler. As a the main contribution of this work, we show that the space-shared scheduling approach has a lot of potential on current and future many-core systems.

The experiences gained from this work provide important guidelines towards better space-sharing. As part of our future work we plan to investigate other space-shared scheduling policies and to improving the performance model in order to capture various application and architecture characteristics better. Also,

in this work, we have only focused on coarse-grained scheduling issues and left the fine-grained task-to-core mapping to the Linux scheduler. To increase the performance further, dynamically managing the active thread counts (i.e., control the amount of parallelism) of applications is a logical next step of this research.

**Acknowledgments.** This work was supported, in part, by BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by the National Research Foundation (NRF) of Korea (Grant 21A20151113068), the Basic Science Research Program through NRF funded by the Ministry of Science, ICT & Future Planning (Grant NRF-2015K1A3A1A14021288), and by the Promising-Pioneering Researcher Program through Seoul National University in 2015. ICT at Seoul National University provided research facilities for this study.

## References

1. Tile-Gx36 Processor. http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx36.pdf. Accessed 28 Feb 2016
2. UG130: Architecture manual. Tilera Corp
3. AMD. AMD Opteron 6300 Series Processors. http://www.amd.com/en-us/products/server/opteron/6000/6300. Accessed 28 Feb 2016
4. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.; The landscape of parallel computing research: a view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006
5. Baumann, A., Barham, P., Dagand, P.-E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new os architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP 2009, pp. 29–44. ACM, New York (2009)
6. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The parsec benchmark suite: characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, pp. 72–81. ACM (2008)
7. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. J. Parallel Distrib. Comput. **37**(1), 55–69 (1996)
8. Breitbart, J., Weidendorfer, J., Trinitis, C.: Automatic co-scheduling based on main memory bandwidth usage. In: Proceedings of the 20th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), JSSPP 2016, May 2016
9. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: a benchmark suite for heterogeneous computing. In: IEEE International Symposium on Workload Characterization. IISWC 2009, pp. 44–54. IEEE (2009)
10. Creech, T., Kotha, A., Barua, R.: Efficient multiprogramming for multicores with scaf. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 334–345. ACM (2013)
11. Dagum, L., Enon, R.: Openmp: an industry standard API for shared-memory programming. IEEE Comput. Sci. Eng. **5**(1), 46–55 (1998)
12. Advanced Micro Devices. BIOS and kernel developer's guide (BKDG) for AMD family 15h models 00h–0fh processors (2012)

13. Emani, M.K., Wang, Z., O'Boyle, M.F.P.: Smart, adaptive mapping of parallelism in the presence of external workload. In: 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1–10. IEEE (2013)
14. Feitelson, D.G., Rudolph, L., Schwiegelshohn, U.: Parallel job scheduling—a status report. In: Feitelson, D.G., Rudolph, L., Schwiegelshohn, U. (eds.) JSSPP 2004. LNCS, vol. 3277, pp. 1–16. Springer, Heidelberg (2005). doi:10.1007/11407522_1
15. Grewe, D., Wang, Z., O'Boyle, M.F.P.: A workload-aware mapping approach for data-parallel programs. In: Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers, pp. 117–126. ACM (2011)
16. Khronos Group: The open standard for parallel programming of heterogeneous systems. https://www.khronos.org/opencl/. Accessed 28 Feb 2016
17. Lifka, D.A.: The ANL/IBM SP scheduling system. In: Feitelson, D.G., Rudolph, L. (eds.) JSSPP 1995. LNCS, vol. 949, pp. 295–303. Springer, Heidelberg (1995). doi:10.1007/3-540-60153-8_35
18. Liu, R., Klues, K., Bird, S., Hofmeyr, S., Asanović, K., Kubiatowicz, J.: Tessellation: space-time partitioning in a manycore client OS. In: Proceedings of the First USENIX Conference on Hot Topics in Parallelism, HotPar 2009, p. 10. USENIX Association, Berkeley (2009)
19. Moore, R.W., Childers, B.R.: Using utility prediction models to dynamically choose program thread counts. In: ISPASS, pp. 135–144 (2012)
20. Mu'alem, A.W., Feitelson, D.G.: Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. IEEE Trans. Parallel Distrib. Syst. **12**(6), 529–543 (2001)
21. Pabla, C.S.: Completely fair scheduler. Linux J. **2009**(184), 4 (2009)
22. Raman, A., Zaks, A., Lee, J.W., August, D.I.: Parcae: a system for flexible parallel execution. SIGPLAN Not. **47**(6), 133–144 (2012)
23. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, Inc (2007)
24. Sasaki, H., Tanimoto, T., Inoue, K., Nakamura, H.: Scalability-based manycore partitioning. In: Proceedings of the 21st International Conference on Parallel architectures and Compilation Techniques, pp. 107–116. ACM (2012)
25. Seo, S., Kim, J., Jo, G., Lee, J., Nah, J., Lee, J.: SNU NPB Suite (2011). http://aces.snu.ac.kr/software/snu-npb/. Accessed 28 Feb 2016
26. Tudor, B.M., Teo, Y.M.: A practical approach for performance analysis of shared-memory programs. In: 2011 IEEE International Parallel & Distributed Processing Symposium (IPDPS), pp. 652–663. IEEE (2011)
27. Tudor, B.M., Teo, Y.M., See, S.: Understanding off-chip memory contention of parallel programs in multicore systems. In: 2011 International Conference on Parallel Processing (ICPP), pp. 602–611. IEEE (2011)
28. Vajda, A.: Programming Many-Core Chips, 1st edn. Springer Publishing Company, Incorporated, New York (2011)
29. Wen, Y., Wang, Z., O'Boyle, M.: Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In: High Performance Computing (HiPC) (2014)
30. Wentzlaff, D., Gruenwald III, C., Beckmann, N., Modzelewski, K., Belay, A., Youseff, L., Miller, J., Agarwal, A.: An operating system for multicore and clouds: mechanisms and implementation. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 3–14. ACM (2010)