Improving Resource Utilization through Demand Aware Process Scheduling*

Brandon Nesterenko
University of Colorado Colorado Springs
bnestere@uccs.edu

Qing Yi
University of Colorado Colorado Springs
qyi@uccs.edu

Jia Rao
University of Texas Arlington
jia.rao@uta.edu

ABSTRACT

Traditional process scheduling in the operating system focuses on high CPU utilization while achieving fairness among the processes. However, this can lead to an inefficient usage of other hardware resources, e.g., the caches, which have limited capacity and is a scarce resource on most systems. This paper extends a traditional operating system scheduler to schedule processes more efficiently against hardware resources. Through the introduction of a new concept, a *progress period*, which models the variation of resource access characteristics during application execution, our scheduling extension dynamically monitors the changes in resource access behavior of each process being scheduled, tracks their collective usage of hardware resources, and schedules the processes to decrease overall system power consumption without compromising performance. Testing this scheduling system on programs on an Intel(R) Xeon(R) E5-2420 CPU with twelve kernels from the BLAS suite and five applications from the SPLASH-2 benchmark suite yielded a 48% maximum decrease in system energy consumption (average 12%), and a 1.88x maximum increase in application performance (average 1.16x).

ACM Reference Format:
Brandon Nesterenko, Qing Yi, and Jia Rao. 2018. Improving Resource Utilization through Demand Aware Process Scheduling. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225132

1 INTRODUCTION

When considering advances in modern computing systems, gains in energy efficiency are equally important as in performance. An operating system's scheduling policy can have great implications on the overall energy efficiency and performance of a system. In particular, when scheduling multiple processes together, their concurrent resource accesses may cause interferences with one another. This can be detrimental to both overall system energy efficiency and the performance of the individual processes.

*This work is funded by NSF through awards CCF-1421443 and CCF-1261584

ICPP 2018, August 13–16, 2018, Eugene, OR, USA

ACM ISBN 978-1-4503-6510-9/18/08…$15.00
https://doi.org/10.1145/3225058.3225132

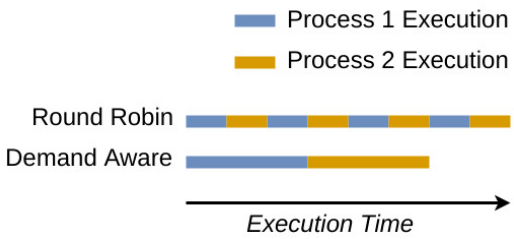

Figure 1: Process execution comparison of scheduling policies

This paper aims to enhance operating system scheduling to be aware of the resource demands of the scheduled processes and thereby to achieve more efficient uses of hardware resources. The end result is a more energy efficient system with speedups of the applications being run [5, 16, 25]. For the scope of this paper, the main resource we consider is the shared memory system, particularly the cache hierarchy that is shared among different CPU cores. Since there is only limited storage capacity in the caches, when multiple processes are scheduled to share them, the data from different processes have to compete for the shared storage and can overwrite each other's data as the result. Most schedulers do not sufficiently consider this behavior, as modern scheduling policies typically prioritize a minimal wait time between processes (fairness), maximum utilization of all available CPUs, and short job completion. This is because schedulers have a restricted a priori knowledge; they are usually only aware of the active task set, and must make decisions based on the presence of real-time events [20, 23]. Figure 1 illustrates this problem with two processes scheduled under a round robin policy; the processes spend extra time and energy by having to reload their data from memory into cache.

We propose to enhance the programming interface between software applications and the operating system, so that the OS scheduler can sufficiently understand the resource access behavior of its processes. The scheduler can then adjust its scheduling policy to prioritize lower system energy consumption and higher application performance. The *demand aware* line in figure 1 illustrates how processes that compete for shared caches are scheduled using our extended scheduler, which eliminates collisions among competing processes by scheduling their conflicting runtime durations one after another instead of concurrently in a round-robin fashion. To obtain the knowledge of which processes may interfere with each other immediately before the interferences occur, our scheduler establishes additional communication with processes through a user-level API. Applications call into this API to provide their just-in-time resource demands to the operating system. Our

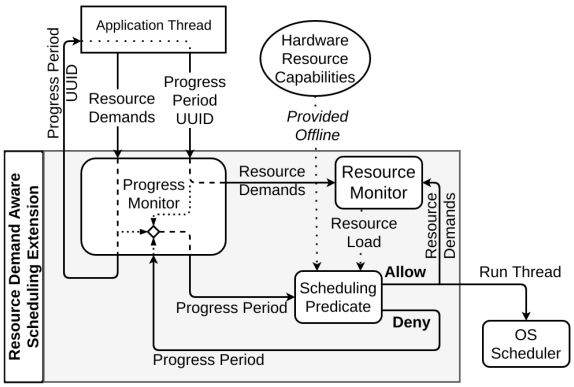

Figure 2: Workflow when starting a progress period in our scheduling environment

system extension then uses these demands to make informed scheduling decisions to decrease system power consumption without compromising the performance of the processes.

The methodology of our scheduling approach relies on two key techniques: (1) decomposing applications into *progress periods* that model their resource demands for various durations of execution, and (2) feeding the progress periods into the os scheduler to make real-time decisions that maximize resource efficiency by the running processes. We assume the responsibility of decomposing an application execution into progress periods can be taken over by existing application performance analysis tools, e.g., HPCToolKit [3] and Intel PIN [13], which profile the execution of an application to discover various aspects of its runtime behavior. While we currently manually modify applications to validate the benefit of such an approach, we additionally study the feasibility of automating this process by using existing profile-driven application analysis tools.

Figure 2 shows the workflow of our scheduling environment, which consists of three components. (1) The *progress monitor*, which is responsible for communicating with applications, maintaining all progress period related information, and attempting to schedule any waiting threads previously blocked due to resource constraints; (2) the *resource monitor*, which maintains the active load of the last level cache and receives resource demands as input from the other components to keep the load values current; and (3) the *scheduling predicate*, which decides whether a thread should be allowed to run at the beginning of each new resource access behavior. The scheduling process is started when a thread provides its resource demands, thereby allowing our scheduler to internally create a progress period and return a number uniquely identifying it back to the thread. When a thread finishes its resource access behavior, the unique identifier is passed back into the progress monitor to signal the ending of the progress period. The scheduling predicate bases its decisions off of the system's cache capabilities and the current load of the last-level cache, which is provided by the resource monitor, to determine whether each progress period passed in by the progress monitor should be scheduled.

The main technical contributions of this paper are as follows:

- We introduce a new concept, a *progress period*, to model the variation of resource demands as an application execution goes through different phases. We provide an interface for applications to identify each progress period by tagging its beginning and ending of a sequence of instructions and then defining resource usage characteristics of these instructions. We study the feasibility of using existing performance analysis tools to automatically discover the progress periods of an application and the resource access behavior of each period.
- We propose a new method to predict resource interference between co-scheduled processes by monitoring the transitions into and out of progress periods at runtime. We extend the traditional operating system scheduler to track the varying resource demands of active processes, and present a demand aware scheduling system that allows the last level cache to be utilized more efficiently than existing approaches.
- We use our scheduling extension to schedule 12 Basic Linear Algebra Subprogram (BLAS) routines and five applications from the SPLASH 2 benchmark suite. We were able to obtain a 48% maximum decrease in system energy consumption (average 12%), and a 1.88x maximum increase in application performance (average 1.16x).

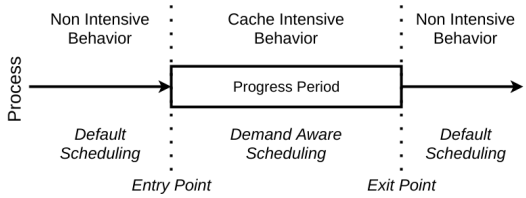

Figure 3: Time lapse for a process with a single progress period being scheduled in our environment.

The remainder of this paper is organized as follows. Section 2 presents our API for applications to communicate characteristics of their progress periods. Section 3 presents our scheduling extension. Section 4 presents experimental results. Section 5 discusses related work. Finally, section 6 concludes.

2 IDENTIFYING PROGRESS PERIODS

Before a process may utilize the features offered by our scheduling environment, its resource demands must be identified within an application in the form of *progress periods*. A single progress period describes a duration of an application execution where its resource demand for data storage remains roughly constant. The resource access behavior of each process may then be characterized by a sequence of progress periods. A progress period is bounded by a sequence of instructions that when executed, notifies the scheduler that a new resource behavior is beginning or ending.

The composition of a progress period therefore consists of: (1) a pair of instructions marking its beginning and ending, and (2) a quantification of its resource usage.

2.1 Progress Period Boundaries

The beginning and ending of a progress period are defined by its entry and exit points within an application's binary executable form. Together, they signal the start and stop of a new resource access behavior to the scheduler. All instructions executed in between these points are required to conform to the resource access behavior defined by the progress period. Figure 3 shows the placements of the entry and exit points within a process's execution, and how the scheduling policy changes between boundaries.

```
1: int main(int argc, char **argv) {
2:  double *A, *B, *C;
3:  int n = 512; // matrix width and heights
4:  double pp_id;
5:  initializeMatrices(n,A,B,C);
6:  pp_id = pp_begin(RESOURCE_LLC, MB(6.3), REUSE_HIGH);
7:  DGEMM(n,A,B,C);
8:  pp_end(pp_id);
9:  displayResult();
11:return 1;
12: }
```

Figure 4: C code sample consisting of one progress period, the call to DGEMM.

2.2 Quantifying Resource Usage

In addition to the execution entry and exit points, a progress period must also specify information about the resource access behavior it will exhibit while executed. This includes the hardware resource being targeted, and how that resource will be used. To model how a progress period accesses the caches of a CPU, we use two integers: a working set size, which represents the total amount of memory to be consumed by the progress period; and a relative temporal locality factor, which specifies how heavily the data in the working set size will be reused. A complete progress period then consists of the following:

(1) A set of instructions marking the execution entry point
(2) A set of instructions marking the execution exit point
(3) The targeted resource
(4) The working set size
(5) The relative amount of data reuse.

2.3 Application Programming Interface

In order to schedule processes against the hardware efficiently, applications need the ability to communicate their resource demands to the operating system. Currently, progress periods are defined through function calls within an application. The first function call starts the progress period, and requires the targeted resource, a working set size, and a data reuse factor; and the second function call ends it.

Figure 4 presents a code snippet that illustrates the function calls beginning and ending a progress period using our API. The code calls the DGEMM kernel, which performs a general matrix-matrix multiplication. When line 6 is executed, it signals to the scheduler that a new progress period has begun, requires 6.3 megabytes of space in the last level cache, and it will frequently reuse the stored data. The function returns a value that uniquely identifies the progress period. The DGEMM kernel at line 7 will then be run once the resource demands requested can be fulfilled by the system. Then, as illustrated in line 8, once the kernel has finished, the application will signal the end of the progress period using the unique identifier obtained in line 6.

2.4 Profiling Application Behavior

Our programmatic approach of defining progress periods allows an application to describe its resource access behaviors in a flexible way. We expect that a developer can automatically obtain such descriptions by using a variety of existing program execution analysis tools including WSS [12], Dyninst ParseAPI[21], and Linux Perf [1]. The actual insertions of the API calls inside an application can also be potentially automated through a compiler or a binary translator.

Although both tasks belong to our future work, to study the feasibility of automatically modifying existing applications to dynamically communicate their resource demands at runtime, we have developed a preliminary profiler, which uses Intel PIN [13] to collect the runtime virtual memory addresses from each load/store instruction within each fixed-size sampling window of instructions. An array is used to keep track of the number of times each unique address is accessed. The array is reset to be empty at the beginning of each sampling window. Its new size at the end of the window is then calculated as the memory footprint of the window. The working set size of the window is calculated as the number of entries in the array that are accessed at least a pre-configured number of times, and the average number of times each entry is accessed is calculated as its reuse ratio.

To identify progress periods, our profiler tries to capture all repetitions of the above data access behaviors at a given granularity. For example, to identify all loops with $\leq x$ instructions in its body and with at least y instructions throughout all iterations, we first sample the runtime statistics of the entire application execution using a fixed-size window of x instructions. The overall application runtime is therefore decomposed into $p_0, p_1, ..., p_n$ consecutive runtime periods, each period p_i summarizing the statistics the $i * x$th, $i * x + 1$th, ..., and $(i + 1) * x - 1$th instructions of the program execution. Then for each y/x consecutive execution periods, say $p_i, p_{i+1}, ..., p_{i+y-1}$, if their runtime statistics are sufficiently similar based on a predetermined threshold, these execution periods can be determined to be the beginning of a significant repetition. The loop is then extended by consider p_{i+y}, p_{i+y+1}, etc., until a period p_j is reached that has significantly different behavior. The periods p_i and p_{j-1} are then determined to be the beginning and ending of the progress period identified. The whole process starts by examining the y/x consecutive periods starting at p_1. if $p_1, ..., p_j$ is identified to be a progress period, the next y/x periods starting at p_{j+1} are examined; otherwise, the next y/x periods starting at p_2 are examined. The whole process repeats until the last period p_n has been examined.

To correlate the detected runtime information with the source code of an application, we sample the linear memory addresses of the JMP instructions retired within each window, and use Dyninst ParseAPI [21] to locate these JMPs within the loop nest structure of the binary. The outermost loop that contains the identified progress period is then used as the beginning and ending of the period. The resource demands for each progress period are set by averaging the metrics from all windows that make up the progress period.

We have used our profiler to identify progress periods for all applications we studied, by manually experimenting with different granularities of window sizes and then manually modifying the application to communicate the relevant information to the operating system. The accuracy of the identified information is validated by manually examining the applications being studied and by the effectiveness of using them to guide our extended os scheduler.

3 RESOURCE DEMAND AWARE SCHEDULING

By utilizing the resource access information that progress periods provide within user applications, the traditional process scheduling

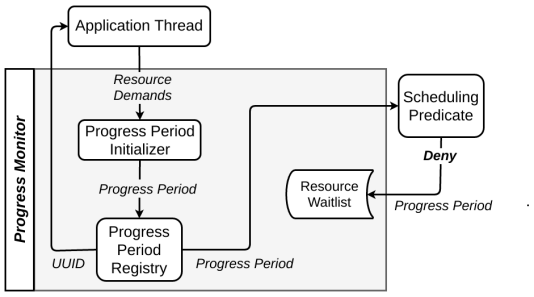

Figure 5: Workflow that shows the progress monitor's sub components when an application begins a progress period

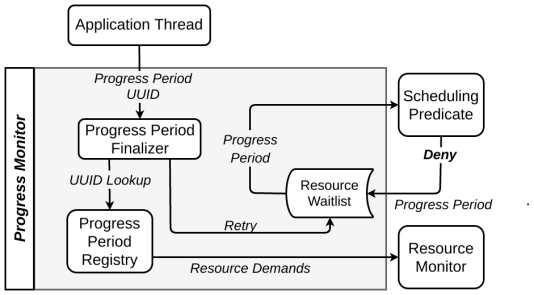

Figure 6: Workflow that shows the progress monitor's sub components when an application ends a progress period

of an operating system can be extended to better manage resource utilization. We have developed such an extension, which exists on top of the underlying Linux 4.6.0 default scheduler, and decides which processes should be run by pausing and resuming processes only at the beginnings and endings of progress periods. In particular, by keeping track of the resource demands of each progress period, our scheduler avoids co-scheduling processes that put the hardware over its maximum capacity, thereby reducing interferences among processes that intensively compete for the same resource. To pause and resume threads, our scheduling extension utilizes a wait queue with wake events inside the Linux kernel. Our system ignores processes that have not provided progress period information, and schedules them directly on the operating system.

3.1 Tracking Progress Periods

Shown in figure 2, our scheduler includes a *progress monitor* component to ensure that the resource usage information in our scheduling environment is always accurate. This component tracks the API calls that indicate the beginning and ending of each progress period. It then uses these transitions to signal both the resource monitor, to maintain the real-time resource usage values of all co-running processes, and the scheduling predicate, to indicate changes in process behavior so that previously paused processes may be resumed as appropriate.

Figures 5 and 6 show the workflows of the progress monitor for scheduling processes when a progress period is entered or exited. The progress monitor stores all active progress period information in a registry, so the resource usage footprint of each progress period can be removed from our environment after the period completes. When a process enters a new progress period, the initial scheduling decision for the process either allows it to continue running, if there is sufficient room for the additional resource demand, or denies its resource demands and requires the process to pause. Processes that are paused are placed on a resource waitlist so they may be rescheduled later when another progress period completes and releases sufficient resources for it to resume execution. As each progress period completes, it is removed from the registry, its resource demands are passed to the resource manager to update the real-time hardware usage tracker, and the process returns to the default scheduling policy provided by the underlying OS until another progress period starts.

3.2 Tracking Hardware Resource Usage

To determine how heavily the running processes are using a system's hardware, our *resource manager* in figure 2 maintains a real-time estimation by saving the resource demands of all active progress periods. A table is used to keep track of the current load level for the resources, where an entry is allocated to each resource to save its current usage level. The resource manager keeps the usage estimation up-to-date any time a process enters or completes a progress period, which signals a different resource access behavior. By comparing the real-time usage estimation with the maximum capacity of the resource, it calculates the amount of free space available to avoid oversubscription.

3.3 Preserving Resource Efficiency

Our *scheduling predicate* component in figure 2 is responsible for making the decision to run or pause a process as it enters or exits a progress period. The decision is based on whether the new period will cause interferences with other running processes, which is in turn determined by examining four parameters: the available capacities of hardware resources, their current usage levels, the resource demand of the new progress period being entered, and a reconfigurable scheduling policy that dictates the limits of each hardware resource. The policy allows users to specify that a certain amount of oversubscription is allowed to provide more concurrency among the applications.

Algorithm 1 details how the scheduling predicate decides when to pause or run a thread. The algorithm starts by calculating the amount of unused space currently available by subtracting the total capacity of the resource with the current load level. Next, it calculates how much space would be left if a new progress period were to be scheduled, and uses this value as input to a predefined policy to see if the period should be allowed: if yes, the new resource demand is passed to the resource manager to increase the active load estimation, and the process is passed on to the underlying operating system to be scheduled; if not, the process is passed back to the progress monitor to be placed on the waitlist.

Our scheduling policy currently uses a single parameter per resource to control oversubscription. Two policy configurations are supported by default.

(1) *RDA: Strict* - The sole focus of this policy is to maximize hardware resource efficiency. It denies any process from running if the additional resource demand will put a hardware resource above maximum capacity. This policy is intended to result in the least amount of energy consumed. However, it may have negative performance implications.

Algorithm 1 Scheduling decision algorithm

```
function TrySchedule(pp, resource)
    remaining ← resource.capacity − resource.usage
    outcome ← remaining − pp.demand
    runnable ← apply_policy(outcome, resource)
    if runnable then
        increment_load(pp.demand)
        schedule(get_process(pp))
    else
        waitlist(pp)
    end if
end function
```

(2) *RDA: Compromise* - this policy aims to balance between a less efficient usage of hardware resources with increased concurrency among the processes. It allows a process to run as long as the addition of its resource demand keeps the current usage level within x times of the total resource's capacity, where x is a pre-configured oversubscription factor. This policy is intended to result in moderate savings in energy consumption while simultaneously boosting the performance of applications. In our study, we have configured the oversubscription factor to be 2, which is shown to be effective in attaining the best balance between energy efficiency and performance.

3.4 Generality and Limitations

By decomposing an application execution into a sequence of progress period intervals, our work allows co-running processes to be scheduled at a granularity dictated by the resource demands of the application and changes in its resource consumption behavior. As the result, concurrency can be attained together with better energy efficiency.

Our current modeling of the resource demands works best when a group of progress periods meet the following constraints: (1) individually, their working sets fit within the capacity of the available caches, (2) collectively, the size of their working sets exceed the total capacity of the shared cache, (3) there is a moderate to high amount of reuse to be exploited within their data accesses, (4) the working set and data reuses stay relatively constant through the duration of each progress period, and (5) there is no blocking synchronization within any of the progress periods. The last restriction ensures that each scheduled progress period will not go to sleep and invalidate the real-time usage estimation.

Our model currently does not yet handle synchronization and cache partitioning, which belong to our future work. A problem with not handling synchronization is that if a group of threads synchronize within a progress period (e.g., using barriers), and one thread is paused due to resource constraints, it is possible that all threads will halt indefinitely. To resolve this, we currently do not allow progress periods to contain blocking synchronizations, so that any durations of application execution that contain synchronizations are scheduled using the existing system's default scheduling policy.

CPU		Intel(R) Xeon(R) CPU E5-2420 1.90 GHz, 12 Cores
Cache	L1-Data	32 KBytes
	L1-Instruction	32 KBytes
	L2-Private	256 KBytes
	L3-Shared	15360 KBytes
Main Memory		16 GiB
Operating System		CentOS 6.6, Linux 4.6.0

Table 1: Machine configuration

Workload	# Proc	# Threads / Proc	Work-set sizes(MB)	Data Reuses
BLAS-1 (daxpy, dcopy, dscal, dswap)	96	1	.6	Low
BLAS-2 (dgemvN, dgemvT, dtrmv, dtrsv)	96	1	.6	med
BLAS-3 (dgemm, dsyrk, dtrmm(ru), dtrsm(ru)	96	1	1.6, 2.4, 2.4, 3.2	High
Water_sp	12	2	1.6, 1.3, 1.3, 1.6	low, low, low, low
Water_nsq	12	2	3.6, 3.6, 3.7	high, high, high
Ocean_cp	48	2	2.1, 0.76, 1.5, 0.59	high, med, high, med
Raytrace	48	4	5.1, 5.2	high, high
Volrend	48	4	1.8, 1.7	high, high

Table 2: Workloads used to test our scheduling extension: each workload detailing its number of processes, threads per process, working set sizes, and level of reuse per working set

Scheduling processes by progress periods may also interfere with task-pool based programming models, where the underlying parallel programming support uses its own scheduler to distribute tasks among pre-allocated threads. To resolve this, if one of these threads enters a progress period and is unable to run, our scheduler temporarily disables the whole thread pool until there is sufficient resources for all of them. Our future work will investigate better taking advantages of the blocked threads.

4 EXPERIMENTAL RESULTS

We have implemented our scheduling extension by extending the Linux 4.6.0 kernel from a CentOS 6.6 install on an Intel E5-2420 processor with 12 cores. The configuration of the machine is shown in table 1.

This section aims to evaluate our prototype OS scheduling extension to quantify its impact over the overall system energy efficiency as well as the individual performance of the applications. Since our extension requires the applications to track their resource demands, which introduces runtime overhead in the form of extra instructions and context switching, we analyze the overhead induced while varying the granularity of progress tracking. Finally, we study the feasibility of using existing performance analysis tools to automatically extract the progress periods of existing applications by using our preliminary profiler to study the potential of modeling the resource demand patterns of applications as functions of their programed behavior in spite of varying input data.

4.1 Experimental Design

We have evaluated our scheduling extension using eight different workloads, shown in Table 2, which are composed by selecting 12 BLAS (basic linear algebra subprograms) [29] and five SPLASH-2 application benchmarks [27]. The 12 BLAS kernels are separated into three groups, each including four BLAS kernels categorized as level-1, level-2, and level-3 respectively. The BLAS-1 kernels all perform vector-vector operations with a minimal level of cache reuse, the BLAS-2 kernels all perform matrix-vector operations with a

medium level of cache reuse, and the BLAS-3 kernels all perform matrix-matrix operations with a high level of cache reuse. Each BLAS kernel as a whole is considered as a single progress period and has been optimized with loop blocking so that individually its working set size fits within the last-level cache of the hardware. Each level of BLAS kernels is grouped into a distinct workload to categorize system behavior under different level of cache reuses. Each of the five SPLASH-2 application benchmarks is broken into multiple progress periods, with an input size that restricts the working set sizes of all progress periods to individually fit within the last level cache of the hardware. Each SPLASH-2 application is in itself treated as a single workload, to quantify how our scheduling extension may impact the performance of individual multi-threaded applications.

We used our preliminary profiler described in Section 2.4 to decompose each SPLASH-2 application into distinct progress periods and to identify the working set size and reuse ratio of each progress period. The reuse ratios are categorized into three levels: low, medium, and high. For each benchmark, an input data set is identified so that the working set size of all progress periods can fit in the last-level cache of the hardware. Then, the appropriate working set sizes and data reuse levels are used to modify the source code of the benchmark to include the progress period API calls.

We have implemented our scheduling extension under two alternative configurations: *RDA: Strict*, which prioritizes hardware utilization efficiency over allowed concurrencies among the processes, and *RDA: Compromise*, which attempts to balance between running more processes concurrently and a less efficient cache utilization. These configurations are described in more detail in Section 3.3. Both scheduling configurations are then compared with using only the default OS scheduling policy to schedule the eight workloads in Table 2.

We compiled all benchmarks using gcc 6.2.0 with the -O3 optimization flag. The energy efficiency and performance of all workloads are measured using perf, which contains a suit of runtime analysis tools on Linux [1]. The following metrics are used to measure the energy efficiency and performance of the overall system when applications are scheduled concurrently on the underlying hardware, with and without our scheduling extension.

(1) The overall amount of energy (in Joules) consumed by the system (CPU + cache + DRAM) and by the DRAM only, measured using Intel's RAPL (Running Average Power Limit) interface, which provides energy consumption information through power metering [14].
(2) The FLOPS (floating point operations per second) of the CPU during the execution of all applications, measured using hardware counters.
(3) The FLOPS executed per Watt. Attained by dividing the total number of floating point operations by the overall amount of energy consumed by the system, this metric represents how efficiently the CPU executes instructions in relation to the amount of energy consumed.

Each measurement was repeated four times, and the values we use are the average of each measurement. The average standard deviation between all runs is 2%.

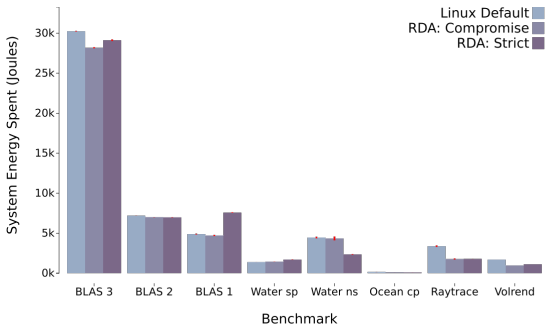

Figure 7: Comparing the resource demand aware scheduling policies against the Linux default scheduling policy in terms of energy, in Joules, consumed by the system (CPU + cache + DRAM) when scheduling the eight workloads under different policies.

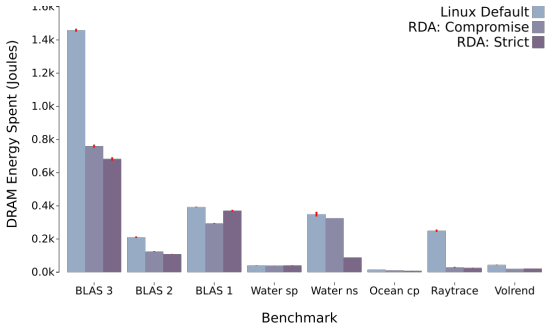

Figure 8: Energy (Joules) consumed only by DRAM when scheduling the eight workloads under different policies

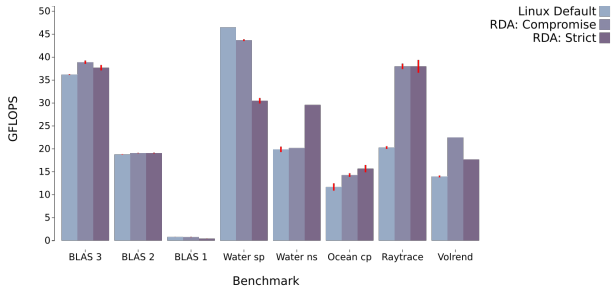

Figure 9: Performance measured in GFLOPS for each workload run under the different scheduling policies

4.2 Performance and Energy Implications

Figures 7 and 8 compare the overall energy, in Joules, consumed by the system (CPU + cache + RAM) and just the DRAM when using different OS scheduling policies for each of the workloads in Table 2. Figure 9 shows the respectively attained performance of each workload, in terms of the average GFLOPS attained by the CPU. Figure 10 compares the GFLOPS per Watt attained for the overall system when using different scheduling extensions.

From these figures, our resource-demand aware scheduling extension can significantly improve both the energy efficiency and the performance of the system, when used to schedule workloads with a medium to high level of data reuses. These workloads are represented by the BLAS-2, BLAS-3, Water_nsquaread, Ocean_cp,

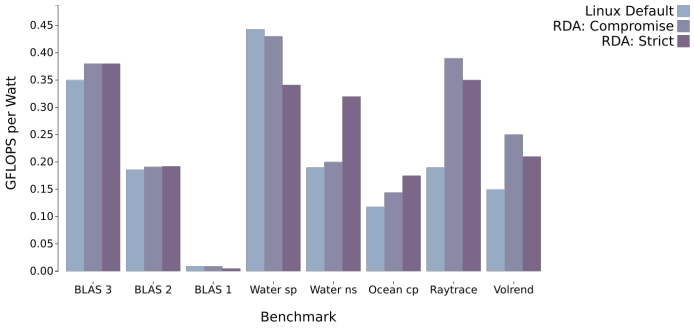

Figure 10: GFLOPS per Watt of system (CPU + caches + DRAM) comparison of scheduling policies on the eight workloads

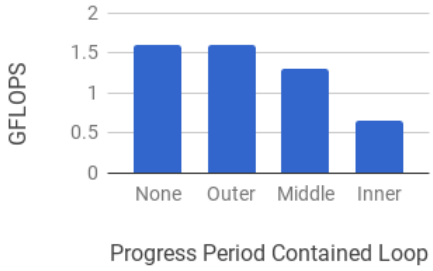

Figure 11: Execution time taken by dgemm with no progress periods, and progress periods placed containing the outer, middle, and inner loop.

Raytrace, and Volrend benchmarks. In particular, when scheduling Raytrace with the strict policy, we attained a maximum speedup of 1.88x and 47% decrease in overall energy consumed, when compared to the Linux default scheduler. BLAS-2 lead to the smallest improvement with both demand aware scheduling policies resulting in a maximum performance increase of 1.02x and system energy consumption decrease of 97%. In terms of overall energy efficiency, the RDA strict configuration attained the best results for Water_nsq and Ocean_cp, with increases over the Linux default policy by 1.68x and 1.36x; and the compromise configuration attained the best for Raytrace and Volrend, with increases of 2.05x and 1.67x; the policies tied for the BLAS-2 and BLAS-3 workloads.

However, when using our scheduling extension for workloads that have low data reuses, represented by the BLAS-1 and water_spatial benchmarks, it had attained results inferior to the Linux default scheduling policy. In particular, for water_spatial, our scheduling extensions had decreased performance by 6% while increasing system energy consumption by 1.04x. The performance degradation from using our scheduling extension is due to the decreases in system-level concurrency, which cannot be compensated by better cache utilization, shown by the almost identical level of energy consumed by DRAM for these two workloads.

By analyzing the amount of energy consumed by DRAM, we show that the strict policy almost always resulted in better LLC utilization than the compromise configuration. However, the compromise policy was still able to attain a better overall system energy efficiency than the strict policy. We determine which policy results in a greater system energy efficiency by analyzing the LLC efficiency. In the cases where the workload measurements of DRAM energy consumption by our RDA configurations are very similar, e.g., BLAS-3, BLAS-2, Raytrace, and Volrend, the additional LLC utilization granted by the strict configuration is inconsequential for the drop in concurrency, and the overall system energy efficiency is higher when scheduled under the compromise policy. For example, the DRAM consumption values for Volrend differ only by 9%, however; the compromise policy attains a 21% speedup when compared to the strict configuration; leading to the compromise policy achieving an overall system energy efficiency increase of 16% over the strict configuration. In cases where the strict policy continues to draw a significant drop in DRAM energy consumption, when compared to the compromise configuration, as shown with workloads water_nsquared and ocean_cp, the additional concurrency from the compromise policy highly degrades LLC efficiency, thereby resulting in the strict RDA scheduling policy being superior to the compromise configuration. For example, in water_nsquared, the compromise policy attains a 7% DRAM energy consumption decrease over the Linux default policy, and the strict configuration continues to attain another 73% DRAM energy consumption decrease over the compromise policy; thereby resulting in the performance of the workload to increase by 1.47x when scheduled via the strict policy in comparison to the compromise configuration. We attribute the difference in results between the two demand aware policies to be application dependent, in that a more precise description for data reuse in each progress period would be needed to determine which policy would work best.

Overall, scheduling processes by their resource demands offers potentials to improve the energy efficiency of the system. However, different scheduling configurations need to be combined for the overall approach to be beneficial. In the general case, the energy efficiency (GFLOPS per Watt) is tightly coupled to the direct performance (GFLOPS). Workloads with higher levels of data reuses will benefit from such a scheduling extension, with both enhanced performance and overall energy efficiency. On the other hand, workloads with low levels of data reuse should be scheduled by using the existing default scheduling policy of the operating system, to allow for a high level of concurrency. The strict policy tended to lead towards bigger drops in overyall system energy consumption, e.g., the maximum decrease was 48%, and was attained from running water_nsquared. The compromise policy tended to give bigger increases in energy efficiency, e.g. the maximum increase in energy efficiency was 2.05x, and was attained from running Raytrace

4.3 Progress Tracking Overhead

To quantify the overhead of tracking progress periods at different granularities, we further broke down a BLAS-3 kernel, *dgemm* to include additional finer-grained progress periods. This kernel contains three nested loops, each evaluated 512 iterations. We experimented with three strategies to decompose it into progress periods: at the outermost loop level, where the whole computation is a single progress period; at the middle loop level, where the whole computation is broken down to 512 equal-sized progress periods; and at the innermost loop level, where the computation is broken down to 262,144 equal-sized progress periods.

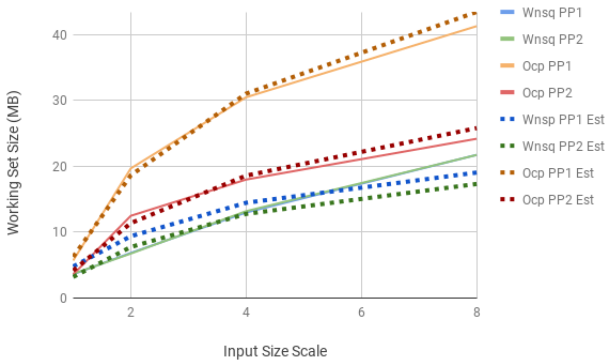

Figure 12: Here we show how the actual working set size of two progress period's from the water_nsquared and ocean_cp applications increase with respect to the input size, and our prediction function results. The input size scales from an original input size, 1x, to be 2x, 4x, and 8x the size.

Figure 11 shows the performance impact in GFLOPS when evaluating the modified *dgemm* kernel with different granularities of progress periods. For this test, a single instance of the kernel was the only active user process run on the host machine with the strict policy active. No runtime overhead is observed when tracking only a single progress period, a 19% performance overhead is incurred when tracking 512 iterations of the middle level loop, and a 59% performance overhead is observed when tracking 512^2 progress periods. Overall, we expect to use a constant to bound the overall number of progress periods used to break down each application, as illustrated in Table 2. This means that for any loops contained in an application, a single progress period should be used at the outermost loop level, to minimize runtime overhead of tracking progress periods.

4.4 Automatically Extracting Progress Periods

To study the feasibility of requiring each application to dynamically report their progress periods to the operating system, we show that a profiler can be built to potentially automatically extract the relevant information, and such information can be inserted into the application in a relatively straightforward fashion that can be potentially automated by a source-level compiler or a binary translator. We have used our preliminary profiler described in Section 2.4 to extract the working set size, reuse ratio, and beginning/ending of each progress period shown in Table 2, demonstrating this can be done in a semi-automatic fashion. The main component that needed developer intervention is actually inserting the API calls into the application, which requires the developer to convert the runtime statistics collected via profiling into numbers that are parameterized by the varying input data sets of the application. In the following, we study the potential of automating this process as well.

Using the water_nsquared and ocean_cp applications from table 2 as examples, we profiled each application using four different input data sizes: 1x, 2x, 4x, and 8x. The 1x input size values the simulated input sizes provided by default from SPLASH-2, where

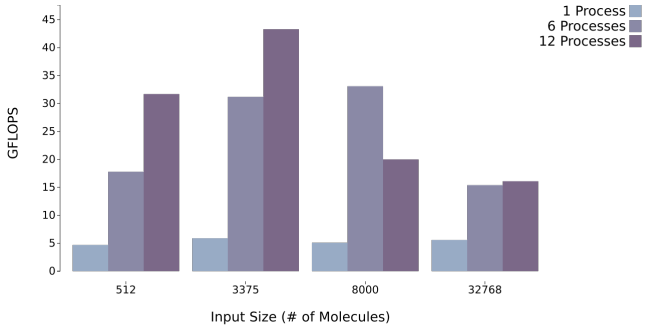

Figure 13: The amount of slowdown for the largest progress period in water_nsquared due to LLC interference from increased data size and concurrent processes running.

their corresponding values are 8000 molecules and 514 cells, for water_nsquared and ocean_cp, respectively. These values are scaled by 2x, 4x, and 8x and slightly adjusted to fit within the runtime restrictions of the program. For water_nsquared and ocean_cp, the are 8000, 15625, 32768, and 64000 molecules; and 514, 1026, 2050, and 4098 cells, respectively.

The top two progress periods are selected for each of the applications, and figure 12 compares the reported working set sizes for the four progress periods (denoted Wnsq PP1 and Wnsq PP2 for water_nsquared; and Ocp PP1 and Ocp PP2 for ocean_cp) when using different input data sets. It can be seen that the working set size does not grow linearly with respect to the input size, but rather in the shape of a logarithmic curve. Therefore, to predict the change in working set size, we run a logarithmic regression over the first three inputs from each progress period to generate four prediction functions (denoted with the *Est* suffix in the figure). To test our prediction functions, we calculate the estimated working set size using the fourth input size. For PP1 and PP2 in water_nsquared, the prediction accuracy is 92% and 80%. For PP1 and PP2 in ocean_cp, the prediction accuracy is 95% and 94%.

To study the performance impact that LLC interference can have on concurrency, we analyze the performance of the longest progress period from water_nsquared when run under varying input sizes and number of total concurrent instances (1, 6, or 12), a shown in figure 13. We set the maximum number of instances to the number of physical cores on our test machine, 12, because the time to complete 1 instance of the process would be the same as the time to complete 12 instances, assuming no LLC interference. For input sizes 512 and 3375, the LLC is not utilized very extensively, and the performance scales fairly well for the inputs. For input size 8000, the LLC is utilized heavily, so the performance scales well from 1 to 6 concurrent processes; however, the performance significantly drops from 6 to 12 concurrent processes. In particular, the GFLOPS drops from 33 to 20, meaning the LLC can hold all data from 6 processes, but not twelve. This indicates that when running twelve or more instances of water_nsquared with an input size of 8000 molecules, co-scheduling the processes in groups of six will attain a higher performance than when running all instances together. For input size 32768, the performance scales from using 1 process to 6; however, from 6 processes to 12, the performance remains unchanged. This is because at 6 processes, the performance

becomes memory bound, and achieves similar performance to that of 12 processes with the 8000 molecule input.

5 RELATED WORK

Specialized techniques for scheduling blocks of code to use hardware more efficiently have been explored for quite some time. In 1988, Colwell, Nix, O'Donnell, Papworth, and Rodman introduced a static technique using compilers to identify blocks of code that were unrelated and may be scheduled concurrently using a VLIW architecture [8]. Govindaraju, Ho, and Sankaralingam developed a more hardware-oriented approach in 2011 that similarly breaks applications into phases identified by a GCC extension, and connects them with a circuit-switched network [11]. Zhu and Reddi implemented a scheduling technique to schedule webpages using the ideal core and frequency configuration to obtain energy savings on mobile devices [31]. In an experimental energy management survey performed by Kambadur and Kim in 2014, they discover that effective parallelization can lead to better energy savings compared to Linux's frequency tuning algorithms[16].

Tillenius et al. [25] developed an approach based on the OmpSs programming model to provide resource-aware scheduling within a task parallel programming environment, which allows developers to include a resource attribute within an omp task definition and manually set the quantity of how a resource will be used. Their scheduler uses these tags to schedule tasks on specific cores, and not oversubscribe resources at run-time. Radojkovic et al. [22] created a statistical approach to task scheduling that explores several thousand random task assignments, and analyzes whether the best performing assignment in the random sample performs close to the optimal solution.

Kihm, Settle, Janiszewski, and Connors [17] developed a scheduling extension based on predicting inter-thread kickouts (ITKO) to co-schedule threads that are less likely to evict each others data. Their strategy is initially profiling an application with Valgrind every 50 million instructions and writing a bit to a file indicating whether or not that interval exceeded an ITKO threshold. They pass this file to the OS, which monitors the number of instructions each thread has executed, and compares it to the file, and schedules jobs based on whether or not the threshold was reached. Our approach is similar to this work; however, maps the behavior to a static code location to track progress, allowing our scheduler to be less reliant on input sensitivity.

Additional scheduling techniques have been explored that rely on sampling values at run-time to see how the active processes utilize the system's resources. For example, Bhadauria and Mckee developed a performance monitoring scheduler that continuously samples hardware counters to dynamically change the thread allocation on cores to obtain better power usage and performance in 2010 [5]. Also, El-Moursy et al. [10] discovered that using the number of ready and in-flight instructions as dynamic counters provides a more consistent result for making co-scheduling decisions. We believe that using real-time hardware counters to determine current resource usage, in combination with demand aware scheduling, would be able to schedule processes much more efficiently, as well have a better prediction basis for scheduling new processes, than either of these ideas alone; and is therefore a subject to explore in later work.

Job scheduling to reducing interference has been explored both at the single-node and at the multi-node levels. Bhimani et al. [6] developed a docker controller for scheduling containers for different types of applications to avoid workload interference and decrease overall execution time. Yang et al. [28] proposed a system to control the number of concurrent jobs in a YARN MapReduce environment to avoid resource waiting deadlock and improve system performance. Our work is currently developed at the single-node level but can be extended to multiple nodes as part of our future work.

Alterantive approaches, e.g., through software scheduling and hardware cache partitioning, have also been exploited to improve hardware efficiency. A study performed by Lin et al. [18] showed that software cache partitioning, particularly with dynamic policies, can have a significant performance impact on real systems. Dhodapkar and Smith[9] presented work to give working sets a signature to be able to detect changes in working sets at run time, thereby allowing saved configurations to be loaded to allow a process to more efficiently use resources, e.g. cache. Jaleel, Najaf-abadi, Subramaniam, Steely, and Emer developed a system that would do two things: (1) use the operating system to schedule the applications to complement each other, and (2) utilize cache partitioning to give particular applications a given size of the LLC at the hardware level [15]. Moseley et al. [19] developed an approach to partition the cache for co-running applications based on a real-time model of their IPC. The IPC model is maintained at runtime using a decision tree based on hardware counters sampled every 25 million cycles. Zhuravlev et al. [32] investigated other causes of performance degradation when co-scheduling threads, including memory controller contention, memory bus contention, and hardware prefetching; and developed an approach to reduce this contention using the cache miss rate sampled via hardware counters. Our progress period based approach can be potentially extended to support these alternative optimizations by enabling them to operate at a finer granularity than whole applications.

For determining working set size, Zhao et al. [30] have observed a pattern that is similar to progress periods. Their system dynamically calculates the working set sizes of an application, and they noticed that only during specific phases of execution is memory profiling required. Then during the off phases, they disable memory profiling to minimize overhead. This pattern is similar to how our scheduling extension works, as it is used only during user-defined progress periods. Their work, however, uses hardware counters to dynamically determine these phases.

6 CONCLUSION

To summarize, scheduling algorithms need to consider the behavior of the applications they schedule to achieve a better system energy savings and performance gain. We first introduced a new concept in relation to program execution and scheduling, a *progress period*, which allows developers to describe sections of an application's execution with resource demand characteristics. Then, we presented an operating system scheduling extension that uses these progress periods to understand the resource behavior of the processes it schedules in order to schedule the programs against the hardware more efficiently, leading to a significant decrease in system energy consumption while simultaneously achieving a slight performance

increase. Our resource demand aware scheduling extension is configurable to allow multiple hardware resources to be targeted, as well as tunable to tweak how strict the scheduler is when balancing resource efficiency vs running more programs concurrently by exposing a system policy. We have demonstrated the practicality of this scheduling environment by testing it on a number of programs with varying resource access behaviors; and shown that it decreases a system's energy consumption by up to 48%, while also increasing the speedup of applications by up to 1.88x

The flexible nature of progress period definition allows them to be used and combined in a wide variety of ways, independent of the original programming model. In particular, some functions are well suited for being wrapped entirely in a progress period, while other functions would have multiple progress periods within them. For example, the ocean_cp SPLASH 2 application has a function, *slave2*, which contains three progress periods because the function has multiple phases. Then another function within the same application, *relax*, has a consistent behavior throughout its execution, therefore allowing a single progress period to contain all of its instructions entirely.

We have shown that our scheduling extension would work best alongside the traditional operating system scheduler. Where the traditional scheduling policy would be used for memory bound applications to maximize concurrency, our resource demand aware scheduling policies would be used for programs that have at least a moderate level of data reuse to maximize LLC efficiency.

For future work, we believe extending our scheduler with cache partitioning would be highly beneficial for two reasons. First, if an application whose working set size is larger than the LLC is scheduled (e.g., streaming applications), we can partition the cache and give this application only a small portion of the cache because it would fetch most data from main memory regardless. Second, if an LLC intensive application that doesn't specify any progress periods is run alongside instrumented programs, the resource monitor would be unaware of the behavior. Allowing the instrumented programs to share a large cache partition would allow them to use the resource without external interference.

REFERENCES

[1] 2016. *PERF(1) - Linux Programmer's Manual.*

[2] John a. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio Buttazzo. 1995. Implications of Classical Scheduling Results For Real-Time Systems. In *Computer*, Vol. 28. IEEE Computer Society, 16–25.

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. 2009. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience, To Appear* (2009).

[4] Nick Barrow-Williams, Christian Fensch, and Simon Moore. 2009. A Communication Characterisation of Splash-2 and Parsec. In *2009 IEEE International Symposium on Workload Characterization*. IEEE.

[5] Major Bhadauria and Sally A. McKee. 2010. An Approach to Resource-Aware Co-Scheduling for CMPs. In *Proceedings of the 24th ACM International Conference on Supercomputing*, Vol. 14. ACM, 189–199.

[6] Janki Bhimani, Zhengyu Yang, Ningfang Mi, Jingpei Yang, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan. 2018. Docker container scheduler for i/o intensive applications running on nvme ssds. *IEEE Transactions on Multi-Scale Computing Systems* (2018).

[7] Jichuan Chang and Gurindar Sohi. 2014. Cooperative Cache Partitioning for Chip Multiprocessors. In *International Conference on Supercomputing 25th Anniversary*.

[8] Robert P. Colwell, Robert P. Nix, John J. O'Donnell, David B. Papworth, and Paul K. Rodman. 1988. A VLIW architecture for a trace scheduling compiler. In *IEEE Transactions on Computers*, Vol. 37. 967 – 979.

[9] Ashutosh S. Dhodapkar and James E. Smith. 2002. Managing multi-configuration hardware via dynamic working set analysis. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 233–244.

[10] Ali El-Moursy, Rajeev Garg, David H. Albonesi, and Sandhya Dwarkadas. 2006. Compatible Phase Co-Scheduling on a CMP of Multi-Threaded Processors. In *Proceedings 20th IEEE Parallel & Distributed Processing Symposium*. IEEE.

[11] Venkatraman Govindaraju, Chen-Han Ho, and Karthikeyan Sankaralingam. 2011. Dynamically Specialized Datapaths for Energy Efficient Computing. In *IEEE 17th International Symposium on High Performance Computer Architecture*. 503–514.

[12] Brendan Gregg. 2018. How To Measure the Working Set Size on Linux. (Feb 2018). http://www.brendangregg.com/wss.html

[13] Intel 2015. *Pin 2.14 User Guide*. Intel.

[14] Intel 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel.

[15] Aamer Jaleel, Hasheem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. 2012. CRUISE: cache replacement and utility-aware scheduling. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 249–260.

[16] Melanie Kambadur and Martha A. Kim. 2014. An experimental survey of energy management across the stack. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 329–344.

[17] Joshua Kihm, Alex Settle, Andrew Janiszewski, and Dan Connors. 2005. Understanding the Impact of Inter-Thread Cache Interference on ILP in Modern SMT Processors. In *Journal of Instruction-Level Parallelism*.

[18] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 367–378.

[19] Tipp Moseley, Joshua Kihm, Daniel Connors, and Dirk Grunwald. 2005. Methods for Modeling Resource Contention on Simultaneous Multithreading Processors. In *2005 International Conference on Computer Design*.

[20] Djamila Ouelhadj and Sanja Petrovic. 2008. A survey of dynamic scheduling in manufacturing systems. In *Journal of Scheduling*, Vol. 12. Springer Science, 417–431.

[21] Paradyn 2016. *ParseAPI Programmer's Guide*. Paradyn.

[22] Petar Radojkovic, Vladimir Cakarevic, Miquel Moreto, Javier Verdu, Alex Pajuelo, Francisco Cazorla, Mario Nemirovsky, and Mateo Valero. 2012. Optimal Task Assignment in Multithreaded Processors: A Statistical Approach. In *The Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[23] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. 1992. Load Distributing for Locally Distributed Systems. In *Computer*, Vol. 25. IEEE Computer Society, 33–44.

[24] Michael D. Smith, Monica S. Lam, and Mark A. Horowitz. 1990. Boosting Beyond Static Scheduling in a Superscalar Processor. In *ISCA '90 Proceedings of the 17th annual international symposium on Computer Architecture*, Vol. 18. ACM SIGARCH Computer Architecture News, 344–354.

[25] Martin Tillenius, Elisabeth Larsson, Rosa M. Badia, and Xavier Mortorell. 2015. Resource-Aware Task Scheduling. In *ACM Transactions on Embedded Computing Systems*, Vol. 14. ACM.

[26] Michael J. Voss and Rudolf Eigenmann. 2001. High-Level Adaptive Program Optimizataion with ADAPT. In *Proceedings of the eigth ACM SIGPLAN sympoosium on Principles and practices of parallel programming*. ACM, 93–102.

[27] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*.

[28] Zhengyu Yang, Janki Bhimani, Yi Yao, Cho-Hsien Lin, Jiayin Wang, Ningfang Mi, and Bo Sheng. 2018. AutoAdmin: Automatic and Dynamic Resource Reservation Admission Control in Hadoop YARN Clusters. *Scalable Computing: Practice and Experience* 19, 1 (2018), 53–68.

[29] Qing Yi, Qian Wang, and Huimin Cui. 2014. Specializing Compiler Optimizations Through Programmable Composition For Dense Matrix Computations. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 596–608.

[30] Weiming Zhao, Xinxin Jin, Zhenlin Wang, Xiaolin Wang, Yingwei Luo, and Wiaoming Li. 2011. Low Cost Working Set Size Tracking. In *USENIX Annual Technical Conference*.

[31] Yuhao Zhu and Vijay Janapa Reddi. 2013. High-Performance and Energy-Efficient Mobile Web Browsing on Big/Little Systems. In *IEEE 19th International Symposium on High Performance Computer Architecture*. 13–24.

[32] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. 2010. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*.