LIRA: Adaptive Contention-Aware Thread Placement for Parallel Runtime Systems

Alexander Collins[*]
University of Edinburgh
a.collins@ed.ac.uk

Tim Harris
Oracle Labs, Cambridge
timothy.l.harris@oracle.com

Murray Cole
University of Edinburgh
mic@inf.ed.ac.uk

Christian Fensch
Heriot-Watt University
c.fensch@hw.ac.uk

ABSTRACT

Running multiple parallel programs on multi-socket multi-core machines using commodity hardware is increasingly common for data analytics and cluster workloads. These workloads exhibit bursty behavior and are rarely tuned to specific hardware. This leads to poor performance due to suboptimal decisions, such as poor choices for which programs run on the same socket. Consequently, there is a renewed importance for schedulers to consider the structure of the machine alongside the dynamic behavior of workloads.

This paper introduces LIRA, a spatial-scheduling heuristic for selecting which parallel applications should run on the same socket in a multi-socket machine. We devise two flavors of scheduler using this heuristic: (*i*) LIRA-static which collects performance data in an offline profiling step to decide the schedule when a program starts, and (*ii*) LIRA-adaptive which operates dynamically based on hardware performance counters available on off-the-shelf hardware. LIRA-adaptive does not require separate, offline workload characterization runs, and it accommodates a dynamically changing mix of applications, including those with phase changes.

We evaluate LIRA-static and LIRA-adaptive using programs from SPEC OMP and two graph analytics projects. We compare our approaches to the best possible performance obtained across all static mappings of 4 programs to 2 sockets, the libgomp OpenMP runtime that comes with GCC and Callisto, a state-of-the-art scheduler. LIRA-static improves system throughput by 10% compared to libgomp, and LIRA-adaptive improves system throughput by 13%. Compared to Callisto, LIRA-adaptive improves performance in 30 of the 32 combinations tested, with an improvement in system throughput of up to 7%, and 3% on average over 32 combinations.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*Scheduling*

Keywords

Multi-core, multi-socket, thread placement, adaptive scheduling

[*]Work done while at Oracle Labs.

1. INTRODUCTION

Clusters consisting of commodity hardware are frequently used for a variety of scientific workloads. These include Beowulf clusters or time-sharing systems for parallel applications [1]. Running parallel jobs together on the same machine as part of a cluster is becoming increasingly important as the desire for efficient use of hardware leads to greater co-location of workloads on the same machine [4]. In addition, many parallel workloads are now *malleable* meaning that workloads can run over a varying number of hardware contexts, using abstractions such as OpenMP and *bursty* meaning that their CPU demand varies during execution.

In this paper, we investigate running a dynamically changing mix of this kind of parallel program on multi-socket shared-memory machines. We explore how to make online decisions about which of these programs' threads should be co-located on a single socket.

We build on Callisto [9], a user-mode framework for exploring the interaction between the system-wide scheduler and the runtime systems in individual programs. Callisto reduces scheduler-related interference by reducing lock-holder pre-emption problems, by reducing load imbalance between worker threads within a program, and by making explicit thread-to-core allocations which adapt to the amount of parallelism available within a program.

In this paper, we extend Callisto to handle more than two programs on a shared-memory multi-core multi-socket machine and develop LIRA-adaptive, an online adaptive scheduler that selects which sets of programs should share cores on the same socket, and is able to respond to phase changes within a program's execution.

We evaluate our adaptive scheduler by comparing it with: (*i*) best-static which selects the best program-to-socket mapping for a given workload by exhaustively trying every combination *a priori*, (*ii*) LIRA-static which selects the program-to-socket mapping using the same heuristic as LIRA-adaptive but based on per-program solo-run profiling, (*iii*) Callisto and (*iv*) libgomp from GCC with scheduling performed by the default Linux OS scheduler. LIRA-static avoids the cost of collecting profiling information during execution, but also prevents adaptation to phase behavior.

- We demonstrate how different program-to-core mappings affect performance and measure the performance degradation caused by ignoring the presence of separate sockets.
- We develop an online adaptive scheduler, which we call LIRA-adaptive, which reduces interference and resource contention in a multi-socket environment.
- We compare LIRA-adaptive to two competing scheduling approaches: Callisto and libgomp OpenMP.
- We also demonstrate that LIRA-adaptive is better than an offline static approach (LIRA-static) using the same heuristic.

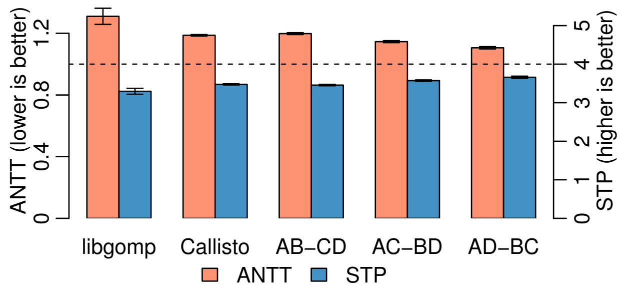

Figure 1: The Average Normalized Turnaround Time (ANTT) and System Throughput (STP) achieved by: (*i*) libgomp from GCC 4.8.0, (*ii*) Callisto and (*iii*) Callisto with the three static program-to-socket mappings. Error bars show 95% confidence intervals for the mean.

2. MOTIVATING EXAMPLE

Figure 1 shows the performance of running four programs concurrently on a dual-socket machine with 8 cores per socket, using different runtime systems and program-to-socket mappings.

The experiment uses ammp (A) and swim (B) from the SPEC OMP 2012 benchmark suite, and pagerank (C) and triangle_counting (D) from the Green-Marl domain-specific graph analytics project [10]. The programs all run concurrently. Each is run for at least 9 repeats, and possibly more to ensure that all 4 programs are running throughout the experiment.

We measure two system-wide performance metrics: Average Normalized Turnaround Time (ANTT) and System Throughput (STP) [6]. Both metrics are discussed further in Section 3.

Ideally, we would like to achieve an ANTT of 1 and an STP of 4. This would be the result if there was no resource contention between programs. In practice, worse results may be seen when contention is significant, whereas better results may be achieved if bursty workloads interact well, for example by overlapping program execution to hide blocking for disk I/O or synchronization.

Figure 1 compares five scheduling variants of this workload:

- libgomp – The OpenMP implementation of libgomp from GCC 4.8.0. Each program is run using a separate instance of the libgomp library. Each is configured to use passive synchronization and 16 OpenMP threads. Therefore, each program has sufficiently many threads to make use of all cores on the system. Scheduling of these threads is performed by the default Linux 2.6.32 scheduler.

- Callisto – This variant uses Callisto [9]. Each program creates 16 OpenMP threads, and Callisto multiplexes these over the 4 cores allocated to each program. The sets of 4 cores will generally be in the same socket, but there is no control over specifically which program gets which set of cores (this may vary over time as programs start and complete). If one program cannot use all of its allocated cores, then Callisto makes these available to other programs; this provides ammp and swim with additional cores when pagerank and triangle_counting are loading their input graphs.

- AB-CD, AC-BD, AD-BC – These remaining three configurations use a modified version of Callisto that fixes each program's threads to a specific quarter of the machine. For programs A–D, the notation indicates which pairs of programs are placed on the same socket. For instance AD-BC indicates that A and D are together, and that B and C are together. Accounting for symmetry, there are three alternative choices of allocating programs to sockets.

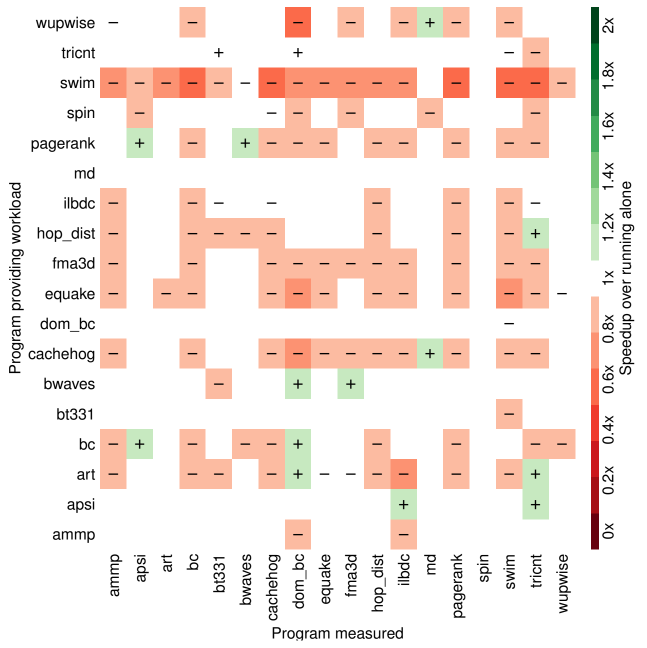

Figure 2: Pair-wise speedup of programs, comparing sharing a socket to using separate sockets. Boxes annotated with a – indicate cases where performance decreased, and + where performance increased. Regions marked with a – show where system performance can be improved.

The results show that the choice of program to socket mapping has an impact on both the ANTT and STP of the system. Callisto achieves better ANTT than OpenMP, which is expected given that Callisto's aim is to reduce interference between programs. This interference can be reduced further by partitioning programs within separate sockets, as shown by the improved ANTT and STP achieved by each of the configurations AB-CD, AC-BD and AD-BC. Moreover, the choice of pairings of programs has a significant effect on performance. The ANTT and STP varies amongst the three configurations, with AD-BC achieving better ANTT and STP compared to the other two. LIRA-adaptive attempts to identify these best pairings at runtime.

3. MEASURING PERFORMANCE

Average Normalized Turnaround Time (ANTT) and System Throughput (STP) are system-level metrics suitable for exploring the performance of multi-program environments.

ANTT is a measure of the perceived slow-down of programs, compared to executing them in isolation in a given environment, and is a lower-is-better metric. STP is a measure of the rate at which the system completes work. It is a higher-is-better metric.

The ANTT and STP of a system running n programs are computed as follows. Here, T_i^S is the execution time of each program when run in isolation, and T_i^M is the execution time of each program when run alongside other programs.

$$ANTT = \frac{1}{n} \sum_i \frac{T_i^M}{T_i^S} \qquad STP = \sum_i \frac{T_i^S}{T_i^M}$$

Ideally, we want to achieve an ANTT of 1 and an STP of n, which corresponds to the execution time being unaffected by running in a multi-program environment.

In order to quantify noise, we run each experiment for multiple repeats. We also run programs repeatedly within experiments to ensure that the same workload exists on the system for the duration of the experiment.

4. SOCKET SCHEDULING HEURISTIC

In this section we describe how LIRA characterises programs at runtime, to identify a schedule that is likely to perform well. Section 4.1 explores the performance degradation that occurs when running pairs of programs on a multi-socket machine, and Section 4.2 describes our heuristic technique for predicting program pairings that minimize this degradation.

4.1 Pairwise Performance Degradation

Figure 2 shows a comparison between running pairs of programs on the same socket to running them on distinct sockets.

We use a 16 core dual-socket machine for this experiment. The programs used are detailed in Section 6.1. Each program is configured to run 4 threads, pinned to either 4 distinct cores on different sockets (A_-B_ using our previous notation), or 4 distinct cores on the same socket (AB-__). This setup ensures that each program is given the same amount of computational resource – 4 threads pinned to 4 distinct physical cores – and that each thread has exclusive use of the core to which it is pinned. Therefore any change in performance is due to thread placement.

These results show that there can be a significant performance impact associated with sharing the same socket as another program. For about half of the program combinations there is an increase in execution time of 20%, and a maximum increase of 50%. For the other half of the programs there is minimal impact on sharing sockets. In some rare cases, there is actually an *increase* in program performance. In two cases, there is a 1.4x increase in performance (discussed in Section 6.2).

These results suggest that a smarter scheduling approach could avoid program slowdown by carefully choosing which programs should share the same socket. It is also interesting to note that there are clear "rows" and "columns" of red and white for some applications. For example, md is amenable to sharing the system as it does not experience or cause slow-down when paired with any programs. In contrast, swim adversely affects the performance of most programs when they run alongside it, except for md. Therefore md and swim are good candidates to co-locate on the same socket, to improve overall system performance.

4.2 LIRA: Heuristic for Socket Scheduling

Modern CPU architectures provide many hardware performance counters, in the form of a set of dedicated hardware registers that are incremented by the control logic of the CPU itself. These can be used to record events on a per-thread basis, with low overhead or impact on the behavior of the program. They include events such as the number of cache misses at different levels of the hierarchy and the number of completed instructions. The specific events that are available is dependent on the underlying hardware. We use the Performance API library [14] to set up and measure hardware counters on our experimental platform. These counters provide some measure of the behavior of programs, that we can use as program features to build our predictive model which is the basis of our scheduling heuristic.

Figure 3 shows the speedup of running pairs of programs concurrently on the same system over running them in isolation, against

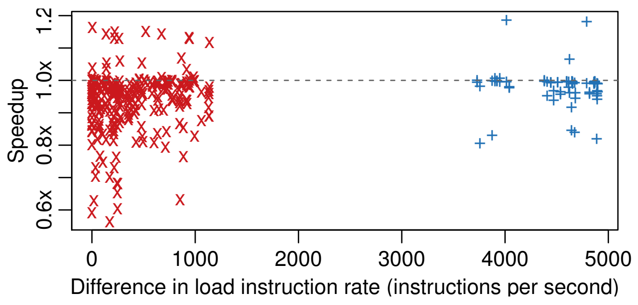

Figure 3: Average speedup of pairs of programs compared to execution in isolation, plotted against the absolute difference in the load instruction execution rate. There are two distinct clusters in the data, shown by the red x and blue + markers.

the absolute difference in the rate at which the programs execute load instructions. This rate is measured separately for each of the program's threads then averaged using the arithmetic mean. For these measurements, we use the same 16 core dual-socket machine as before. These results show that the average slowdown is greatly reduced when the difference in load instruction rate is maximized. Program pairs with a large difference in load instruction rate (the cluster to the right of the plot) have a geometric mean speedup of 98% and a maximum slowdown of 19%. In contrast, program pairs with a small difference in load instruction rate (the cluster to the left of the plot) have smaller geometric mean speedup of 93% and a much higher maximum slowdown of 44%. This shows that pairs of programs with different load instruction rate are more likely to achieve good performance. The intuition here is that, by pairing programs in this way, pressure on the memory system is reduced.

We use this as our heuristic for predicting which pairs of programs will cooperate more effectively when run on the same socket.

The LIRA Heuristic

Given a set of programs to run on the system, each with a given load instruction rate, we choose a mapping from programs to sockets such that the absolute difference in the instruction rate of the programs on each socket is maximized. Consider a dual socket machine with four programs running, programs A and B are scheduled to the first socket, and programs B and C to the second socket. The programs have load instruction rates R_A, R_B, R_C and R_D. The chosen schedule is the one that maximises the expression:

$$\text{abs}(R_A - R_B) + \text{abs}(R_C - R_D)$$

5. SPATIAL SCHEDULING FOR SOCKETS

In this section we introduce LIRA-adaptive, an online adaptive scheduler built on top of Callisto. Section 5.1 explains how Callisto's spatial thread scheduling works. Section 5.2 discusses Callisto's behavior in a multi-socket environment. We then describe two variants of our multi-socket-aware scheduler: Section 5.3 describes LIRA-static which uses profile data to perform static scheduling, and Section 5.4 describes LIRA-adaptive which performs online adaptive scheduling.

5.1 Callisto's Thread Scheduler

The Callisto runtime system uses *dynamic spatial scheduling* to allocate threads to physical cores. Each program that runs on the system spawns multiple *worker* threads and pins each thread to each physical core. Of the threads pinned to each core, one is designated the *high priority* thread, and the remainder as *low*

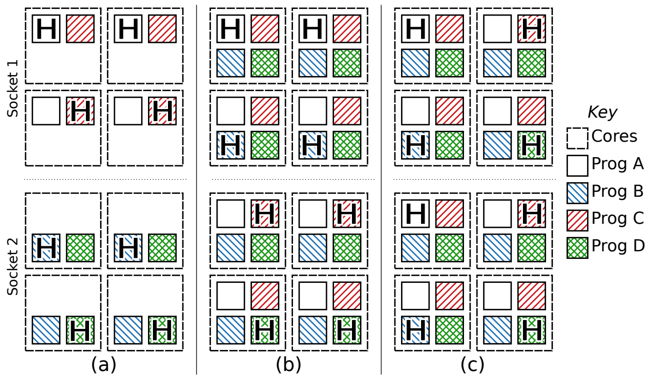

Figure 4: Example thread to socket schedules. High priority threads are marked with an H. (a) produced by LIRA-static (pairings A,B and C,D are determined to be optimal using the scheduling heuristic in Section 4.2), (b) Callisto's spatial scheduling, and (c) A pathologically bad schedule produced by Callisto, with maximal intra-socket communication overhead.

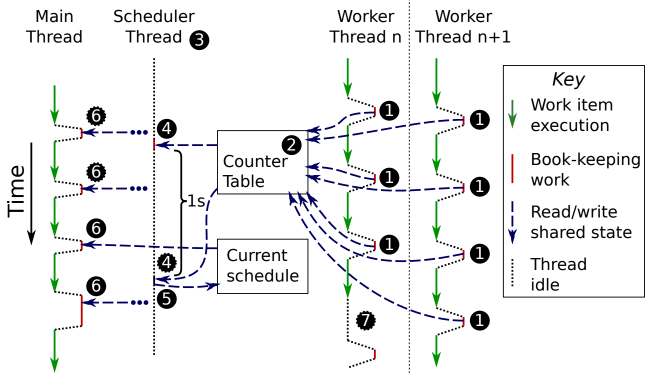

Figure 5: LIRA-adaptive, the online adaptive scheduler.

priority threads. An example of this is shown in Figure 4. Callisto ensures that each program has an equal share of high priority threads, and that the main thread for each program is given high priority. The aim of this is to ensure that the main thread can always run, as it often acts as a producer of parallel tasks, and so its performance is critical to the performance of the program as a whole. This also provides a fair distribution of resources across all running programs.

Callisto's aim is to run high priority threads most of the time. This means that the high priority threads experience low interference from other threads running on the system. For example, they can make full use of core-local caches, without the threat of other programs evicting cache lines that would lead to performance degradation. This setup also reduces the number and frequency of context switches, reducing the overhead they incur.

In order to maintain good utilization of resources, a low priority thread is allowed to run when the high priority thread is not runnable, for example when the high priority thread blocks for disk accesses or synchronization. Due to the bursty nature of many parallel workloads this is essential to make good use of the available hardware resources. Callisto limits the frequency with which context switching to low priority threads can occur using a configurable hysteresis threshold, typically around 10ms. If a high priority thread blocks for longer than a fixed number of processor cycles, it is stopped and a low priority thread allowed to run. The high priority thread is only allowed to run again after it has been runnable for sufficiently many processor cycles.

5.2 Multi-Socket Scheduling

Callisto's thread scheduler treats the system as a homogeneous array of cores. It arbitrarily assigns programs to cores, and allows a program to have threads running on different sockets. This means it does not necessarily allocate programs to sockets in a manner that reduces interference. Callisto's spatial scheduler can lead to the situation where a low priority thread is run on a different socket from the high priority threads. This is likely to incur additional intra-socket communication as data is copied to the caches on the other socket. Synchronization may also have to be performed across the socket boundary in this case, which may cause the high priority threads to block whilst waiting for the low priority thread to complete. For example, Callisto could produce the schedule shown in Figure 4 which is pathologically bad. Each thread needs to communicate across the socket boundaries, which introduces communication overheads that would be unnecessary if the threads were scheduled to the same sockets.

Our scheduler uses the LIRA heuristic to extend Callisto by considering the fact that the cores exist in separate sockets. Our approach aims to automatically allocate programs to sockets such that interference and contention for resources is reduced. When there are more programs than sockets, we also keep all of the threads for each program on the same socket, to avoid the situation where a low priority thread is run on a separate socket. However, within each socket, we use Callisto to schedule the threads. This improves utilization within each socket, by allowing low priority threads to run if the high priority threads block.

We develop two scheduling techniques: LIRA-static (Section 5.3) uses profile data collected *a priori* to decide which programs to allocate to which sockets, and LIRA-adaptive (Section 5.4) observes the programs at runtime to adaptively allocate programs to sockets.

Our profile-based and online adaptive approaches rely on being able to anticipate when running programs on the same socket will lead to bad performance, compared to running them on separate sockets, and avoid these cases. In order to do this, we devise a model that maps the properties of the pairs of programs to a performance estimate. We use hardware performance counters to provide these properties.

5.3 LIRA-static: Profile-Driven Scheduler

Our profile-driven scheduler uses information about program behavior collected *a priori* to schedule programs to sockets.

To run a program on the system, it must first be profiled. The application programmer provides a sample input and the program binary to the system. The program is then run exclusively on a single socket of the machine, and hardware performance counters are used to measure its behavior. The values of these are converted to rates (normalized by the total execution time of the program) and stored in a database for use in scheduling decisions.

When a program is run on the system, the scheduler examines the database for the behavior data for every program that is running. This data is used to predict the best allocation of *programs* to sockets. We then use the original Callisto strategy to schedule each program's threads within each socket. This means that, within each socket, each program spawns and pins one thread to every core, and each program has an equal share of high priority threads. This prediction is made using the LIRA heuristic described in Section 4.2. Figure 4 shows an example of this spatial scheduling.

This scheduling decision is only made when a program is invoked. The schedule does not adapt during program execution. In our experiments we focused solely on the case where sets of programs are run simultaneously.

This static profile-driven approach requires a potentially expensive training phase, however it incurs no runtime overhead. This approach is used as an additional baseline to compare against our more sophisticated online adaptive scheduler, described in the following section.

5.4 LIRA-adaptive: Online Scheduler

Our online adaptive scheduler uses the same LIRA heuristic as the profile driven scheduler, described in Section 5.3. This heuristic is used to choose the best program to socket allocation during program execution. This removes the need for a separate profiling step, makes the approach input agnostic, and allows the schedule to adapt to changes in program behavior during program execution.

The online adaptive scheduler consists of the following two components. Figure 5 shows a timing diagram of the operations performed by these components.

- *Performance Monitoring*

  Each thread periodically measures its hardware performance counters (shown by ❶ in Figure 5), and updates a process-shared table with this information (❷). The time interval between updates to this table is configurable. We include a sensitivity study of this parameter in Section 6.4.

  Each thread measures the number of executed instructions and the number of executed load instructions since the last update. These values are used to compute the rate at which load instructions are executed since the previous update, for the program as a whole by averaging across all threads. Each thread also measures the number of CPU cycles spent in a runnable state since the last update. A thread is in the runnable state if it is a high priority thread and is not blocked for I/O or synchronization. This is used to determine whether a thread is idle as described in Section 5.4.1.

  The process-shared table stores these hardware performance counter measurements for each thread running in each process on the system. The cache miss rates and number of cycles are smoothed using an exponential moving average. This smoothing avoids short lived changes from affecting the scheduling decision, which would incur large overheads due to frequently moving threads to different cores.

- *Scheduling*

  Each program spawns an additional thread to perform scheduling decisions (❸). These threads are pinned to the same cores as each program's main thread. They periodically check the information stored in the shared-process table (containing performance information collected by the performance monitor) to decide if the thread schedule should change (❹). The time interval between these updates is configurable. We include a sensitivity study of this parameter in Section 6.4.

  The scheduler computes the arithmetic mean of the cache miss rates for each thread in each program. The LIRA heuristic (described in Section 4.2) is used to assign a score to every possible placement of programs on sockets. The schedule with the highest score is chosen as the new schedule.

  This decision is written to a process-shared piece of memory (❺), so that the main threads in other programs can detect the change and apply the new schedule (❻).

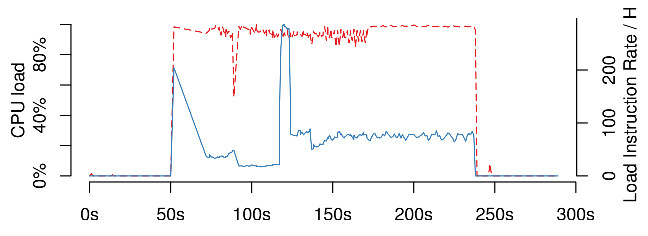

Figure 6: The CPU load (red, dashed line) and load instruction rate (blue, solid line) over time for pagerank. This demonstrates the loading and completion phases with low CPU load, and phase changes in load instruction rate.

  Work is allocated to threads based on the schedule, and threads that are not allocated work simply remain idle ❼.

5.4.1 Dealing with Bursty CPU Load

Programs often involve an input or output phase with significant amounts of I/O to disk. For example, the graph analytics benchmarks load a large graph from disk before performing computation over it. During these phases, CPU usage is minimal – the worker threads are essentially idle. Therefore the threads do not run very often and there is no meaningful load instruction rate to measure. Figure 6 shows an example of this behavior.

LIRA-adaptive handles this by first classifying the programs running on the system as idle or active. This is done by measuring the number of cycles that each high priority threads spends in a runnable state (i.e. when not blocking for I/O or synchronization). This is converted to a percentage CPU load and averaged across all program threads. If this average CPU load is below 50% the program is considered idle. Note that the choice of this threshold is not important as the common case is that the program's average CPU load is either near 100% or near 0%.

The schedule is determined based on three cases: (*i*) no idle programs – the load instruction rate LIRA heuristic is used to schedule programs to sockets; (*ii*) at least 1 idle program – the load instruction rate of the idle program is assigned a value of 0; and (*iii*) more idle programs than sockets – the schedule is left unchanged.

6. EVALUATION

In this section we analyse the performance of our profile-driven and online adaptive schedulers compared to the libgomp OpenMP implementation and Callisto. Section 6.1 details the setup for our experiments, Section 6.2 compares our scheduling techniques against OpenMP and Callisto, and Section 6.3 compares against an optimal static policy.

6.1 Experimental Setup

We use two performance metrics to compare and contrast the scheduling approaches – ANTT and STP. These metrics are described in detail in Section 3.

For our experiments we use a dual-socket machine, with a pair of Xeon E5-2660 processors clocked at 2.20GHz. Each processor has 8 physical cores with 2 hardware threads per core. We disable hyperthreading to focus on the effects of multiple sockets, and will investigate the effect of hyperthreading in future work. Each socket has 128GB of main memory, for a total of 256GB, and runs Linux 2.6.32. We use GCC 4.8.0 to compile the benchmark programs.

We use 18 benchmark programs taken from four different sources. Firstly, we use the 11 benchmarks from SPEC OMP 2001 and 2012 that are supported by Callisto. These are the benchmarks that do not use manual locking via calls to `omp_set_lock` and `omp_unset_lock`. These calls are used by (*i*) nested parallel sections, (*ii*) the ordered directive and (*iii*) explicit tasks.

We also include an implementation of the betweenness-centrality graph algorithm [3] written using CDDP, a constrained data-driven parallelism programming model [8]. Four graph analytics programs are also included. They are written in the domain specific Green-Marl language [10], which is compiled to OpenMP using the Green-Marl compiler.

Finally, we include a pair of micro-benchmarks: `spin` and `cachehog`. `spin` simply executes CPU-bound computation, with a very small working set, so as to put minimal stress on the memory system. `cachehog` executes memory-bound computation, with the aim of maximising the number of misses in the last-level cache. Running these synthetic workloads alongside other programs helps us understand the reasons for the behavior that we see.

We use input sizes that require approximately one minute of execution time when run alone on the machine. For the SPEC OMP 2006 benchmarks, we used the “large” inputs, and slightly reduced size inputs for the SPEC OMP 2012 benchmarks. For the graph analytics benchmarks we use a large Twitter graph with 42 million nodes and 1,500 million edges.

Given our set of 18 benchmark programs, we choose 32 random combinations of 4 programs from the set. We run each of the 32 sets of programs using each of the runtime environments: the libgomp OpenMP implementation from GCC, the Callisto runtime library, our static profile-driven scheduler (LIRA-static) and our online adaptive scheduler (LIRA-adaptive). We also measure the performance of every permutation of statically allocating programs to sockets, to provide best and worst case bounds for LIRA-static (worst-static and best-static). To compute the ANTT and STP for each instance, we record the execution time for each program run in isolation on the machine. We therefore run each benchmark program in isolation, utilizing all 16 cores on the machine and the libgomp OpenMP implementation.

Note that using this experimental setup we are comparing our online adaptive scheduler, with its performance tracking instrumentation, against libgomp and Callisto without runtime instrumentation. Our results therefore include any runtime instrumentation overhead. These overheads are amortized by the increased performance and are quantitatively analysed in Section 6.4.

To quantify the error in our measurements, we run each program for at least 9 repeats. The execution time of each benchmark program differs, therefore we repeatedly execute each of the four programs. This maintains the system workload for each program. We measure the time for the entire execution of each program, including loading data from disk.

6.2 Comparison with libgomp and Callisto

Figure 7 shows the percentage improvement in ANTT and STP achieved by each approach, compared to Callisto. This is computed as the percentage change in ANTT/STP for each program combination compared to the ANTT/STP achieved by Callisto.

Our results demonstrate that LIRA-adaptive performs the best. In all cases it achieves performance at least as good as Callisto, and usually better. On average, LIRA-static performs similarly to Callisto. This is likely due to both Callisto and the static approach choosing a single schedule for the entire program run. The schedule does not adapt to changes in program behavior during execution. This demonstrates that the adaptive approach used by LIRA-adaptive is required. Moreover, any overhead incurred by LIRA-adaptive's more complex implementation is amortized by the performance gains it provides.

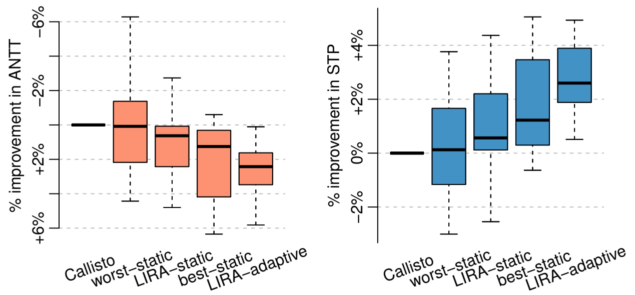

Figure 7: Box plots showing the percentage improvement in Average Normalized Turnaround Time (ANTT) (left) and System Throughput (STP) (right) achieved by each runtime system, across all 32 combinations of 4 concurrently running programs. For ANTT, lower is better, and for STP, higher is better. Thick horizontal lines show the median, shaded boxes show the interquartile range, and whiskers show the maximum/minimum.

Figure 8 shows the ANTT and STP for each program set achieved by both our static profile-driven scheduler (LIRA-static) and online adaptive scheduler (LIRA-adaptive), compared to the libgomp OpenMP implementation, the Callisto runtime system, the best possible performance (best-static), and the worst possible performance (worst-static), when choosing a schedule statically.

In most cases, libgomp performs the worst. This is not surprising, as the benchmarks have likely been tuned assuming that the program will be using the machine exclusively. This demonstrates the need for socket aware scheduling given concurrently running programs. In most cases, Callisto performs better than libgomp, achieving an ANTT close to 1 and an STP close to 4 in each case. LIRA-static performs, on average, slightly better than Callisto. LIRA-adaptive performs similarly to the profile-driven approach for ANTT, improving performance in roughly half the cases, but degrading it in others. However, it significantly improves STP in many cases, achieving an STP greater than 4 in some cases. In some cases, LIRA-adaptive achieves an ANTT lower than 1. This shows that by pairing together programs that interact well performance can actually be improved compared to running in isolation. This is due to stalls in execution being avoided by context switching to the other concurrently running program, but in such a way that the switch does not harm the performance of either program.

Three interesting cases are 10, 16 and 26. In these cases libgomp achieves the best ANTT. Callisto harms performance slightly, and LIRA-static harms performance further. This is due to the increasing amount of runtime overhead introduced by each approach, and the fact that the programs in these cases do not interfere with one another (as shown in Figure 2). There is therefore no performance to be gained by carefully choosing the program-to-socket mapping, and therefore no performance gain that can be used to hide these overheads. In these three cases LIRA-adaptive also performs worse than libgomp, but at least as well as Callisto.

6.3 Comparison with Optimal Static Policy

Figure 8 also compares against two static oracles (best-static and worst-static), which exhaustively try all static thread-to-socket allocations to find the best/worst choice. The results show that, in most cases, the profile-driven approach achieves the best performance it can (the static oracle provides an upper bound on the performance that a static scheme can achieve). This supports the hypothesis that

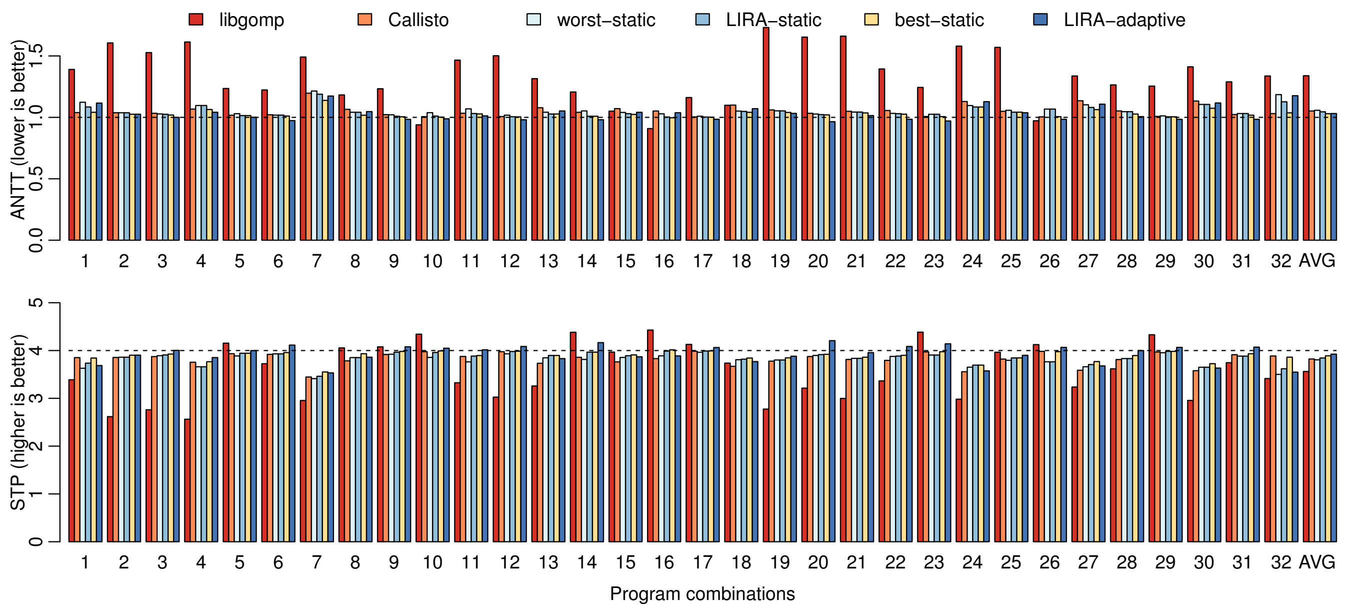

Figure 8: Comparison of the Average Normalized Turnaround Time (ANTT) and System Throughput (STP) achieved by different approaches, for 32 random combinations of 4 concurrently running programs. For ANTT, lower is better, and for STP, higher is better. AVG shows the arithmetic mean of the ANTT and STP for each approach.

the LIRA heuristic is a good way of choosing non-interfering pairs. The results also show that in some cases there is large scope for improving performance using a dynamic approach. In some cases this comes at a cost in ANTT, but improves STP.

6.4 Sensitivity Analysis

Figure 9 shows the effects of varying the two implementation parameters present in LIRA-adaptive: (*i*) the time delay between scheduler invocations, controlling the frequency at which the system can adapt to changes, and (*ii*) the time delay between samples of the hardware performance counter, controlling the accuracy of the load instruction rate and CPU load information used in the LIRA heuristic. This experiment was performed identically to previous experiments. We run 32 combinations of 4 concurrently running programs, on a dual-socket 16-core shared-memory machine.

Time Delay Between Scheduler Invocations.

For this parameter, we investigate time delays of 0.1 seconds (used by LIRA-adaptive), 1 second and 5 seconds. We compare against LIRA-static, where the scheduler is invoked exactly once, at the start of the programs' execution, but with a priori knowledge of the performance counters averaged over the program's entire execution. The time delay between counter samples was set to 2×10^9 cycles.

The top two plots in Figure 9 show the effect on ANTT and STP. As the time delay is increased, performance decreases (shown by an increase in ANTT and a decrease in STP). The median ANTT and STP is similar for each plot, however the spread of ANTT and STP across combinations increases. This increased spread shows that some programs performance is adversly affected by increasing this parameter value, whilst others maintain the same performance.

In the case of only running the scheduler at the start of the programs' execution, the performance across all benchmarks decreases significantly. This is shown by the increase in ANTT and decrease in STP. This demonstrates that best performance is achieved when the scheduler is invoked frequently, so that the runtime system can adapt the scheduler to the changing program behavior.

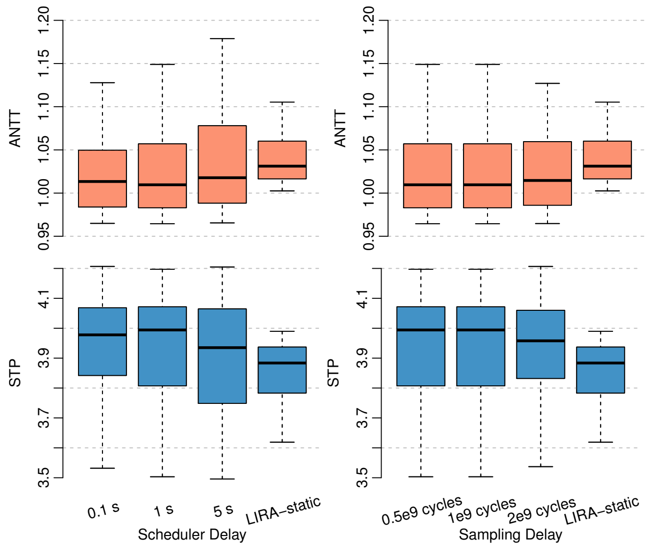

Figure 9: Box plots showing (*i*) the effect of varying the time delay between scheduler invocations on Average Normalized Turnaround Time (ANTT) (top left) and (*ii*) System Throughput (STP) (bottom left), and (*iii*) the effect of varying the time delay between samples of the hardware counters on ANTT (top right) and (*iv*) STP (bottom right). For ANTT, lower is better, and for STP, higher is better. The thick horizontal line shows the median, the box shows the interquartile range, and the whiskers show the maximum and minimum values. LIRA-adaptive **uses a time delay of 0.1 seconds between scheduler invocations, and a time delay of 0.5×10^9 between hardware counter samples.**

Time Delay Between Hardware Counter Samples. For this parameter, we investigate time delays of 0.5×10^9 cycles (the value used by LIRA-adaptive), 1×10^9 cycles and 2×10^9 cycles. We also compare against LIRA-static. The time delay between scheduler invocations is 5 seconds for this experiment.

The bottom two plots in Figure 9 show the effect on ANTT and STP. Performance is similar for time delays of 0.5×10^9 and 1×10^9 cycles. Performance decreases for a time delay of 2×10^9 cycles, shown by an increase in the median ANTT, and decrease in the median STP. Performance is worst for LIRA-static, where the scheduler is unable to respond to changes in program behavior. This shows that frequent measurements of the hardware counters are required to provide the scheduler with timely information, so that it can adapt the schedule effectively.

7. RELATED WORK

Callisto [9] runs multiple parallel applications on a shared machine. The experiments only consider pairs of workloads sharing a dual-socket machine, and assume that workloads are ambivalent to exactly which resources they receive. Zhuravlev et al. provide an extensive survey [17]. We highlight the work most relevant to our own here, and additional recent papers.

Snavely et al. [15] explore co-scheduling of threads in SMT systems. Their system dynamically explores different combinations of thread placement. However, unlike LIRA-adaptive, a profiling phase is required. McGregor et al. [13] select pairs of threads to co-schedule on a hyperthreaded processor. They examine bus transactions, stall cycles and LLC miss rate, and select sets of threads to run per-quantum, attempting to balance the chosen metric between quanta. Their results suggest that using stall cycles are effective, reflecting the fact that contention between the pairs of hyperthreads was most significant. Knauerhase et al. [11] coschedule "light" and "heavy" tasks where pairs of cores share an LLC. Their experiments suggested that cache misses per cycle was the best indicator of interference. As with Fedorova et al. [7], they increase the amount of CPU time given to "light", reflecting the fact that "light" tasks were more likely to suffer from interference.

Zhuravlev et al. investigate scheduling techniques for interference between threads sharing an LLC [16]. They observe that contention in memory controllers, memory buses, and prefetching hardware could be significant. They sort threads by miss rate, and distribute them such that the total miss rate is balanced across LLCs. They described an online algorithm which dynamically measures miss rates.

Bhadauria and McKee [2] investigated co-scheduling applications, using performance counters to identify when programs fail to scale, then applying search heuristics to choose which programs to run together. Their system exploits the fact that some applications experience better scaling while receiving an apparently unfair resource allocation. Programs to co-schedule are selected based on profiling information from initial sampling runs.

ReSense [5] is a system for dynamically controlling the number of threads in concurrent applications. Programs sensitivity to sharing memory is characterised from single-program profiling designed to stress each resource. Thread-to-core mappings are decided when applications/threads start/stop.

Libutti et al. [12] explored a user-mode resource management mechanism to select workloads to co-schedule. The hardware is divided into "binding domains" (BDs), with BDs either allocated to individual processes, or shared between them, using an offline training phase is needed.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we establish the need for a *socket-aware* scheduler for data analytics and cluster workloads, run on shared multi-socket multi-core machines. Over a suite of programs, our adaptive scheduler achieves a 31% improvement in ANTT and a 13% improvement in STP on average, compared to the libgomp OpenMP runtime system. Compared to Callisto, it improves system throughput by up to 7%, and 3% on average across all 32 combinations.

In the future, we plan to evaluate our scheduler on a wider range of workloads, and for more concurrently running programs. We also plan to evaluate it on a wider variety of hardware and explore hyper-threading. Finally we plan to explore the use of finer grained scheduling, to co-schedule individual threads.

References

[1] Top500. `http://www.top500.org`. Accessed: 2015-03-20.

[2] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for CMPs. In *Proc. of the 24th Intl. Conf. on Supercomputing*. ACM, 2010.

[3] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25, 2001.

[4] G. S. Choi, J. Kim, D. Ersoz, A. B. Yoo, and C. R. Das. Coscheduling in clusters: Is it a viable alternative? In *Intl. Conf. on Supercomputing*, Nov 2004.

[5] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. ReSense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Transactions on Architecture and Code Optimization*, 10(4), Dec 2013.

[6] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3), 2008.

[7] A. Fedorova, M. Seltzer, and M. D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proc. of the 16th Intl. Conf. on Parallel Architecture and Compilation Techniques*. IEEE, 2007.

[8] T. Harris, Y. Lev, V. Luchangco, V. Marathe, and M. Moir. Constrained data-driven parallelism. In *Proc. of the 5th Workshop on Hot Topics in Parallelism*, Jun 2013.

[9] T. Harris, M. Maas, and V. J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proc. of the 9th ACM European Conf. on Computer Systems*, 2014.

[10] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proc. of the 17th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[11] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3), May 2008.

[12] S. Libutti, G. Massari, P. Bellasi, and W. Fornaciari. Exploiting performance counters for energy efficient co-scheduling of mixed workloads on multi-core platforms. In *Proc. of the PARMA-DITAM Workshop*. ACM, 2014.

[13] R. L. McGregor, C. D. Antonopoulos, and D. S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Proc. of the 19th IEEE Intl. Parallel and Distributed Processing Symp.* IEEE Computer Society, 2005.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho. PAPI: A portable interface to hardware performance counters. In *Proc. of the DoD HPCMP Users Group Conf.*, 1999.

[15] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *9th Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2000.

[16] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *15th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*. ACM, 2010.

[17] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Computing Surveys*, 45(1), 2012.