International Journal of Innovative
Computing, Information and Control
Volume **9**, Number **4**, April **2013**

 ISSN 1349-4198
pp. 1511–1522

MULTICORE SCHEDULING BASED ON LEARNING FROM OPTIMIZATION MODELS

George Anderson, Tshilidzi Marwala
and Fulufhelo Vincent Nelwamondo

School of Electrical Engineering
University of Johannesburg
Auckland Park 2006, Johannesburg, South Africa
georgeganderson@gmail.com; tmarwala@uj.ac.za; fnelwamondo@csir.co.za

Received February 2012; revised June 2012

Abstract. *Scheduling for multicore computer systems is a challenge since subsets of cores may share resources, such as a cache. Performance for workloads may therefore vary depending on which tasks are scheduled to run on the same subset of cores. There is therefore a need for contention-aware scheduling. Our study involves implementation of an optimal multicore scheduler which has perfect prediction of negative consequences of all combinations of tasks scheduled on the hardware platform we are using. It uses an Integer Program. We show that it is indeed optimal for our workloads, supporting its authors' claims. We also implemented a scheduler which uses a combination of a machine learning model (M5 Prime) and Linear Programming to schedule tasks on CPU cores and compared it to a state-of-the-art scheduler known as Distributed Intensity (DI). Some workloads exhibited moderate improvements in unfairness, but not much in runtime. A scheduler based on another machine learning model was implemented, this time a Multilayer Perceptron (MLP). To do this, we had to generate workloads for the Optimal Scheduler, store its decision for various types of workloads, and train a Multilayer Perceptron model on this data. We then implemented a scheduler with the Multilayer Perceptron model, which tries to mimic the decision of the Optimal Scheduler. Our results show that our scheduler is better than DI in 7 out of 9 test workloads (mostly by 10%), and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads (exactly equal in 4). The MLP scheduler is faster than the Optimal Scheduler by a wide margin, and faster than the Linear Programming scheduler.*

Keywords: Multicore scheduling, Machine learning, Induction of knowledge, Optimization

1. **Introduction.** The Operating System scheduler assigns tasks to Central Processing Units (CPUs) [1, 2, 3]. The scheduler decides which tasks will use a particular CPU next and for how long. Modern CPU designs have shifted from trying to increase clock speeds of CPUs to incorporating more processor cores on a single CPU chip, providing facilities for parallel execution of programs, similar to the traditional SMP (Symmetric Multiprocessing) architectures [4, 5]. Instead of having multiple CPUs, modern computers have multiple cores on a single CPU, which is cheaper and more energy efficient. With the advent of multicore systems, the processing elements (cores) share some resources, for example, the LLC (last level cache), memory controller, memory bus, and prefetch hardware. Therefore, when two or more tasks are running on cores that share these resources, there is resource contention; their performance will be degraded, compared with if each task were to run on its own [6, 7]. Generally, on multicore architectures, competing tasks face performance degradation when co-scheduled together, on separate cores sharing resources, while cooperating (communicating) tasks face performance degradation when

not co-scheduled [8]. Because of this degradation, it is important that when the scheduler is making its decisions, which task should run on which core, it takes resource contention into account.

Siddha et al., while Senior Engineers at Intel, the largest computer chip manufacturer, emphasized the importance of moving to multicore architectures [9]. They highlighted the importance of scheduling in taking full advantage of multicore architectures, including the need to identify and predict task needs and minimize resource contention, i.e., the need for contention-aware scheduling. Without contention aware scheduling, schedulers cannot make good decisions. Contention-aware scheduling studies fall into two categories: those targeted at on-line scheduling, where scheduling decisions must be made while tasks are running, so the scheduler must be very fast; and optimal scheduling, where the scheduler is not used on-line, but tries to come up with optimal schedules for the purpose of knowing the best possible schedule, in order to help develop scheduling algorithms [10]. There have been studies carried out into optimal contention-aware scheduling, but not many; optimal algorithms have been produced as well as heuristic approximations to the optimal algorithms [10, 11, 12]. *However, these algorithms require a priori knowledge of performance degradation of tasks when running in various combinations on cores and are relatively slow; any performance improvement in the execution time of such schedulers increases their usefulness, which is what we achieve* [13, 14]. Also, the heuristic algorithms do not produce schedules as good as the optimal algorithms. The current state-of-the-art contention-aware scheduler is the Distributed Intensity (DI) scheduler [6]. It has the best performance (execution time spent by the scheduler) when generating schedules; however, the schedules it produces are not very close to the optimal schedules (most about 10% worse, with our benchmarks). *We combine the on-line and optimal scheduling approaches by developing a scheduler which targets generating optimal schedules while exhibiting fast execution time.* To the best of our knowledge, this approach has not been used before for contention-aware schedulers. We close the gap between optimal schedulers and approximations, by implementing and evaluating optimization and machine learning-based schedulers.

In our study, we implement an optimal contention-aware scheduling solution [14] which we try to equal using less time-intensive processing. The optimal solution uses Integer Programming. This approach assumes knowledge of various performance values for various combinations of core assignments. It serves as a benchmark. We also implement the DI scheduler, as presented in [6], and our own approach, which combines the DI approach with Linear Programming. We implement predictive models based on Machine Learning which, given a set of tasks running on a chip, can predict the degradation in task performance for each task. The results were not so good, compared with DI. We then implemented a scheduler based on another machine learning model, this time a Multilayer Perceptron [15, 16]. To do this, we had to generate workloads for the Optimal Scheduler. Once the Optimal Scheduler did its work, the Multilayer Perceptron was used to learn, for each type of workload, the assignment of tasks to cores. Generating workloads involved analyzing the metrics associated with the workloads available in order to generate new performance figures for new workloads. We then implemented a scheduler with the Multilayer Perceptron model, named MLP, which tries to mimic the decision of the Optimal Scheduler. Our results are very promising. The MLP scheduler is better than DI in 7 out of 9 test workloads, and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads. The MLP scheduler produces schedules very close to the optimal scheduler and much better schedules than DI. Also, it has much faster execution time than the optimal scheduler, and relatively close scheduler execution time to DI.

The rest of this paper is organized as follows: Section 2 discusses related work on scheduling, Section 3 discusses the Optimal Scheduling algorithm and Distributed Intensity (DI), Section 4 discusses our implementation environment, Section 5 describes our Linear Programming-based Scheduler (ModLP), Section 6 describes the Multilayer Perceptron Scheduler (MLP), Section 7 presents the results of our experiments, including a discussion on our workloads, and on the results themselves, and Section 8 concludes.

2. **Related Work.** The closest work to ours is that done by Blagodurov et al. [6]. In their study, the authors investigated the best approaches to classifying threads, based on performance metrics obtained via CPU counters on modern Chip Multiprocessors (CMPs). These include metrics such as Cycles Per Instruction (CPI), Instructions Per Cycle (IPC), and Misses Per One Thousand Instructions (MPI), used to estimate number of cache misses. The classification approaches are used to classify threads when they compete for resources. Based on the classification study, the authors developed a scheduling algorithm known as Distributed Intensity (DI). Their results showed that its performance was close to the Integer Programming-based optimal solution. In our work, we investigate how task degradation could be predicted using machine learning approaches. We then selected the best model for use in a scheduler that combines DI with linear programming, and also to generate workloads to train a Multilayer Perceptron Scheduler.

Rai et al. [17, 18] carried out studies on characterizing and predicting L2 cache behavior. They used machine learning algorithms to build models which could then be used for characterizing and predicting. The machine learning algorithms they used were Linear Regression, Artificial Neural Networks, Locally Weighted Linear Regression, Model Trees, and Support Vector Machines. The methods generate regression models, which involves fitting a model that relates a dependent variable y to some independent variables, $x_1, x_2, \ldots, x_n$. The model is expressed in the form of an equation:

$$y = f(x_1, x_2, \ldots, x_n)$$

The class variable to be predicted was named "solo run L2 cache stress". The attributes (independent variables) used were

i. L2 cache references per kilo instructions retired.
ii. L2 cache lines brought in (due to miss and prefetch) per kilo instructions retired. This shows the stress put by a program on the L2 cache.
iii. L2 cache lines brought in (due to miss and prefetch) per kilo L2 cache lines referenced. This shows the re-referencing tendency of a program.
iv. Fractional L2 cache occupancy of a program. This gives a rough estimate of fraction of space occupied by a program in the L2 cache while sharing with another program.

Rai et al. [19] extended their work by developing a machine learning based meta-scheduler. The used their predictive models mentioned in the previous paragraph to aid scheduling decisions, achieving a 12% speedup over the Linux CFS scheduler. Their meta-scheduler divides tasks into two groups: those with high solo run L2 cache stress and low solo run L2 cache stress. This serves to reduce cache contention and competition for shared resources, thereby improving performance. Our work combines Machine Learning not just to predict degradation in performance, but to assign tasks to cores. We base our solutions on the optimal solution, and achieve schedules as good as, or very close to, the optimal solution. We do not use as many statistics for prediction as Rai et al., making our method compatible with a wider range of CPU architectures.

Apon et al. [20] studied the use of a Stochastic Learning Automata (SLA) for assigning tasks to parallel systems. The automaton keeps track of various possible states (for example, number of tasks executing, number of tasks waiting), possible actions (number

of processors assigned to tasks), a matrix with probabilities of choosing actions from each state, responses on whether an action was "good", "bad", or "neutral", and an updating scheme for updating probabilities in the matrix. Their results showed substantial improvement of the SLA approach, with varying probabilities, compared with a fixed assignment scheme.

Illikkal et al. [21] developed a rate-based approach to resource management of CMPs. They make use of rate-based Quality of Service (QoS) methods to decrease resource contention. They make use of hardware Voltage and Frequency (V/f) Scaling and Clock Modulation/Gating. V/f involves changing the frequency of a core in order to reduce power consumption, while Clock Modulation involves feeding the clock to the processor for short durations. The core is active only for these durations. The time the core is halted translates to reductions in cache space and memory bandwidth requirements of the task running on the core. Tasks on the other cores then experience less contention for the shared resources. These could have higher priority, for example, hence the need for reduced contention. Illikkal et al.'s results showed their approach to be flexible and effective at ensuring QoS.

Hoh [22] made use of a vector-based approach to co-schedule tasks which were dissimilar in their use of CPU resources. Each task had a vector associated with it, which included elements for various CPU performance counters, such as number of all instructions completed, number of floating point instructions, number of special (e.g., graphic) instructions, number of L2 cache cycles, etc. When selecting a task, the task with the greatest vector difference from the average is selected. Their approach was integrated into the Linux Kernel 2.6.26 Completely Fair Scheduler. Performance tests showed that Hoh's approach was slightly worse than the vanilla Linux kernel when only 2 tasks were running, due to his method's overhead, while it was much better than the vanilla kernel when more than 2 tasks were running.

Klug et al. [23] implemented a scheduling system which forces pinning of tasks to specific CPU cores. After a time constant, their autopin system changes the pinning so various pinning combinations are tried. Once the best pinning combination is discovered, the tasks use it until they terminate. Experiments conducted with SPEC OMP benchmarks show substantial improvements over default Linux scheduler pinning behavior. The autopin approach will not be so useful if application behavior changes significantly after an ideal pinning combination has been discovered.

Our work tries to make use of Machine Learning and Optimization to develop an innovative and efficient solution for multicore scheduling, equal to or very close to the optimal, and succeeds.

3. The Optimal Scheduling Algorithm and a Relatively Near-Optimal Scheduling Algorithm.

3.1. Optimal scheduling.

Jiang et al. [14] carried out a study on optimal scheduling of tasks on a multicore system. Their optimal solution is based on Integer Programming. It solves a partition problem, in which n jobs are in $m = \frac{n}{u}$ sets, where m is the number of chips, each having u cores. Each set is as large as the number of cores in a chip. The objective function is

$$\min \sum_{i=1}^{\binom{n}{u}} d(S_i) \cdot x_{S_i}$$

The constraints are

$$x_{S_i} \in \{0,1\}, \quad 1 \le i \le \binom{n}{u};$$

$$\sum_{k:1\in S_k} x_{S_k} = 1; \sum_{k:2\in S_k} x_{S_k} = 1; \ldots; \sum_{k:n\in S_k} x_{S_k} = 1.$$

x_{S_i} is a binary variable, 1 if S_i is one of the sets in the final partition result or 0 if it is not. $|S_i| = u$.

$$d(S_i) = \sum_{j\in S_i} d_{j,S_i-\{j\}}$$

$d(S_i)$ is the sum of degradations of all the tasks in S_i, when they co-run on a single chip, as opposed to running alone on the same chip. Co-run degradations are obtained by measuring the CPI (Cycles Per Instruction) when a task runs alone, and when it runs together with the other tasks in its set. We use the inverse (Instructions Per Cycle) instead. This Integer Programming formulation of the multicore scheduling problem has been shown to be optimal.

3.2. **Relatively near-optimal scheduling.** The algorithm known as DI (Distributed Intensity) gives results relatively close to the Optimal Scheduler, with very low runtime costs [6]. It uses a particular metric for tasks: MPI (Last Level Cache Misses Per One Thousand Instructions). These are measured when a task runs on its own, with no other tasks running on cores on the same chip. Tasks with high MPI are cache intensive, so the idea is to schedule them on chips with cores running other non-cache intensive tasks; basically, to distribute the load. This is done by sorting the tasks to be scheduled in a list based on MPI, then picking the first and last to run on a chip, the second first and second last to run on another chip, etc. In our study, we implement DI and try to beat its performance, and actually succeed.

4. **Implementation Environment.** We carried out all our experiments in an operating system scheduling simulator known as AKULA [7, 13]. AKULA presents various classes in an object-oriented framework which are used to implement custom schedulers and to modify the simulation environment. It has two ways of running simulations. One is bootstrapping, which involves logging performance of benchmarks running in various combinations on a multicore platform. For example, if the architecture being used comprises two sets of four cores each, with the 4 cores sharing resources such as the Last Level Cache (LLC), and there are eight benchmarks, then there are 8 ways of running the workload one benchmark at a time on a set of 4 cores, there are 28 ways of running 2 benchmarks on a set of 4 cores, there are 56 ways of running 3 benchmarks on the set of 4 cores, and there are 70 ways of running 4 benchmarks on a set of 4 cores. All these combinations must be evaluated by running the actual benchmarks and logging performance, using a tool in AKULA called the profiler.

Once the profiler has done its work and a scheduling algorithm has been implemented in AKULA, the scheduler can be tested using bootstrapping, which will keep track of the progress each task makes per clock tick, depending on which other tasks are running with it in the same set of cores sharing critical resources, based on performance data obtained by the profiler. When a task has made up to 100% progress, it terminates. This means that the task has run for enough ticks to simulate its required execution time; the simulator will thus change the status of the task to completed and remove it from the set of tasks still competing for CPU time.

Another way of running a simulation in AKULA is to attach benchmarks tasks to actual hardware cores using an appropriate utility. Thus, the scheduler runs in user space on an

operating system, but is still able to control placement of tasks on cores. In our paper, we make use of the boostrapping approach to evaluating our scheduling algorithms. The architecture we used for our experiments consists of two sets of four cores each, with each set sharing the Last Level Cache.

5. **A Linear Programming-Based Multicore Scheduler.** In order to develop a Linear Programming-based multicore scheduler, we need to have a model which, given some performance figures of tasks, can predict the degradation in performance (a sort of cost function). We experimented with several prediction models, including Multilayer Perceptron, Linear Regression, and M5 prime. The Multilayer Perceptron had a correlation coefficient of 0.7616 (1 would show perfect correlation between predicted output and actual output). The Linear Regression model had a correlation coefficient of 0.8332, and the M5 Prime model had a correlation coefficient of 0.9906. Since the M5 Prime had the best prediction results, it was chosen to implement our own scheduling algorithm, based on Linear Programming. We focus on a specific case of a two-chip platform, with each chip having four cores, giving eight cores in total. The problem is formulated as an Assignment Problem [24, 25], one of assigning tasks to agents (in this case, tasks to cores). Figure 1 illustrates the assignment problem.

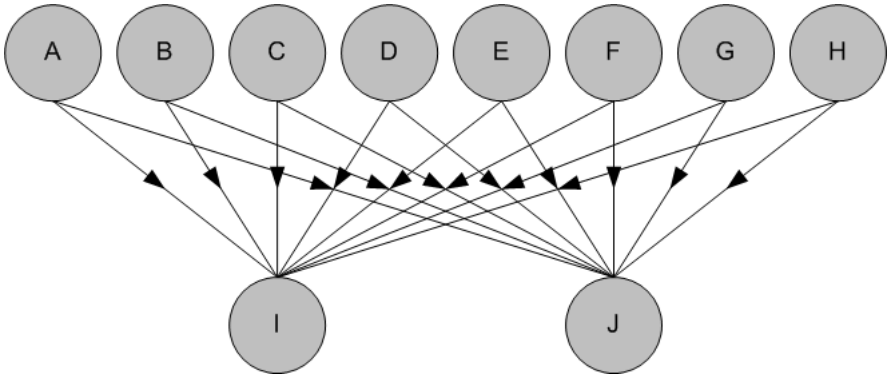

FIGURE 1. An assignment problem graph: Vertices A-H are the tasks, each of which have to be assigned to one of two sets of cores, I and J. Each edge has a weight, which is the cost of assignment to a particular set of cores. In our model, the cost is also dependent on other assignments.

The optimization problem is formulated as follows:

$$\text{Min} \quad \sum_{i=1}^{m}\sum_{j=1}^{n} d(x_{ij})$$

Subject to

$$\sum_{i=1}^{m} x_{ij} = 1, \quad \forall j = 1, 2, \ldots n$$

$$\sum_{j=1}^{n} x_{ij} = 4, \quad \forall i = 1, 2, \ldots m$$

x_{ij} is 1 when task j is scheduled on core i, and is 0 otherwise. $d(x_{ij})$ is the degradation of the task when run in its "home" set. This degradation is a prediction obtained using an M5 Prime model. The first constraint states that the sum of all x_{ij} for a particular j is 1, meaning that any given task is only assigned to one set of cores (there are two sets

of 4 cores each). The second constraint states that the sum of all x_{ij} for a particular i is 4, meaning that a set of 4 cores is assigned exactly 4 tasks.

The home set is obtained by sorting the 8 tasks in the workload according to their MPI (cache Misses Per One Thousand Instructions) metrics. This method was used by Blagodurov et al. in [6] in the DI scheduling algorithm. The reasoning is that tasks with high MPI should be scheduled with tasks with low MPI, since the high MPI tasks are cache-hungry and will get in each others way if scheduled together. When the 8 tasks in the workload are sorted by their MPIs, the 1st and the last can be removed from the sorted set and assigned to cores, until there are no tasks left in the sorted set. This means that the sum of all MPIs is distributed evenly among the chips. Our scheduling system thus combines DI with Linear Programming.

6. **A Multilayer Perceptron-Based Multicore Scheduler.** We borrow an idea from a study carried out by Kirschner on learning from optimizaton models [26]. In his study, he employed the use of machine learning algorithms to learn how optimization methods work, in order to generate rules and models that could then be used to carry out scheduling for processing of chemicals, without the long time delays when regular optimization methods are used. Our work differs in the nature of the problem, multicore scheduling as opposed to chemical process scheduling, the optimization method to be learned from (we use Integer Programming), and also in the Machine Learning tools used; we use M5 Prime and Multilayer Perceptron models while Kirschner used Decision Trees, Nearest Neigbor, Bagging, Boosting, and Winnow. We used M5 Prime to generate artificial workloads to train the Multilayer Perceptron model. The following sequence of steps was used to develop the Multilayer Perceptron multicore scheduler.

i. Use available workload to generate prediction model based on M5 Prime and MPI metric.
ii. Fit log normal distribution to original figures.
iii. Generate new MPI and runtime length figures for new tasks (i.e., new workloads) using the log normal distribution. We generated 165 workloads with 8 tasks each.
iv. Run Optimal scheduler on the new workloads and log the various scheduling decisions (assignment of tasks to cores for each workload).
v. Train Multilayer Perceptron (MLP) on this new core assignment data and save the model. The training data consisted of 1320 instances; each instance specifies how one task is scheduled when seven other tasks are also competing to be scheduled.
vi. Implement scheduler in AKULA simulator to make use of the MLP model to assign tasks to cores for test workload.
vii. Capture various performance metrics.

7. **Results.**

7.1. **Workloads.** The benchmark statistics in the workloads were obtained from the AKULA software release [7]. The actual benchmarks are from the SPEC CPU2006 benchmark suite [27]. We selected 9 benchmarks, based on the availability of performance figures in AKULA, and also since they represent a good mix of shared-resource-intensive and shared-resource-non-intensive applications [28]. They are *gcc, mcf, gobmk, libquantum, gamess, milc, namd, povray* and *lbm*. The 9 benchmarks were combined together in 9 different ways, giving the 9 workloads which we used for our experiments (see Table 1). Each workload has 8 benchmarks. For the purpose of evaluating performance of computer-bound workloads, the same number of tasks are used as there are cores. In actual compute-bound environments, this is normally the case [6].

TABLE 1. Workload composition

#	Constituent Applications
1	gcc, lbm, mcf, milc, povray, gamess, namd, gobmk
2	gcc, lbm, mcf, milc, povray, gamess, namd, libquantum
3	gcc, lbm, mcf, milc, povray, gamess, gobmk, libquantum
4	gcc, lbm, mcf, milc, povray, namd, gobmk, libquantum
5	gcc, lbm, mcf, milc, gamess, namd, gobmk, libquantum
6	gcc, lbm, mcf, povray, gamess, namd, gobmk, libquantum
7	gcc, lbm, milc, povray, gamess, namd, gobmk, libquantum
8	gcc, mcf, milc, povray, gamess, namd, gobmk, libquantum
9	lbm, mcf, milc, povray, gamess, namd, gobmk, libquantum

TABLE 2. Average degradation and unfairness performance figures for various workloads and optimal, DI, ModLP, MLP1 and MLP2 schedulers (best figure is Bold; average degradation not compared with optimal)

	Avg Degradation					Unfairness			
Workload	Optimal	DI	ModLP	MLP1	MLP2	Optimal	DI	ModLP	MLP2
1	15.44928	15.89507	18.14677	15.89507	**15.36548**	209.85335	195.45571	**168.94460**	201.54266
2	9.06622	10.34547	13.64042	10.34547	**9.12833**	133.95346	**132.30041**	147.66707	132.92729
3	21.65493	24.58019	24.59467	24.58019	**21.65493**	171.26594	**168.86807**	169.19926	171.26594
4	21.72809	24.77222	24.61435	24.77222	**21.72809**	170.77428	**167.87780**	169.13020	170.77428
5	21.62507	24.54359	24.60900	24.54359	**21.62507**	171.64460	169.47427	**169.16689**	171.64460
6	16.53802	18.04719	18.55790	18.04719	**16.95042**	**186.27800**	189.81524	170.80955	188.04769
7	16.22186	**16.37744**	18.48113	**16.37744**	18.74328	189.71004	190.12294	**165.65228**	207.81127
8	14.37630	**14.61350**	16.58507	**14.61350**	14.75016	207.00794	205.57945	**176.72302**	207.48043
9	18.19894	20.43654	22.21250	20.43654	**18.19894**	186.29819	187.96851	**156.25516**	186.29819

TABLE 3. Average degradation percentage improvement for DI, ModLP, and MLP2, for all workloads

	Avg Degradation Percentage Improvement			
Workload	DI Over Optimal	ModLP Over DI	MLP2 Over DI	MLP2 Over Optimal
1	–2.89%	–14.17%	**3.33%**	**0.54%**
2	–14.11%	–31.85%	**11.76%**	**–0.69%**
3	–13.51%	**–0.06%**	**11.90%**	**0.00%**
4	–14.01%	**0.64%**	**12.29%**	**0.00%**
5	–13.50%	**–0.27%**	**11.89%**	**0.00%**
6	–9.13%	–2.83%	**6.08%**	–2.49%
7	**–0.96%**	–12.85%	–14.45%	–15.54%
8	–1.65%	–13.49%	**–0.94%**	–2.60%
9	–12.30%	–8.69%	**10.95%**	**0.00%**

7.2. **Schedulers.** We tested the 5 different schedulers described so far in this paper.

i. Optimal. Based on Integer Programming model and knowledge of degradations for all combinations of tasks running on sets of 4 cores on our test platform.
ii. DI. Distributed Intensity scheduler used by Blagodurov et al. [6].
iii. ModLP. DI used as a starting point followed by Linear Programming optimization. It uses an M5 Prime model for prediction.
iv. MLP1. Multilayer Perceptron with 8 input nodes, 2 output nodes (1 for each chip), and 1 hidden layer with 5 nodes. The MLP1 model had an accuracy of 77.35%, a Precision score of 0.777, a Recall score of 0.773, and an F-Measure score of 0.773 on the training/test data. The accuracy is average for two classes (chip 1 and chip 2). Precision indicates out of all workloads in the test set classified to run on chip 0, how many were actually scheduled to run on chip 0 by the optimal scheduler. The same

applies to chip 1. Recall indicates out of all workloads scheduled to run on chip 0 by the optimal scheduler, how many were actually classified like that by the model. The F-Measure score is a combination of the two scores, which tries to find a balance between Precision and Recall [29]. The higher the scores the better, generally.

v. MLP2. Similar to MLP1, but this time the Multilayer Perceptron has only 2 nodes in the hidden layer. The MLP2 model had an accuracy of 67.5%, a Precision score of 0.679, a Recall score of 0.675, and an F-Measure score of 0.673 on the training/test data. The scores for MLP2 are lower than for MLP1 but scores are based on a test set and do not always tell the whole story. On the actual workloads, the MLP2 scheduler made better scheduling decisions.

Table 2 shows the results for the various workloads, for the Optimal, DI, ModLP, MLP1 and MLP2 schedulers, for average degradation and unfairness. The better figures are shown in bold. Optimal would have better average degradation for all workloads, hence is in a separate section for that metric. Degradation is given by:

$$Degradation = 100 \times \frac{(runtime - solotime)}{runtime}$$

Runtime is the time the task takes to run to completion when scheduled with other tasks, while solotime is the time the task takes to runs when scheduled to run alone. Average degradation is the average percentage degradation over all tasks. A lower figure is better. One can see that the MLP schedulers yield better average degradation for all workloads. MLP2 is better in 7 out of 9 workloads and MLP1 is better in 2 workloads.

For future comparisons, MLP2 is selected as the MLP scheduler of choice, given its impressive performance in the average degradation metric.

When it comes to fairness, ModLP is better than the other 3 schedulers (even Optimal) in 5 out of 9 workloads. Because of how the problem is formulated, it does a better job of balancing the load. The unfairness metric captures by how much systems improve performance of certain tasks at the expense of other tasks. A lower number is better. Unfairness is calculated as:

$$Unfairness = 100 \times \frac{\sqrt{\frac{\sum_{i=1}^{N} d_i^2}{N} - \frac{\sum_{i=1}^{N} d_i}{N}}}{\frac{\sum_{i=1}^{N} d_i}{N}}$$

where d_i is the degradation of task i and N is the number of tasks. This metric is closely related to the Coefficient of Variation [30]. We want as little variation in degradation among the various tasks as possible. Since the ModLP scheduler first sorts the tasks as in DI, and then balances the load using Linear Programming, utilizing an M5 Prime prediction model to predict the impact of certain scheduling decisions, it has good unfairness results.

Table 3 gives improvements as percentages; Improvement of Scheduler A over Scheduler B. Bold percentages indicate equal or better performance. Less than one percent difference in the negative direction is also considered equal. One can see that DI is only equal to Optimal in one workload. ModLP is equal to or better than DI in 3 workloads. MLP2 is better than DI in 7 out of 9 workloads. Out of these, 5 are differences of more than 10%. This is a significant improvement over DI. MLP2 is also equal to DI in one workload.

And now, very importantly, MLP2 is equal to Optimal in 6 out of 9 workloads, and strictly equal (0.00%) difference in 4 workloads. This is very significant because it means that Optimal scheduling can be done without going through a time-intensive optimization process such as Integer Programming.

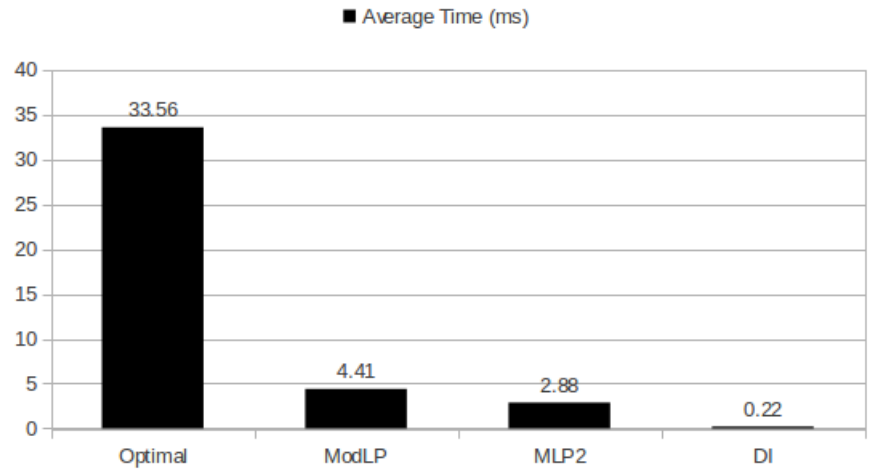

FIGURE 2. Average run times for the 4 approaches we studied

Figure 2 shows the average runtimes of the various schedulers evaluated. These are execution times; how long the scheduler spent doing its work per workload, on average. Note that the focus is on the scheduler, not the whole software system which includes the simulator, and which in a live operating system would include other components such as the memory manager. The Optimal scheduler spent the most time per workload: 33.56ms. This is because of its use of Integer Programming. ModLP is next, taking 4.41ms per workload. It is faster than the Optimal because Linear Programming executes faster than Integer Programming, but it also had to use an M5 Prime model for degradation predictions. MLP2 is next, taking 2.88ms per workload. It is faster than Optimal and ModLP because solving the scheduling problem corresponds to simply plugging in the process MPI figures, after which executing the Multilayer Perceptron involves some matrix multiplications, while Optimal and ModLP have to execute complex Integer and Linear Programming algorithms. The fastest scheduler is DI with 0.22ms spent per workload. DI is very fast because it works with a simple sort mechanism.

7.3. **Significance of results.** The MLP2 scheduler is better than DI, the state-of-the-art scheduler, in 7 out of 9 test workloads, mostly by a wide margin, and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads, exactly equal in 4. The MLP scheduler produces schedules very close to the theoretical optimum and much better schedules than DI. Also, it has much faster execution time than the optimal scheduler, and relatively close scheduler execution time to DI. Given these performance results, MLP2 could be easily used as a batch scheduler [1, 2, 3]. In this scenario, tasks (jobs) are submitted for execution in a non-interactive manner; they run to completion once started, without prompting the user for input. Such systems are found in many High Performance Computing (HPC) environments, where large servers exist with many processing units [31, 32]. Current research is focusing on using variants of DI for this purpose, including cluster environments [32]. These studies have shown that DI is superior to the Linux operating system scheduler, even for cluster environments. Our scheduler outperforms DI, with not much additional cost (in terms of run time), so is well suited for this role, be it on single-machine environments or on cluster nodes. MLP2 could also be modified for integration into an online scheduler, and at various scheduling points the MLP model could be used to reschedule tasks. For example, when a task finishes execution, and when a new task is started. The important issue is that the MLP model should be trained using training sets representative of the actual workloads which will run on the target machines.

8. **Conclusion.** Our results show that the MLP scheduler is better than DI in 7 out of 9 test workloads, and approximately equal to the Optimal Scheduler in 6 out of 9 test workloads. We show that an MLP scheduler can equal the Optimal scheduler in the average degradation metric and significantly outperform the DI scheduling approach (by more than 10% in most cases). Also, an LP scheduler can yield better fairness, thus providing a promising avenue for further research. Both ModLP and MLP are faster than the Optimal scheduler by a wide margin, but out of the two, only MLP can match the Optimal's scheduling results. MLP is slower than DI, but makes up for that with far superior scheduling results.

Acknowledgment. The authors gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

REFERENCES

[1] A. S. Tanenbaum, *Modern Operating Systems*, Pearson Education, Inc., Upper Saddle River, 2009.
[2] A. Silberschatz, P. B. Galvin and G. Gagne, *Operating System Concepts*, John Wiley & Sons, Inc., Hoboken, New Jersey, 2009.
[3] D. M. Dhamdhere, *Operating Systems: A Concept-Based Approach*, McGraw-Hill, 2007.
[4] K.-F. Faxén, C. Bengtsson, M. Brorsson, G. Håkan, E. Hagersten, B. Jonsson, C. Kessler, B. Lisper, P. Stenström and B. Svensson, *Multicore Computing – The State of the Art*, Swedish Institute of Computer Science, 2008.
[5] T. Nowotny, M. K. Muezzinoglu and R. Huerta, Bio-mimetic classification on modern parallel hardware: Realizations on NVIDIA® CUDA™ and OpenMP™, *International Journal of Innovative Computing, Information and Control*, vol.7, no.7(A), pp.3825-3837, 2011.
[6] S. Blagodurov, S. Zhuravlev and A. Fedorova, Contention-aware scheduling on multicore systems, *ACM Trans. on Computer Systems*, vol.28, no.4, pp.1-45, 2010.
[7] S. Zhuravlev, S. Blagodurov and A. Fedorova, AKULA: A Toolset for experimenting and developing thread placement algorithms on multicore systems, *Proc. of the 19th Intl. Conference on Parallel Architectures and Compilation Techniques*, Vienna, Austria, pp.249-259, 2010.
[8] E. Frachtenberg and U. Schwiegelshohn, New challenges of parallel job scheduling, in *Job Scheduling Strategies for Parallel Processing*, E. Frachtenberg and U. Schwiegelshohn (eds.), Berlin, Springer Berlin/Heidelberg, 2008.
[9] S. Siddha, V. Pallipadi and A. Mallick, Process scheduling challenges in the era of multi-core processors, *Intel Technology Journal*, vol.11, no.4, pp.361-369, 2007.
[10] K. Tian, Y. Jiang, X. Shen and W. Mao, Optimal co-scheduling to minimize Makespan on chip multiprocessors, *Proc. of the 16th Workshop on Job Scheduling Strategies for Parallel Processing*, New York, NY, USA, 2012.
[11] Y. Jiang, X. Shen, J. Chen and R. Tripathi, Analysis and approximation of optimal co-scheduling on chip multiprocessors, *Proc. of the 17th Intl. Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, pp.220-229, 2008.
[12] K. Tian, Y. Jiang and X. Shen, A study on optimally co-scheduling jobs of different lengths on chip multiprocessors, *Proc. of the 6th ACM Conference on Computing Frontiers*, New York, NY, USA, pp.41-50, 2009.
[13] S. Zhuravlev, *Designing Scheduling Algorithms for Mitigating Shared Resource Contention in Chip Multicore Processors*, Master Thesis, Simon Fraser University, 2011.
[14] Y. Jiang, K. Tian, X. Shen, J. Zhang, J. Chen and R. Tripathi, The complexity of optimal job co-scheduling on chip multiprocesors and heuristics-based solutions, *IEEE Trans. on Parallel and Distributed Systems*, vol.22, no.7, pp.1192-1205, 2011.
[15] C. M. Bishop, *Pattern Recognition and Machine Learning*, 1st Edition, Springer Science+Business Media, LLC, New York, USA, 2006.
[16] G. T. Shobha and S. C. Sharma, Knowledge discovery for large data sets using artificial neural network, *International Journal of Innovative Computing, Information and Control*, vol.1, no.4, pp.635-642, 2005.
[17] J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, Characterizing L2 cache behavior of programs on multi-core processors: Regression models and their transferability, *World Congress on Nature and Biologically Inspired Computing, NaBIC 2009, IEEE*, Coimbatore, pp.1673-1676, 2009.

[18] J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, On prediction accuracy of machine learning algorithms for characterizing shared L2 cache behavior of programs on multicore processors, *Proc. of the 1st Intl. Conference on Computational Intelligence, Communication Systems and Networks*, Indore, India, pp.213-219, 2009.

[19] J. K. Rai, A. Negi, R. Wankar and K. D. Nayak, A machine learning based meta-scheduler for multicore processors, *Intl. Journal of Adaptive, Resilient and Autonomic Systems*, vol.1, no.4, pp.46-59, 2010.

[20] A. W. Apon, T. D. Wagner and L. W. Dowdy, A learning approach to processor allocation in parallel systems, *Proc. of the 8th Intl. Conference on Information and Knowledge Management*, New York, NY, USA, pp.531-537, 1999.

[21] R. Illikkal, V. Chadha, A. Herdrich, R. Iyer and D. Newell, PIRATE: QoS and performance management in CMP architectures, *SIGMETRICS Performance Evaluation Review*, vol.37, pp.3-10, 2010.

[22] E. Hoh, *Vector-Based Scheduling for the Core2-Architecture*, Study Thesis, System Architecture Group, University of Karlsruhe, Germany, 2009.

[23] T. Klug, M. Ott, J. Weidendorfer and C. Trinitis, Autopin – Automated optimization of thread-to-core pinning on multicore systems, *Trans. on High-Performance Embedded Architectures and Compilers III, LNCS*, vol.6590, pp.219-235, 2011.

[24] E. Çela, Assignment problems, in *Handbook of Applied Optimization*, P. M. Pardalos and M. G. C. Resende (eds.), New York, Oxford University Press, Inc., 2002.

[25] D. W. Pentico, Assignment problems: A golden anniversary survey, *European Journal of Operational Research*, vol.176, no.2, pp.774-793, 2007.

[26] K. J. Kirschner, *Empirical Learning Methods for the Induction of Knowledge from Optimization Models*, Ph.D. Thesis, Georgia Institute of Technology, 2000.

[27] J. L. Henning, SPEC CPU2006 benchmark descriptions, *SIGARCH Computer Architecture News*, vol.34, no.4, pp.1-17, 2006.

[28] S. Blagodurov, S. Zhuravlev, S. Lansiquot and A. Fedorova, Addressing cache contention in multicore processors via scheduling, *Tech. Rep. TR 2009-16*, Simon Fraser University, Burnaby, BC, Canada, 2009.

[29] M. Sokolova and S. Szpakowicz, Machine learning in natural language processing, in *Handbook of Research on Machine Learning Applications and Trends: Algorithms, Methods, and Techniques*, E. S. Olivas, J. D. M. Guerrero, M. M. Sober, J. R. Magdalena and A. J. S. López (eds.), Hershey, New York, Information Science Reference, 2010.

[30] R. Jain, *The Art of Computer Systems Performance Analysis*, John Wiley and Sons, Inc., 1991.

[31] D. Feitelson, L. Rudolph and U. Schwiegelshohn, Parallel job scheduling – A status report, in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph and U. Schwiegelshohn (eds.), Springer Berlin/Heidelberg, 2005.

[32] S. Blagodurov and A. Fedorova, In search for contention-descriptive metrics in HPC cluster environment, *Proc. of the Second Joint WOSP/SIPEW Intl. Conference on Performance Engineering*, New York, NY, USA, pp.457-462, 2011.