Adaptive OpenMP Task Scheduling Using Runtime APIs and Machine Learning

Ahmad R. Qawasmeh, Abid M. Malik, Barbara M. Chapman
Department of Computer Science
University of Houston
Houston, Texas
Email: {arqawasm, ammalik3, chapman}@cs.uh.edu

Abstract—Task-based programming models adopt different scheduling strategies to exploit parallelism in irregular applications. These scheduling strategies differ in terms of exploiting data locality, maintaining load balance, and minimizing overhead. OpenMP tasks allow programmers to express unstructured parallelism at a high level of abstraction and make the runtime responsible about the burden of scheduling parallel execution. For irregular applications, the performance of task scheduling cannot often be predicted due to the nature of application, the used compiler, and the platform/architecture dependencies. In this work, we introduce an automatic, portable, and adaptive runtime feedback-driven framework (APARF) that combines standard low-level tasking runtime APIs, a developed profiling tool, and a hybrid machine learning model. We employ APARF to select the optimum task scheduling scheme of any given application using similarity analysis through the correlation between the captured runtime APIs with low profiling costs. Our hybrid model predicts the best scheduling strategy for a variety of unseen applications with an average accuracy of 93%, while maintaining a 100% training accuracy. An average performance enhancement of 25% was obtained compared with the default configuration, when APARF was applied on different unseen programs. APARF was examined against a real application (Molecular Dynamics), where we achieved up to 31% performance improvement. Compared to Intel, PGI and GNU compilers, our predicted scheme achieved better performance in most cases.

Index Terms—Adaptive scheduling; Machine learning; OMPT; Collector APIs; OpenMP Tasks; Similarity analysis

I. INTRODUCTION

OpenMP is a standard directive-based API for parallel shared memory programming. The introduction of tasks in OpenMP allows applications exhibiting irregular parallelism in the form of recursion and pointer based data structures to be parallelized in a directive-based manner. The task construct allows the developer to dynamically create asynchronous units of work to be scheduled by the runtime, while task synchronization is provided by additional tasking constructs.

An OpenMP task scheduler can be distinguished based on the implementation of queue organizations used to schedule and synchronize tasks among threads, work-stealing capability, and queue contention. Two crucial issues, related to tasks, should be managed by any OpenMP task scheduler: 1) Data locality 2) Load balancing. OpenMP tasks, accessing the same data blocks, might be assigned to the same thread using a greedy approach to exploit better locality. However, this approach might result in incapable work-stealing. Equitable distribution of work among threads is necessary to prevent processors from being idle. However, this equal distribution is hard to achieve in many cases. Synchronization among threads is one factor, which leads to load-imbalance. Load balancing operations are applicable, but might produce overhead costs due to high latency remote memory accesses between processors. As a consequence, understanding the aforementioned conflicting goals requires an accurate profiling mechanism to obtain low-level runtime information about applications, scheduling schemes, and an increasingly complex memory hierarchy that includes shared caches and non-uniform memory access (NUMA) characteristics.

Enhancing OpenMP scheduling is a topic that has been repeatedly investigated over the years [1] [2] [3] [4]. However, most existing approaches are either limited to loop parallelism, restricted in their portability, incur high overheads or affect compiler optimizations. In this work, we take a completely different approach for improving task scheduling. We introduce a portable feedback-driven framework that integrates a runtime system with a tool-chain that consists of an external profiling tool and a hybrid machine learning model. This integration provides a mechanism for adaptive task scheduling across two different platforms (Intel and AMD) while focusing on programs exhibiting unstructured behavior. Using a standard OpenMP Tools API, OMPT [5], low-level runtime events are captured by our developed tool for a set of benchmarks and are passed to a machine learning model to build a trained classifier. The trained classifier is then used to predict the optimum scheduling configuration for new unseen programs and the classifier performance is evaluated against the default OpenMP runtime strategy and against other compilers.

Figure 1 demonstrates that selecting the correct task scheduling configuration has a significant impact on performance. Sort and Health benchmarks (from the Barcelona OpenMP Task Suite [6]) are shown. Our main motivation lies in predicting the optimum task scheduling scheme for a given program by presenting an adaptive and portable solution.

This paper is organized as follows. Section II discusses the related work. Section III describes the components of our framework. Section IV explains our prediction methodology. Section V presents the experimental results and shows a detailed analysis. Finally, Section VI concludes our contributions and discusses directions for the future work.
II. RELATED WORK

Rather than proposing a new task scheduling scheme, this paper aims to construct an adaptive, and portable framework that selects the optimal scheduling scheme out of existing schemes implemented in different compilers and runtime systems via standard runtime APIs proposed in OpenMP. Load balancing is a crucial issue that affects OpenMP performance. An adaptive data structure [7] was proposed in the runtime for balancing tasks execution by dynamically moving tasks between resources. A dynamic scheduling strategy was proposed in [8] coupled with an analytical model that resizes chunks to prevent under-utilization and load unbalance of CPUs and GPUs. A hybrid task scheduling strategy [9] was proposed that outperforms existing work-first and breadth-first strategies. An adaptive task creation strategy [10], AdaptiveTC, was proposed to support load-balancing via effective work-stealing schemes. Sato et al. [11] introduced an efficient work-stealing strategy, in which tasks are stolen by chunks from the bottom of a victim stack. In case of task synchronization, LIBKOMP [12] is an OpenMP runtime system that expresses dependencies between tasks to prevent unnecessary synchronization. Another work [13] reduced task scheduling overheads by using a cut-off technique to improve performance.

Regarding adaptive loop scheduling, Durand et al. [1] presented an OpenMP loop scheduler which adapts the granularity of parallel loops depending on data placement. Another work [2] presented a thread scheduling policy for irregular OpenMP programs based on thread affinities. This work was implemented before the introduction of tasks in OpenMP and hence is not aware of task scheduling adaptation. Jung et al. [3] presented an approach to disable threads in parallel loops in the presence of contention. A machine learning approach [4] was used to predict the best loop scheduling strategy and scalability based on source to source instrumentation measurements, which affect compiler optimizations. Thoman et al. [14] presented a loop scheduling policy for deriving the optimal loop schedule for a given loop based on compiler analysis. This technique was different than other pure runtime solutions [15] proposed to enhance loop scheduling.

III. ADAPTIVE RUNTIME FEEDBACK-DRIVEN FRAMEWORK (APARF)

This Section and the following one give a detailed explanation about the components of our adaptive framework (APARF) and demonstrate the process of feedback adaptation. The overall picture of APARF is depicted in Figure 2.

Fig. 2: APARF framework

A. Runtime Support for OpenMP Tasks

Our framework is built on the OpenUH [16] compiler, which supports OpenMP 3.1, and was evaluated against Intel, PGI and GNU compilers. OpenUH offers a configurable OpenMP task pool framework that allows users to control task creation and scheduling at runtime through a set of environment variables. We currently have four different task pools implemented that utilize global, distributed, hierarchical, and hybrid queue organizations. Our task configurations employ both depth-first and breadth-first schedulers. In the following, we explain the task configurations used in our analysis:

- **O64_OMP_TASK_POOL**: Gives control over the organization of task queues, which have an influence on efficiency, queue contention and work-stealing. Four different options are available. Using DEFAULT, each thread has two queues; One for tied tasks and another for untied tasks. Using SIMPLE, each thread has one queue which holds any unscheduled tied or untied tasks that it creates. Using SIMPLE_2LEVEL, each thread has a fixed-size queue for the tasks it creates. A global queue can be used when the private queue is filled. In PUBLIC_PRIVATE, each thread has a public and private queue. An environment variable, O64_OMP_TASK_POOL_GREEDVAL, is used to control how many tasks are placed in each queue. Two values were used: One means all tasks are placed in public queue and available for stealing. Two offers a different approach, where half of the created tasks are placed in each queue. A value of 128 is used in our experiments as the size of task queues.
- **O64_OMP_TASK_QUEUE**: Influences how tasks are scheduled. Five types of queues are used: DEQUE puts task to tail of queue, gets task from tail of queue, steals task from head of queue, donates task to head of queue. FIFO puts and donates tasks to tail of queue, gets and
steals tasks from head of queue. CFIFO allows concurrent access to head and tail of a given queue. LIFO puts and donates tasks to tail of queue, gets and steals tasks from tail of queue. INV_DEQUE is similar to DEQUE, however it gets task from head of queue and steals task from tail of queue.

- **O64_OMP_QUEUE_STORAGE**: Specifies low-level control of the queue API provided by the runtime. Four options were used: ARRAY, DYN_ARRAY, LIST, and LOCKLESS (like array, except no locks are used).
- **O64_OMP_TASK_CHUNK_SIZE**: Controls the number of tasks that may be stolen at once from a queue. Two values were used: one and two.

The default task runtime scheme has a DEFAULT task pool, ARRAY storage, DEQUE task queue, chunk size of 1, size of queue 128, and disabled GREEDVAL.

B. OMPT and ORA OpenMP Tools APIs

OMPT [5] is a new standard OpenMP Tools Application Programming Interface (API) that enables portable tools to collect performance analysis information of OpenMP programs. The design of OMPT takes advantage of two prior OpenMP tools APIs: the POMP API [17] and the Collector API (ORA) [18]; hence OMPT supports trace-based measurements and asynchronous sampling provided in the POMP API and the Collector API respectively. Both ORA and POMP APIs were proposed before the introduction of OpenMP tasks. An open source implementation [19] of the ORA was developed in the OpenUH runtime library. We recently proposed new APIs to the ORA implementation in OpenUH to support OpenMP tasks. The new APIs cover task creation, scheduling, suspension, switching, execution and completion. A full description of these new APIs can be found in [20]. We adapted the ORA implementation including our new task APIs to be compatible with OMPT.

C. A Profiling Tool

We developed a profiling tool that implements a single handler to handle all events. Our tool can interact with the OpenMP runtime through the API calls implemented in the OMPT. It requests notification of a specific event by passing the name of the event to be tracked as well as a callback function to be invoked by the OpenMP runtime each time the event occurs. The functionality of the callback functions, associated with these events, is controlled by the tool. When any master or worker thread enters any OpenMP region, it fires an event which is processed by our tool. The event contains a unique ID for the parallel region which is used to resolve the context of the worker thread. Regarding tasks, the task instances called from the same calling context are aggregated to reduce overhead costs. We integrated PAPI [21] hardware counters (HWCs) into our tool to automate the process of measuring these HWCs beside execution times for the encountered events. In this way, we are able to thoroughly analyze different task configurations against a variety of OpenMP task benchmarks and applications. This analysis is performed through capturing and visualizing the new proposed task events as well as the other encountered events. Our tool can interact with any runtime supporting OMPT or ORA.

IV. HYBRID PREDICTIVE MODELING

While it is possible to collect various kinds of performance measurements from running programs, such data are overwhelming for human processing; hence, we rely on machine-learning techniques that include data pre-processing, clustering, and classification for constructing our prediction model.

A. Programs for Training

Selecting a set of representative programs is crucial for building an effective classifier. To train our classifier, we used eight BOTS benchmark applications presented in [6]. These task-based applications are: Sort, Fibonacci, Health, Strassen, Sparse, Alignment, FFT, and Nqueens. According to [6], these benchmarks vary in terms of domain, computation structure, compute/memory bound, # of tasks, nested tasks, and task granularity. This is crucial for a machine learning model to comprise all aspects.

Each program runs for two input sizes, each with three thread numbers (2, 12, 24), and also with four compiler optimizations (-O0, -O1, -O2, -O3) on two different platforms. The thread numbers were chosen carefully based on the number of cores per socket and the maximum number of available cores per node. The total number of data instances is 192 on each platform. We obtained the average of 5 runs per instance in both training and predictions phases. Running these instances against all possible task scheduling configurations (several hundreds in our case) gave us an initial observation that these applications can be categorized into three groups based on their performance behavior. Programs in each group share the same optimal task configuration.

In order to show that our approach is portable across different multi-cores systems, we targeted two platforms that vary in terms of processor, speed, #cores, hyper-threading availability, instruction set, governors on the cores, memory speed and bandwidth, cache size, and processors connectivity.

The first is an x86-64 cc-NUMA Linux system with a four 2.2 GHz 12-core AMD Opteron processors (48 cores total) and 512 KB L2 cache per core, and 10 MB L3 cache shared by all cores. The second is an x86-64 cc-NUMA Linux system with two 2.5 GHz 12-core Intel Xeon processors (24 cores total) and 512 KB L2 cache per core, and 15 MB L3 cache shared by all cores.

B. Identification of Performance Events

Since we are targeting task scheduling performance, our model must include all events that may be encountered by any OpenMP program. These events include task creation, execution, suspension, completion, parallel region, implicit/explicit barrier, and loop/single/master region low-level runtime events proposed in the OMPT/ORA APIs. These 7 events clearly express task scheduling performance while excluding the portions of applications not affected by scheduling policies.
Two crucial issues, related to tasks, should be managed by any OpenMP task scheduler: 1) Data locality 2) Load balancing. Regarding load balancing, we found that threading measurements collected by our tool and spent by each running thread on each of the 7 runtime events can best describe this issue. As far as data locality, we focused on hardware performance event counts corresponding to cache behavior that include total cache misses, total cache accesses, CPI, and total TLB misses. We chose level 2 cache as it is the last level of private caches and reflects the performance of each thread per core. Next, for each candidate event, we ran each instance in the three optimal scheduling configurations obtained based on our initial observation and noted the event counts to see if the variation between the different configurations is significant. We calculated the mean and maximal deviation for each event. Cache misses gave us the highest level of variation and hence, we chose to compute the cache miss rate for each event to normalize the collected data. We ended up having 14 timing and cache miss rate features, while the combined total number of instances on both platforms is 384.

C. Model Construction

In order to make the basis of our methodology more solid, we consulted the unsupervised K-Means clustering algorithm using the identified set of performance features. We ran each instance against the default OpenMP task scheme in OpenUH and obtained the performance features. We performed this step to ensure that our problem is a three-way classification as obtained by our initial observation. We developed a machine learning Java tool based on Weka [22] API. Different machine learning tools only give the resulting clusters with the centroids without specifying the clusters to which each instance belongs. Our Java code allows us to show that. The cluster obtained for each instance was exactly the same as our initial observation. This shows that our identification of features was accurate. We decided to name each one of the three classes by the optimal task pool configuration obtained for its programs.

- simple (0), where pool is SIMPLE, storage is ARRAY, queue is LIFO, chunk size is 2, and size of queue is 128
- public (1), where pool is PUBLIC_PRIVATE, storage is ARRAY, GREEDVAL is 2, queue is DEQUE, chunk size is 1, and size of queue is 64
- default (2), where pool is DEFAULT, storage is DYN_ARRAY, queue is INV_DEQUE, chunk size is 2, and size of queue is 128.

Then, we manually labeled (classified) each training data instance by adding the corresponding class obtained from K-Means as a separate field. After that, we integrated four different classification techniques in our machine learning code that include Naive Bayes, Multi-layer Perceptron Artificial Neural Network (ANN), multi-class Support Vector Machine (SVM), and Random Forest decision tree. We used 8-fold cross-validation, which partitions our original training set on each platform into 8 subsets of 24 instances. This means that all instance are used for both training and validation, and each instance is used for validation exactly once. Perceptron ANN produced the best classification results with 100% accuracy, while the other algorithms produced over 95%. For this reason and also because of its proven success in modeling both linear and non-linear regression problems, and it is robust to noise, we decided to choose Perceptron ANN in the prediction phase. Reader can refer to [22] to have more information about these classification and K-Means clustering methods.

Although we ended up reducing our search-space from hundreds of classes to three classes only, running each unseen program using the three optimal task configurations still takes way more time than applying its data set to the training data for predicting the best scheduling configuration, especially when we use large input sizes, which, in some cases, may take hundreds of seconds in the worst case.

The confusion matrix resulted from the Perceptron classifier is shown in Table I and was exactly the same on both platforms.

<table>
<thead>
<tr>
<th>Actual class</th>
<th>Predicted class</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>92  0  0</td>
</tr>
<tr>
<td>public</td>
<td>0   52 48</td>
</tr>
<tr>
<td>default</td>
<td>0   0   0</td>
</tr>
</tbody>
</table>

Our measurements on both platforms showed that varying the optimization flags has a negligible impact on performance. Table II illustrates the average improvement, against the default runtime scheme implemented in OpenUH, for each program, which represents 24 instances, based on the classes obtained after training our classifier on both platforms. In general, the execution time and improvement percentage on Intel were better than AMD. All instances in Fibonacci and Health benchmarks were classified in the same representative group and shared the same optimal scheduling scheme, which is simple (0). Sort, FFT, and Nqueens instances were placed in the same class as Fib, except when 24 threads were used. These benchmarks in general create a massive amount of very fine-grained tasks, which produce large task creation and suspension times. In such programs, data locality should efficiently be exploited to reduce the generated overheads. A simple task pool configuration exploits data locality by allocating one queue to each thread. As a result, tasks working on the same block of code will have higher chances to be placed in the same queue. Tasks in this predicted/best configuration are scheduled in a LIFO order to allow a depth-first execution of tasks. This execution exploits data locality in the best manner. The remaining 8 instances of Sort, FFT, and Nqueens were classified as public (1). Knowing that these benchmarks produce less number of tasks compared with Fibonacci, increasing the number of threads will result in high queue contention; hence, it is more efficient to use a private queue per thread to reduce contention. For Strassen, the 24 instances were public (1).

For Alignment and Sparse, all instances were default (2). Work stealing is crucial in such benchmarks to maintain better load-balancing for the coarse-grained tasks. Using the
DEFAULT task pool, a thread has more freedom to steal work (from the untied queues) when it has tasks tied to it that are not suspended in a barrier. Time spent on task execution was significant. This means that synchronization overheads among threads are negligible.

**TABLE II: Training data improvement**

<table>
<thead>
<tr>
<th>Test (AMD)</th>
<th>Default (AMD)</th>
<th>Test (Intel)</th>
<th>Default (Intel)</th>
<th>Improvement (AMD)</th>
<th>Improvement (Intel)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fib</td>
<td>7.9</td>
<td>9.4</td>
<td>7.5</td>
<td>8.5</td>
<td>20%</td>
</tr>
<tr>
<td>Health</td>
<td>190</td>
<td>188</td>
<td>160</td>
<td>160</td>
<td>10%</td>
</tr>
<tr>
<td>Sort</td>
<td>12.7</td>
<td>16.2</td>
<td>13.5</td>
<td>15.5</td>
<td>23%</td>
</tr>
<tr>
<td>FFT</td>
<td>5.9</td>
<td>7.7</td>
<td>6.5</td>
<td>12.5</td>
<td>33%</td>
</tr>
<tr>
<td>Neighbors</td>
<td>29</td>
<td>27.5</td>
<td>31.5</td>
<td>31.5</td>
<td>5%</td>
</tr>
<tr>
<td>Strassen</td>
<td>13.5</td>
<td>14.5</td>
<td>13.4</td>
<td>14.4</td>
<td>7%</td>
</tr>
<tr>
<td>Sparse</td>
<td>14.3</td>
<td>14.3</td>
<td>12.3</td>
<td>12.3</td>
<td>4%</td>
</tr>
</tbody>
</table>

Figure 3 exhibits the behavior of the training programs after normalizing the time measurements with one data size, while two threads and -O3 flag were used. This Figure shows a similar correlation between the captured performance events for the programs located in the same class.

**TABLE III: Unseen programs for prediction**

<table>
<thead>
<tr>
<th>Program</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTS</td>
<td>An unbalanced tree search benchmark that benefits from dynamic load balancing techniques [6].</td>
</tr>
<tr>
<td>Floorplan</td>
<td>Computes the optimal floorplan distribution of a number of cells [6].</td>
</tr>
<tr>
<td>EPCC</td>
<td>An OpenMP task micro-benchmark used to measure task and taskwait overheads [23].</td>
</tr>
<tr>
<td>Whetstone</td>
<td>A synthetic benchmark for evaluating the performance of computers in terms of power [24].</td>
</tr>
<tr>
<td>MD</td>
<td>A real scientific application that implements a simple molecular dynamics simulation, using the velocity Verlet time integration scheme. We implemented a task-based version for this application [25].</td>
</tr>
</tbody>
</table>

**B. Evaluation results**

In order to verify the importance of our prediction framework, we compared our predicted scheduling scheme against the default runtime scheme implemented in OpenUH, Intel 15.0, PGI 15.5, and GCC 4.8. Out of 120 unseen instances, only 8 instances were wrongly classified, where 4 of them belong to the MD application, and the other 4 to the EPCC benchmark. The wrong predicted instances in both MD and EPCC represented the smaller input size, while 24 threads were used. Our experimental results showed that the performance measurements of these instances were unstable and inconsistent not only with OpenUH, but also with the other compilers. The size of input data and thread mapping to multi-cores across different sockets were the main reason, although we tried different thread affinity policies. The prediction accuracy of 93% ensures that our approach is reliable.

Table IV shows the classification results of the unseen programs and the average predicted time of all instances per program in seconds on Intel and AMD platforms. The average times on Intel Xeon processor platform outperformed those obtained on AMD Opteron platform. Except for Floorplan, all instances per program were located in the same class.

One major observation obtained from Figure 5 is the stability of our prediction methodology across different platforms. The simple scheduling scheme was dominant among the unseen programs. This emphasizes the impact of task granularity on the performance of task scheduling. GCC execution times
were the worst among the tested compilers. Our predicted scheme surpassed the other compilers in most cases.

<table>
<thead>
<tr>
<th>Program</th>
<th>Class</th>
<th>Average predicted time in sec (Intel)</th>
<th>Average predicted time in sec (AMD)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UTS</td>
<td>24 public(1)</td>
<td>7.5</td>
<td>10.5</td>
</tr>
<tr>
<td>Floorplan</td>
<td>16 simple(0), 8 default(2)</td>
<td>2.4</td>
<td>2.9</td>
</tr>
<tr>
<td>EPCC</td>
<td>24 public(1)</td>
<td>11</td>
<td>17.6</td>
</tr>
<tr>
<td>Whetstone</td>
<td>24 simple(0)</td>
<td>11.2</td>
<td>17.3</td>
</tr>
<tr>
<td>MD</td>
<td>24 simple(0)</td>
<td>182</td>
<td>235</td>
</tr>
</tbody>
</table>

To have better understanding of the results, we analyze two programs that were differently classified, as shown in Figure 6. Our task-based implementation of MD targeted the computation phase, which is the most intensive portion of MD. A single construct was specified in the computation loop to ensure there is no redundancy in creating tasks among threads. A task is created for each nested loop and taskwait is defined at different levels of nested parallelism to ensure correctness. This task implementation results in creating millions of tasks that cause high queue contentions and synchronization overheads. In such fine-grained task application, load balancing is not a big concern, and hence work stealing should not be a bottleneck. A depth-first execution must be used while preventing stealing from other threads to better exploit data locality and relieve queue contentions. The simple scheduling scheme was the predicted optimal solution to handle these scheduling issues for the MD application. It gave about 30% of improvement over the default scheme of OpenUH, and was also better than the other compilers except Intel, as shown in Figure 5. Figure 6a shows the stability maintained across both platforms and demonstrates the synchronization issue exhibited by MD when the default scheme was used.

The Unbalanced Tree Search (UTS) consists of two task directives encapsulated by a single region and implies nested tasks. UTS presents significant load imbalance problems to any scheduling scheme while also exhibiting a medium level of task granularity on average, which requires some locality to exploit. A work sharing scheme can alleviate the load imbalance issue, but at the expense of data locality, queue contention, and synchronization. Hence, our classifier selected the public task pool configuration, in which tasks are equally distributed among public and private queues per each thread. As shown in Figure 5, the average improvement gained was close to 30% against the default scheme of OpenUH, GCC, and PGI on both platforms. However, Intel performance was better. Figure 6b demonstrates the stability maintained across both platforms while varying the number of threads.

EPCC, Whetstone, and Floorplan produced a very good improvement, when the predicted scheme was applied, as seen in Figure 5. For these three benchmarks, our predicted scheme was better than all other tested schemes including the Intel compiler. The behavior of these programs instances was similar to the other instances located in the same classes.

C. Overhead cost

The introduction of the new OMPT APIs in the runtime produces an average overhead of less than 1%, as shown in our previous work [20]. Our tool has a profiling overhead of 25% on the Intel platform and 23% on the AMD platform, while capturing both timing and HWCs measurements together. Compared to tracing performance tools, our approach produced lower overheads and does not require any source code modification or instrumentation.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented an automatic, portable, and adaptive runtime modeling framework for enhancing the performance of any given OpenMP task-based program. By capturing low-level standard runtime events implemented in both the OMPT and ORA APIs as features, our modeling approach predicts the optimum scheme for scheduling an unseen program on the underlying hardware. In contrast to other adaptive scheduling approaches, our approach improves the performance of programs exhibiting irregular parallelism across multiple platforms by demonstrating the correlation between low-level runtime events that best examine task scheduling schemes. We evaluated the performance of our predicted scheme by comparing it to three state-of-the-art OpenMP scheduling schemes implemented in the Intel, PGI and GNU compilers, as well as the default OpenMP runtime scheme of OpenUH. Our experimental results on two different multicore platforms (Intel Xeon and the AMD Opteron processors) showed that our predicted scheme produces the best performance beside the Intel scheme on average and also maintains a great stability across programs, compiler optimizations, data sets, thread sets and architectures.

Path forward, we plan to strengthen the usefulness of our feedback-driven framework by predicting the power and energy consumption behavior for unseen programs. Power and energy sensors will be integrated in our tool to capture measurements relevant to what we have presented in this work.

ACKNOWLEDGMENT

Many thanks go to our colleagues in the HPCTools group. This work is supported by the National Science Foundation under grants CCF-1148052 and CRI-0958464.

REFERENCES


TABLE IV: Unseen programs classification
Fig. 5: Performance improvement percentage of our predicted scheduling scheme compared to the default runtime scheme in OpenUH and the task scheduling schemes implemented in GCC, PGI, and Intel compilers

Fig. 6: Correlation measurements obtained for MD and UTS


