Abstract

Matching program parallelism to platform parallelism using thread selection is difficult when the environment and available resources dynamically change. Existing compiler or runtime approaches are typically based on a one-size fits all policy. There is little ability to either evaluate or adapt the policy when encountering new external workloads or hardware resources. This paper focuses on selecting the best number of threads for a parallel application in dynamic environments. It develops a new scheme based on a mixture of experts approach. It learns online which, of a number of existing policies, or experts, is best suited for a particular environment without having to try out each policy. It does this by using a novel environment predictor as a proxy for the quality of an expert thread selection policy. Additional expert policies can easily be added and are selected only when appropriate. We evaluate our scheme in environments with varying external workloads and hardware resources. In all cases, this work outperforms existing schemes and surprisingly improves workload performance.

1. Introduction

We now live in a world where, across the spectrum, hardware platforms are parallel and diverse, ranging from mobiles to the cloud. In the past, parallelism was restricted to HPC environments running a single application at a time with fixed, known resources. This is no longer the case, mainstream applications have to share dynamically varying resources.

Matching program parallelism to platform parallelism is a real challenge for compilers when the environment is shared, dynamic and unknown at compile time. Runtime systems such as [1, 16, 19] can overcome this, but are program agnostic and slow to react. In this paper we focus on one area of parallelism mapping, selecting the best number of threads for a parallel program. It is the key decision when reconciling program parallelism with available resources. There has, in fact, been significant work from the compiler and runtime communities in improving workload-aware thread selection.

All approaches are characterised by a one-size fits all assumption. They have a single monolithic model or policy that matches a program to its parallel environment. There is little ability to examine whether the policy fits the current setting or whether another would perform better. No matter how parameterized the policy is, it is highly unlikely that a scheduling policy developed today will always be suited for tomorrow. One critical problem with current approaches is that they cannot be easily updated or extended. Adding additional expertise requires expensive rewriting (or retraining) the policy. Furthermore, improving one of the policy heuristics may adversely affect others.

Our paper develops a new approach based on predictive modeling that considers a number of thread selection policies (experts) at runtime and selects the one that it believes will perform best at every parallel loop. As the program, workload and hardware resources change, different policies will be dynamically selected at runtime. Such an approach is known as a Mixture of Experts [11]. Critically, it does not try out different policies, varying the number of threads for a particular program at runtime, as this is too expensive.

The central issue is: how do we, at runtime, evaluate whether a particular policy is good? We cannot afford to try them all out and pick the best. Furthermore, once we have selected a policy and followed its decision, we still do not know how good it was, as the environment might have changed. There is no monitor we can look at to evaluate its performance. This is a key challenge to thread selection.

We overcome this by developing a novel approach that uses models that not only predict what the right number of threads should be for a program, they also predict what the environment will look like. Given this ability to determine whether...
a policy is accurate, we dynamically monitor the prediction accuracy of each model, selecting an expert whenever we think it is the most accurate for a particular environment.

We extensively evaluate our approach against existing policies in dynamic environments, outperform all existing approaches and surprisingly never slowdown the workload. We consider the case when the workloads themselves use affinity scheduling, are smart and adaptive. In each case our approach improves the performance of the target program and workloads.

2. Motivation

Realistic systems are highly dynamic with programs sharing the system resources. Figure 1 shows real workload behavior derived from a log over a period of 50 hours activity in a high performance computing system (2912 cores, 5824 H/W contexts, 24GB RAM).

The program performance using these techniques is seen in Figure 2. The analytic approach improves over the OpenMP default but is outperformed by either expert due to its delay in reacting to change. Having the ability to dynamically switch between experts significantly improves performance further still.

3. Mixture of Experts: Overview

Our approach is to use a number of different policies or experts to predict the best number of threads at a given instance. Each expert is trained offline and we dynamically select the best expert to use at runtime. To make things concrete, in our experiments throughout this paper, we consider the case where the training data is divided arbitrarily amongst 4 experts based on program scaling behavior and H/W configuration (explained in Section 4.1). We analyse this decision in Section 7.

This Mixture of Experts approach is a supervised learning technique for systems composed of many distinct models. An expert selector model decides which expert should be invoked for each dynamic case. In parallelism mapping, central to our formulation is the concept of reward i.e. determining how good a mapping is. Given a number of mapping policies, this approach learns online which expert is best suited to each dynamic decision.

3.1 Offline Expert

Each expert has two models associated with it: (a) thread predictor ‘w’ and (b) an environment predictor ‘m’. Assume ‘p’ represents any vector p. Let c_L denote the parallel loop code, e_t, the corresponding runtime environment, f_t = c_L | e_t, the features combining code and environment information at time stamp t.

Thread Prediction: We define a thread predictor ‘w’ that predicts the thread number which maximizes speedup given the input features vector.

\[ w(f_t) = n \]

This policy is learnt offline and applied dynamically at runtime. There is no re-learning of policy at runtime.

Environment Prediction: Environment predictor m is a predictive model which is trained to predict the future environment state given current features. At time-stamp t, given the current system state encoded as a feature vector f_t, this model returns the possible environment at time-stamp, \( e_{t+1} \).

\[ m(f_t) = \hat{e}_{t+1} \]

If this prediction is incorrect then the thread prediction will be incorrect. While it is hard to determine the accuracy of w, it is easy to judge the accuracy of m at the next time stamp. As m and w are built from the same training data, they are correlated. For this training set it is observed that if m is accurate, so is w. Environment prediction is the key to monitoring the accuracy of the experts.

3.2 Expert Selector

We assume we have a number of different predictors or experts, each of which has an associated predictor pair (m^k, w^k). The role of the mixture of experts model M is
to select the best expert ‘k’ which is predicted to give the best performance:

\[ M(f_k) = k |\text{argmin}_k \|e_k^f - e_i^f\| \]

in other words, select the expert that is most accurate in predicting the environment. This process is shown in Figure 3. The code and environment features, \( f \), are input to the online expert selector \( M \) which determines which expert to select based on the features. The thread prediction \( n \) of the selected expert is then output. The model \( M \) adapts over time based on the accuracy of each expert’s prediction of the future environment \( \hat{e} \).

4. Our Approach

Here we describe how each individual expert is built. This is followed by a description of how a model is learnt online to select the best expert based on the accuracy of each expert’s environment prediction.

4.1 Individual Experts

Each expert is an offline trained mapping policy where each policy contains two models as mentioned above \( w, m \). There are numerous ways of selecting the number and type of training data for each expert. For this paper, we used the following arbitrary approach to build 4 experts: We first separate the training programs into 2 sets: those that scale well and those that do not. We then built an expert for each set on 2 different platforms: a 12 core machine and a 32 core machine, giving 4 experts in all.

4.2 Predictive Modeling

Machine learning techniques using supervised learning are employed for training the experts [7, 14]. This uses the standard three stage process: (i) generate training data; (ii) train and build a model using cross-validation (iii) deploy this learnt model in an unseen setting.

4.2.1 Training Data Set for Experts:

Building an expert model requires training data. The training experiments consisted of one target and one workload from NAS suite where each program runs until the other finishes. These runs are repeated by varying the number of threads for both programs. No retraining is needed on other platforms of interest. It is to be noted that only NAS programs were used for training but programs from SpecOMP, Parsec are used for evaluation. We capture features \( f = [c, e] \) where \( c \) are static code features and \( e \) environment features and record the number of threads \( n \) that leads to best performance. The training is performed for each expert and incurs a one-off cost. Once the experts are built there is no further re-learning.

4.2.2 Features:

While predictive modeling is relatively automated, it critically relies on good feature selection. During the training phase 134 features, \( f \), were collected, comprising of many code (\( c \)) and environment (\( e \)) parameters available within our LLVM-based compiler and Linux. From these, 10 features were chosen that were found to be critical to the models based on the quality of information gain. These are listed in Table 1. At loop \( i \), the feature vector \( \hat{f}_i = (f_{i1}, \ldots, f_{i10}) \) is formed by these 10 features.

4.2.3 Linear Regression

We use a linear regression technique employing standard least squares to build two models that fit the training data. Other models could equally be used. We employ the standard leave-one-out cross validation methodology that ensures we keep training and test data separate, i.e. if we are trying to predict the number of threads for program \( bt \), we ensure that \( bt \) is not part of the training set. Learning a model for this data is simply finding the best linear fit to the data i.e. determining weights for each selected feature \( w_1 f_1 + \ldots + w_n f_n + \beta \). This results in simple 10-dimensional linear model \( n = \frac{w f}{\bar{w}} \) and \( \bar{e} = m f \) where the weights (coefficients)

<table>
<thead>
<tr>
<th>Features</th>
<th>( E_1 )</th>
<th>( E_2 )</th>
<th>( E_3 )</th>
<th>( E_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>type</td>
<td>( w )</td>
<td>( m )</td>
<td>( w )</td>
</tr>
<tr>
<td>( f^1 ) load/store count</td>
<td>compiler</td>
<td>1.05</td>
<td>-0.47</td>
<td>-0.84</td>
</tr>
<tr>
<td>( f^2 ) instructions</td>
<td></td>
<td>-1.52</td>
<td>0.35</td>
<td>1.12</td>
</tr>
<tr>
<td>( f^3 ) branches</td>
<td></td>
<td>0.87</td>
<td>1.15</td>
<td>0.84</td>
</tr>
<tr>
<td>( f^4 ) workload threads</td>
<td>Linux</td>
<td>-0.62</td>
<td>0.39</td>
<td>0.05</td>
</tr>
<tr>
<td>( f^5 ) processors</td>
<td></td>
<td>0.08</td>
<td>0.46</td>
<td>0.98</td>
</tr>
<tr>
<td>( f^6 ) run queue size (runq-sz)</td>
<td></td>
<td>0.003</td>
<td>0.29</td>
<td>0.02</td>
</tr>
<tr>
<td>( f^7 ) cpu load (ldavg-1)</td>
<td></td>
<td>0.002</td>
<td>0.17</td>
<td>0.03</td>
</tr>
<tr>
<td>( f^8 ) cpu load (ldavg-5)</td>
<td></td>
<td>-0.013</td>
<td>0.64</td>
<td>0.227</td>
</tr>
<tr>
<td>( f^9 ) cached memory</td>
<td></td>
<td>-0.07</td>
<td>0.01</td>
<td>0.002</td>
</tr>
<tr>
<td>( f^{10} ) pages free list rate</td>
<td></td>
<td>0.004</td>
<td>0.002</td>
<td>-0.08</td>
</tr>
</tbody>
</table>

\( \beta \) regression constant | | -1.21 | 0.25 | -6.8 | 0.28 | -3.03 | 0.33 | -2.5 | -0.05 |
and \(m\) are listed in Table 1. Each of the 4 experts has its own weights.

5. Experimental Setup

<table>
<thead>
<tr>
<th>Hardware</th>
<th>32-core Intel Xeon L7555 @1.87GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>64-bit openSUSE 12.3 version</td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc 4.6 -O3 optimization</td>
</tr>
</tbody>
</table>

This section describes the hardware platform and benchmarks used. It describes the adaptive techniques we compare against and outlines the dynamic environment used.

5.1 System

The experimental setup used as the evaluation platform is listed in Table 2. Target and workloads begin their execution at the same time and continue running till the other finishes. Each experiment was repeated 3 times and the mean value of program execution time reported.

5.2 Applications

We use a range of multi-threaded programs from various domains: all OpenMP-based C programs from NAS, SpecOMP and Parsec benchmark suites with respective largest input datasets.

5.3 Policies

We evaluated our approach against the following adaptive policies discussed in Section 9:

Default: OpenMP default policy [4] assigns a thread number equal to the current number of available processors.

Analytic: In [19] an analytical model determines the degree of parallelism at runtime based on observed speedups at fixed time-intervals and estimated using regression techniques

Offline: In [7] a machine learning heuristic predicts a thread number at runtime based on an offline-trained model.

Online: [16] is a robust adaptive scheme that employs hill-climbing technique to change the thread count at runtime based on execution time.

5.4 Experimental Scenarios

We evaluated our approach in a dynamic execution environment with varying co-executing workloads and the number of processors. The effect of other external system issues such as network contention are reflected in the set of runtime features used by our model.

Workloads: The external workload consists of multiple parallel programs selected from the above benchmarks. We vary the number of workload programs chosen from above programs classified as ‘small’ and ‘large’.

For each workload type, we consider different sets of programs as shown in Table 3. All results are averaged over these different benchmark sets. The same external workload is reproduced for all evaluated policies in all cases. This ensures a fair comparison across different mapping policies.

Hardware: To reflect any change in hardware, we vary the number of available processors during program execution. Changes in the number of processors can be due to several factors including hardware failures, assigning more/less cores for other high/low priority jobs, turning them off for saving power. We assume the hardware changes less frequently than workloads. The number of available processors is varied in two different frequencies: low and high where it is reduced or increased every 20 seconds and 10 seconds in low frequency and high frequency settings respectively.

6. Evaluation

In all cases, the baseline is OpenMP 3.0 default policy and the average values (hmean) are harmonic means to avoid outliers.

6.1 Dynamic Environment

Here we summarize the experimental results averaged across all programs under all evaluated scenarios.
6.2 Impact on Workloads

Any optimization scheme improving the target program performance should ideally exert minimal impact on the co-executing workloads. Figure 5(a) shows the impact of the evaluated schemes on the external workloads averaged across all experiment settings. All improve relative to the default on average, though online degrades the workload performance in certain cases. The offline and analytic models marginally improve over the online scheme. Our approach outperforms these techniques by improving workloads performance by 1.19x. This surprising result is primarily due to a reduction in system-wide contention benefiting target and workload.

6.3 Adaptive workloads

In this paper we have assumed that workloads vary in size and duration, but do not adapt their scheduling policy. Here we study the combined execution time when one program co-executes with another and both can adapt i.e. execute using different scheduling policies. Ideally an optimal combination of policies stabilizes the system, leading to faster execution of both programs. Figure 5(b) shows the measured speedups averaged across all program pairs. The baseline of 1.0 is the performance when each program employs the default policy. As observed from previous results, the mixtures approach is able to boost the targets’ performance over other evaluated schemes. What is interesting is what happens as the workload become smart and adapt using the same poli-
cies. If both programs use an online policy, they achieve only 1.08x improvement over the default. Using offline for both programs increases the performance to 1.27x, while analytic boosts this to 1.42x. The mixtures approach, however, if employed by both programs, delivers 1.81x speedup, a significant improvement over all of the other policies. Rather than fighting each other, employing a smart scheduling policy boosts each program’s performance. Employing the mixture of experts approach does this significantly.

6.4 Evaluation on Real-world Case-study
To demonstrate how our approach works in a real live system, we also ran a small-scale study with a real workload pattern in Figure 1. During the observed period, there was a hardware failure such that half of the processors were unavailable for 2 hours. This pattern was simulated on the platform from Section 5 where the number of workload threads was scaled down in proportion with the maximum number of processors. We ran all the benchmarks with this scenario and show the summarized speedup results in Figure 6(a). On average the speedups were online: 1.19x, offline 1.34x: analytic 1.43x and mixture 1.61x. Mixture of experts is clearly the superior policy, achieving improvement 1.32x, 1.21x, 1.15x over online, offline and analytic.

7. Analysis
7.1 Number of experts
One of the central claims of the mixture approach is that experts can be added over time, helping improve performance. In this analysis, we measured the target speedup with an increasing number of experts in the large workload, low frequency scenario. Figure 6(b) shows the average performance achieved across all benchmark programs in this scenario using a varying number of experts. Individually, each expert gives low performance. Experts 3 and 4 are most accurate here and give speedups of 1.22x and 1.27x. The mismatched experts 1 and 2 give performance of only 1.2x and 1.15x. However, adding experts steadily improves performance. This shows that the slight additional cost to determine the environment prediction accuracy is more than compensated by the performance gains. The mixture approach gives a 22% improvement over a best single-expert.

8. Conclusion
In this paper we presented a new technique based on a mixture of experts approach for efficient thread number selection. It determines at runtime, the best offline expert out of a collection of experts, as there is no one-size fits all universal best policy. It also provides a mechanism to gracefully add additional expertise knowledge. On evaluating with varying workloads and hardware resources, this approach improves over all existing approaches. Future work will explore the open problem of determining the ideal number of experts and the trade-off in number of experts vs. training data size. It will also investigate whether other modeling techniques such as SVMs trained on the same data or hand written analytic models can be selected by a mixtures approach. To ensure portability and robustness of our approach, we also plan to evaluate on alternative hardware platforms.

9. Related Work
Runtime adaptation techniques try to adjust the mapping based on the current execution scenarios. They typically lack “self-awareness” as they may involve considerable time-lag. Early feedback driven policy [20], assumes known programs and a static environment. ReSense [5] uses resource sensitivity to map co-located applications. A sensitivity score obtained by offline characterization per-program determines thread mappings at runtime. This approach assumes known programs and the scores do not reflect the sensitivity changes at runtime. We assume no such prior knowledge of programs. More sophisticated solutions proposed in [8–10] use control theory to adapt. However, the monitoring process is discrete (fixed time-steps) and slow while our approach adapts rapidly and continuously.

Techniques to improve program performance and utilization proposed in [21, 22] reduce resource contention caused by the programs. However they do not consider the dynamic nature of workloads. Petabricks [1] determines best runtime implementation of a program to adapt to a dynamic system. This approach requires different implementations offline, narrowing the scope of applicability. Few solutions aim to minimize resource utilization using work-stealing [2, 17] and other proposals target to avoid over-subscription in [3, 13]. These solutions require extensive offline profiling and are not effective on new unseen programs.

The paper closest to our work is [19] which builds and improves on [18]. It uses an analytic model to determine the degree of parallelism at runtime. The analytic policy relies on passive monitoring whereas we employ active monitoring continuously to measure the model quality. Other closely related work is [7]. Here a machine learning based technique is proposed that uses program and system parameters to adaptively map a program. However, it lacks a monitoring mechanism to detect the efficiency at runtime. [16] is an adaptive policy that uses hill-climbing technique to adjust the degree of parallelism using an orchestration mechanism proposed in [15]. Although this policy is robust to system changes, there is a delay to reach the best thread number and may stick in local optimum.

Mixture of Experts (ME) was first proposed by [11]. It used a set composed of several expert neural networks. Expectation-maximization algorithm is used to adjust parameters to each expert in [12]. Here the learning process is treated as a maximum likelihood problem. Boosting technique [6] describes a probabilistic model of improving learning by using a product of experts that greedily selects models incrementally.
References


