Celebrating Diversity: A Mixture of Experts Approach for Runtime Mapping in Dynamic Environments

Murali Krishna Emani
University of Edinburgh, UK
m.k.emani@sms.ed.ac.uk

Michael O'Boyle
University of Edinburgh, UK
mob@inf.ed.ac.uk

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. We then consider the case when workloads use affinity scheduling or are themselves adaptive and show that our approach, in all cases, outperforms existing schemes and surprisingly improves workload performance. On average, we improve 1.66x over OpenMP default, 1.34x over an online scheme, 1.25x over an offline policy and 1.2x over a state-of-art analytic model. Determining the right number and type of experts is an open problem and our initial analysis shows that adding more experts improves accuracy and performance.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Run-time environments; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel Programming

Keywords Parallelism Mapping, Dynamic Environment, Machine Learning, Mixture of Experts

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 [1] is shared, dynamic and unknown at compile time. Runtime systems such as [3, 6, 24, 28] 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. Schemes that try to combine offline models with runtime tuning [11, 14, 16, 20] can exploit prior knowledge of the program but are limited by the assumptions of the environment.

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 Mixture of Experts [17]. Critically, it does not try out different policies, varying the number of threads 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.

[1] We use *environment* to describe dynamic workloads/hardware resources

PLDI'15, June 13–17, 2015, Portland, OR, USA
© 2015 ACM. 978-1-4503-3468-6/15/06...$15.00
http://dx.doi.org/10.1145/2737924.2737999

A natural question to ask is: what is the right number of policies and how do we train them ? This is an open problem. In Section 3, we consider the simplest case of having just two experts while in Section 4 and the remaining part of this paper, we use four experts. In Section 8 we show that for the same amount of training data, a mixture of experts approach outperforms a single monolithic model and that increasing the number of experts improves performance.

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.

This paper makes the following contributions:

- First to employ mixture of experts for thread selection in dynamic environments.
- First to use an environment predictor as a proxy for selecting the best thread predictor.
- First to combine offline prior models and online learning.
- Shows that additional experts improves performance.

2. Related Work

Runtime adaptation techniques try to adjust the mapping based on the current execution scenarios. They typically lack "self-awareness" i.e. there is either no mechanism to detect the quality of the technique online or, if there is a provision, it involves considerable time-lag. Early feedback driven policy [30], assumes known programs and a static environment. ReSense [9] 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 and are able to accurately capture the contention arising due to co-execution.

More sophisticated solutions proposed in [12, 13, 15] use control theory to adapt. However, the monitoring process is discrete (fixed time-steps) and slow while our approach adapts rapidly and continuously. Optimal resource allocation using semi-bandit feedback policies in [19] learns the policy from the scratch at runtime while we exploit offline knowledge. It suffers from considerable delay in reaching the optimal allocation.

Techniques to improve program performance and utilization proposed in [31, 32] reduce resource contention caused by the programs. However they do not consider the dynamic nature of workloads. Petabricks [2, 3] determines best runtime implementation of a program to adapt to a dynamic system. This approach requires different implementations offline, narrowing the scope of applicability. Sambamba [29] adapts programs online but they are generic and slow to respond to changes in the environment. Speculatively executing multiple threads add significant overhead in the former. Few solutions aim to minimize resource utilization using work-stealing [5, 25] and other proposals target to avoid oversubscription in [7, 20]. These solutions require extensive offline profiling and are not effective on new unseen programs.

The paper closest to our work is [28] which builds and improves on [27]. It uses an analytic model to determine the degree of parallelism at runtime. Based on observed instantaneous performance, it executes for fixed time intervals with two randomly chosen thread numbers. The new thread number is then estimated using regression techniques. Such exploratory process incurs significant overhead. Our work avoids this overhead by making instantaneous decisions. The analytic policy relies on passive monitoring whereas we employ active monitoring continuously to measure the model quality.

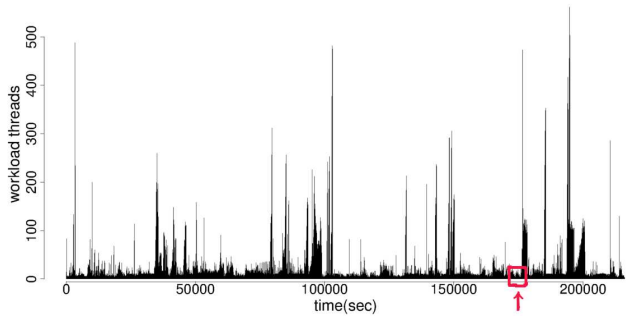


Figure 1. Highly dynamic system activity observed in a live system showing number of threads vs. time.

Other closely related work is [11]. 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. [24] is an adaptive policy that uses hill-climbing technique to adjust the degree of parallelism using an orchestration mechanism proposed in [23]. 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 [17]. It used a set composed of several expert neural networks. Expectation-maximization algorithm is used to adjust parameters to each expert in [18]. Here the learning process is treated as a maximum likelihood problem. Boosting technique [10] describes a probabilistic model of improving learning by using a product of experts that greedily selects models incrementally. Unlike standard ME, we do not select an expert and evaluate its performance online as there is no way to tell how well an expert performed. Instead, we are the first to use a proxy environment predictor as a measure of quality and then maximise likelihood. This prevents the need for the online trial of experts, reducing overhead and allowing rapid adaptation.

3. 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). Zooming in at point 175,000 (highlighted), we replicated this pattern in a scaled down experiment on a 12-core machine. As shown in the top graph of Figure 2, the number of workload threads and available processors varies over time. To aid comparison, we use a benchmark and workload used in [11] and [28]. We evaluate the latter, labelled *analytic*, which is the state-of-art mapping policy and the best alternative scheme. Here we wish to select the right number of threads for target `lu` co-executing with `mg`, both from NAS benchmark. The remaining graphs in Figure 2 shows the number of threads selected by four different policies over time, reacting to changes within the target program and the external environment.

The analytic policy first runs a parallel section with varying number of threads to measure speedup behavior before settling on a preferred number. This causes delay as can be seen at time step t_0. Just when the number of resources drops, this scheme increases the number of threads based on out-of-date data collected before t_0. It eventually settles down, but this delay has cost. The next two graphs show the behavior of 2 *experts*. Each expert (E^1, E^2) uses an offline trained model that predicts the best thread number. They differ in the state space that they are trained for, E^1 is more sensitive to changes in the number of processors than E^2, and consequently select different thread numbers.

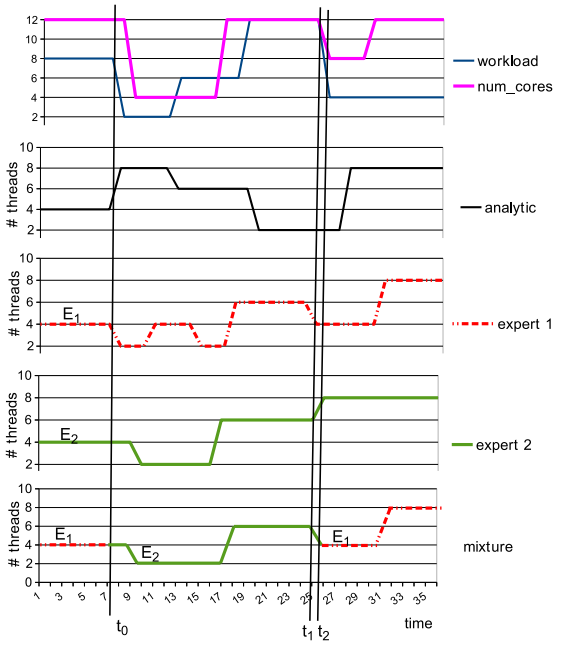


Figure 2. A snap shot of the dynamic system around $175,000^{th}$ second window. Top graph shows the number of workload threads and the number of cores available vs. time. Remaining graphs shows the number of threads selected by different policies over time. Change points at t_0, t_1, t_2 are highlighted. Analytic is delayed in reacting to change. The mixture approach selects expert 1 until t_0, expert 2 until t_2 and expert 1 thereafter.

The final graph shows the number of threads selected by our new approach *mixture*. It chooses the expert which is best-suited for that current execution environment. For example, consider the timeline at t_1, t_2. Initially, expert E^2 is the best model at t_1. When the execution environment changes at t_2, the selector switches to expert E^1 as it is more appropriate than E^2. As we show in Section 4.3, E^1 does in fact predict the environment more accurately at t_2 than E^2, so this was the right decision.

The program performance using these techniques is seen in Figure 3. 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.

4. Mixture of Experts

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 5.1). We analyse this decision in Section 8.

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.

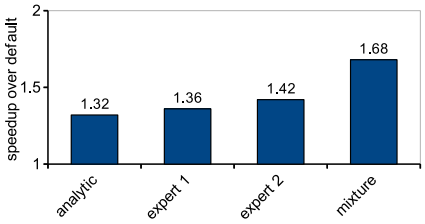


Figure 3. Selecting an optimal policy at runtime improves program performance

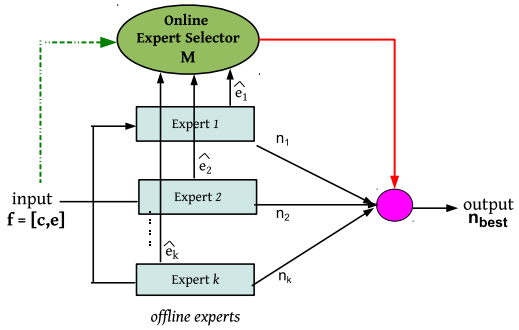


Figure 4. Mixture of experts: Depending on the input state f consisting of code c and environment e , the online model M chooses an expert most likely to select the best number of threads n. Based on the environment prediction accuracy of each expert, $\hat{e}$, it updates its choice over time

4.1 Offline Expert

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

Thread Prediction: A model x is learnt that given a thread number n returns an approximation of eventual speedup

$$x(n, \underline{f_t}) = \hat{V}(n)$$

where $\hat{}$ is used to denote an approximation and $\hat{V}(n)$ is the predicted program speedup with n threads. We then define a *thread predictor* 'w' that selects the thread number that is predicted to maximize speedup:

$$w(\underline{f_t}) = n|(argmax_n(x(n, \underline{f_t})))]$$

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 $\underline{f_t}$, this model returns the possible environment at time-stamp, $\hat{e_{t+1}}$.

$$m(\underline{f_t}) = \underline{\hat{e_{t+1}}}$$

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, as discussed in later sections.

Environment prediction is the key to monitoring the accuracy of the experts. Existing experts that are generated using machine learning can be retrofitted by retraining them, using the same original training data, to predict the environment as well. It is more challenging for hand-crafted or ad-hoc experts as a new environment predictor would need to be created. Alternatively, we could online, periodically select an expert (with no environment predictor) and see how it affects the environment and record the result, slowly building an environment predictor automatically over time.

4.2 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:

$$M(\underline{f_t}) = k|argmax_k(argmax_n(x^k(n, \underline{f_t})))]$$

in other words, select the expert that is expected to predict the number of threads that will lead to maximum speedup. This selection is to be performed *online*. Learning the best gating function M in this way is not feasible, however, as we cannot, at runtime, evaluate the performance of x^k and hence the thread predictor w^k. Instead, we use the environment predictor and instead formulate the prediction as:

$$M(\underline{f_t}) = k|argmin_k\|\hat{e_t^k} - e_t\|$$

in other words, select the expert that is most accurate in predicting the environment. As this can be evaluated at each time step, it can be used to build, online, the mixture of experts model M. This process is shown in Figure 4. 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 $\hat{\underline{e}}$. In the next section we describe how the experts are built and how M is learnt and deployed at runtime.

5. 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.

5.1 Individual Experts

Each expert is an offline trained mapping policy where each policy contains *two* models as mentioned above w, m. Any (potentially external) expert that determines these two parameters, via whatever means, can be included in the existing mixture.

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. We defined a program as being scalable if it achieves at least $P/4$ speedup where P is the number of processors. This allocation of training data to experts is shown in Figure 5.

5.2 Predictive Modeling

Machine learning techniques using supervised learning are employed for training the experts [11, 21]. 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.

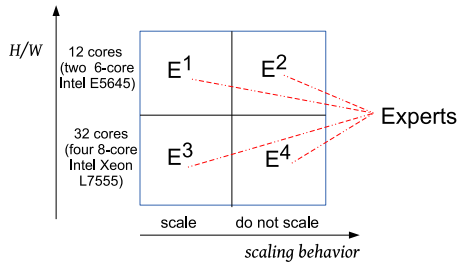


Figure 5. Diagram showing how four experts are selected.

5.2.1 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 $\underline{f} = [\underline{c}, \underline{e}]$ where $\underline{c}$ are static code features and $\underline{e}$ 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.

5.2.2 Features

While predictive modeling is relatively automated, it critically relies on good feature selection. During the training phase 134 features, $\underline{f}$, 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 $\underline{f}_i$=($f^1{}_i$, .., $f^{10}{}_i$) is formed by these 10 features. The code features at every loop were normalized to the total number of instructions in the program. Code features characterize a program's inherent compute, memory or I/O intensive properties. The runtime features characterize contention and load in the system. In this paper, the environment is formalized as the norm of the runtime features in this feature set (f^4 tof^{10}).

Although all experts uses the same features, they vary in importance across each expert. Figure 6 shows the importance of selected features across different experts. We define *feature impact* (π) as the drop in prediction accuracy of the model when this feature alone was removed from the feature-set. The resultant normalized values for all four experts form the pie-charts. Each slice of the pie-chart corresponds to how crucial is the feature for that expert. These features are ranked in relative importance to the experts. For example, run queue size is more critical to expert E^1 and less important to other experts. Certain features such as *#processors* are nearly the same for all experts.

5.2.3 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 ($w_1 f_1 + .. + w_n f_n + \beta$). This results in simple 10-dimensional linear model $n = \underline{w}\underline{f}$ and $\hat{e} = \underline{m}\underline{f}$ where the weights (coefficients)

Table 1. List of features, regression coefficients

Features			E_1		E_2		E_3		E_4	
	Description	type	w	m	w	m	w	m	w	m
f^1	load/store count	compiler	1.05	-0.47	-0.84	1.02	0.14	1.1	0.05	0.74
f^2	instructions	"	-1.52	0.35	1.12	-0.78	0.95	1.10	0.03	1.03
f^3	branches	"	0.87	1.15	0.84	0.05	-0.87	0.54	-0.57	1.12
f^4	workload threads	Linux	-0.62	0.39	0.05	0.44	-0.48	0.44	0.004	0.39
f^5	processors	"	0.98	0.46	0.98	0.002	0.99	0.142	0.92	0.74
f^6	run queue size (runq-sz)	"	0.003	0.29	0.02	0.23	-0.15	0.25	0.22	0.28
f^7	cpu load (ldavg-1)	"	0.002	0.17	0.03	0.09	0.473	0.07	0.01	0.09
f^8	cpu load (ldavg-5)	"	-0.013	0.64	0.227	0.6	-1.07	0.15	-0.62	0.59
f^9	cached memory	"	-0.07	0.01	0.002	0.05	0.007	0.06	0.03	0.12
f^{10}	pages free list rate	"	0.004	0.002	-0.08	-0.04	0.01	0.14	-0.14	0.003
β	regression constant		-1.21	0.25	-6.8	0.28	-3.03	0.33	-2.5	-0.05

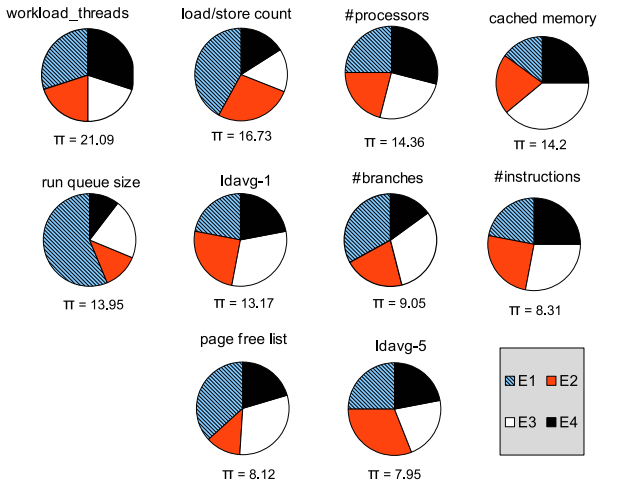


Figure 6. Impact of selected features on the experts. A slice in each pie-chart corresponds to how crucial is the feature for that expert. Feature impact (π) value below a pie-chart is averaged across all experts.

w and m are listed in Table 1. Each of the 4 experts has its own weights.

5.3 Expert Selector / Feature Space Partitioning

A mixture of experts model M consists of a series of hyperplanes $\underline{S}$ in the 10-dimensional feature space $\underline{f}$. These hyperplanes define the regions in the feature space where one expert is more accurate than the others. The environment prediction error, a for expert k is defined as

$$a^k = \|\underline{\hat{e^k}}_t\| - \|\underline{e}_t\|$$

The hyperplanes $\underline{S}$ are learnt online such that the error of a predictor k in this region is less than the average error of the other predictors. ('$<$' is based on Euclidean distance)

$$argmin_{\underline{S}^k} \|(a^k - \frac{\sum_{i \neq k} a^i}{K-1})\|, S^{k-1} < \underline{f} \leq S^k$$

We initially partition the space evenly and adjust S online based on the accuracy of each expert. To minimize runtime overhead, we only use data from the last timestep to update the model.

5.4 Example

To demonstrate how our approach works consider the workload timeline shown in Figure 2. At timestamp t_1, the feature-vector $\underline{f}_1$ is

$$\underline{f}_1 = [0.032, 0.026, 0.2, 4, 8, 16, 4.76, 2.17, 1.11, 1.65]$$

Expert E^1 predicts thread number n_1^1 and environment $\hat{e_1^1}$ by the product of weights w_1, m_1 from Table 1 with $\underline{f}_1$ as

$$n_1^1 = \underline{w^1} \cdot \underline{f}_1 = 4; \qquad \|\hat{e_1^1}\| = \underline{m^1} \cdot \underline{f}_1 = 12.56$$

Similarly expert E^2 predicts

$$n_1^2 = \underline{w^2} \cdot \underline{f}_1 = 6; \qquad \|\hat{e_1^2}\| = \underline{m^2} \cdot \underline{f}_1 = 7.2$$

The Mixture of Expert selection $\underline{S}^1$ hyperplane is = [0.04,0,02, 0.2, 6, 10, 14, 4.00, 2.00, 1,1.5] and as $\underline{f}_1 < \underline{S}^1$, it selects E^2 as its expert and chooses 6 threads. This, in fact, turns out to be the correct decision as the actual measured environment is $\|\underline{e}_1\| = 8.713$ which is closer to E^2's prediction of 7.2 rather than E^1's of 12.56. Later at timestamp t_2, the feature-vector $\underline{f}_2$ is

$$\underline{f}_2 = [0.045, 0.013, 0.1, 12, 12, 6, 2.73, 2.17, 0.01, 1.21]$$

Here, expert E^1 predicts n_2^1 and $\hat{e_2^1}$ as

$$n_2^1 = \underline{w^1} \cdot \underline{f}_2 = 4; \qquad \|\hat{e_2^1}\| = \underline{m^1} \cdot \underline{f}_2 = 13.94$$

Similarly expert E^2 predicts

$$n_2^2 = \underline{w^2} \cdot \underline{f}_2 = 8; \qquad \|\hat{e_2^2}\| = \underline{m^2} \cdot \underline{f}_2 = 8.504$$

Here, the mixture of experts selects expert E^1 as $\underline{S}^1 < \underline{f}_2$ and chooses 4 threads. This is the correct decision as the actual measured environment is $\|\underline{e}_2\| = 11.763$ which is closer to E^1's prediction of 13.94 rather than E^2's of 8.504. If there was a misprediction, the hyperplane S would be updated to reclassify this feature point.

6. Experimental Setup

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

6.1 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.

6.2 Applications

We use a range of multi-threaded programs from various domains: all OpenMP-based C programs from NAS [1], SpecOMP [26]

Table 2. H/W and S/W configurations for evaluation.

Hardware	32-core Intel Xeon L7555 @1.87GHz 4 one-socket nodes, 8 cores/socket 64GB RAM, 24MB shared LLC
OS	64-bit openSUSE 12.3 version 3.7.10 kernel
Compiler	*gcc* 4.6 -O3 optimization

and Parsec [4, 22] benchmark suites with respective largest input datasets. These representative parallel programs consisting of compute and memory-bound programs are diverse with emerging workloads from various domains. For example, `blackscholes` from Parsec is a financial application which computes options pricing using Black-scholes partial differential equations. SpecOMP programs target high performance computing (HPC) domain. NAS programs are derived from computational fluid dynamics applications. Such a selection of programs ensures that we evaluate our approach on a wide variety of programs.

Table 3. Workload configuration

Workload	Benchmarks
small	(i) `is,cg` (ii) `ammp,fft`
large	(i) `bt,sp,equake,is,cg,art` (ii) `bscholes,lu,bt,sp,fmine,art,mg`

6.3 Policies

We evaluated our approach against the following adaptive policies discussed in Section 2:

Default: OpenMP default policy [8] assigns a thread number equal to the current number of available processors.

Analytic: In [28] 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 [11] a machine learning heuristic predicts a thread number at runtime based on an offline-trained model.

Online: [24] is a robust adaptive scheme that employs hill-climbing technique to change the thread count at runtime based on execution time.

6.4 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.

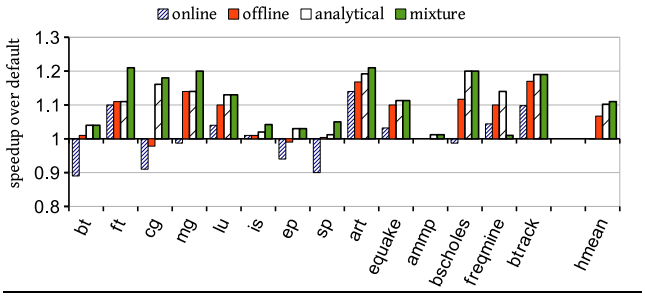


Figure 7. Evaluation of policies in an isolated static system. Mixtures approach adds no overhead.

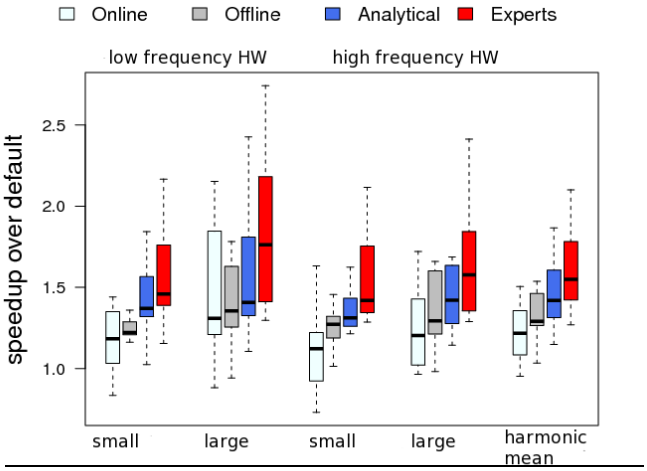


Figure 8. Speedup comparison of each scheme per workload and frequency of hardware change averaged across all benchmarks. Overall, on average, online, offline and analytic approaches improve performance by 1.23x, 1.33x and 1.39x respectively. Our approach outperforms these by achieving 1.66x mean (1.54x median) improvement.

7. Evaluation

First we show the results of each policy on an isolated static system. Next we consider dynamic systems with varying workloads. In all cases, the baseline is OpenMP 3.0 default policy and the average values (hmean) are harmonic means to avoid outliers.

7.1 Isolated and Static Environment

Result 1: *Mixture of Experts improves performance with no overhead in a static system under isolation.*

Figure 7 shows the results of the evaluated schemes in a system which is static (no changes in the environment) and isolated (no co-executing workloads). Online spends too much time trying different thread numbers slowing down a few programs. However, offline and analytic approaches adjust to find thread number leading to good speedup. The mixtures approach never slows down the target and improves `mg, cg, art`. These involve irregular memory accesses and barriers and spawning many threads slows down the program. Our approach analyzes this behaviour and determines the optimal thread number of the best expert which is most suited for a given program. On average, the mixtures approach improves 1.11x over the default and by 4% over the analytic scheme. This highlights that, although our approach is aimed at dynamic environment, it is reassuring that we incur no overhead in static systems.

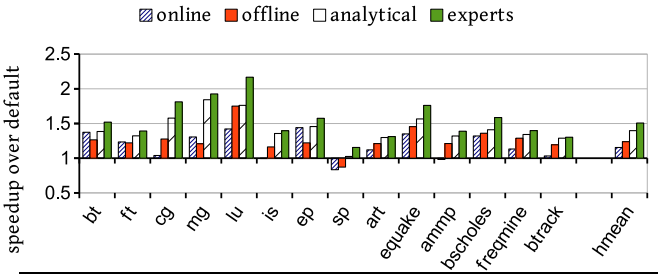


Figure 9. For targets executing with small workloads and low frequency hardware changes, our approach improves 1.5x over default, 1.3x over online, 1.22x over offline and 1.09x over analytic.

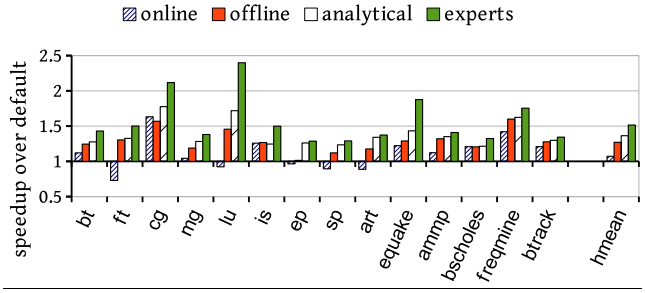


Figure 10. For targets executing with small workloads, high frequency hardware changes, our approach improves 1.51x over default, 1.41x over online, 1.19x over offline and 1.12x over analytic.

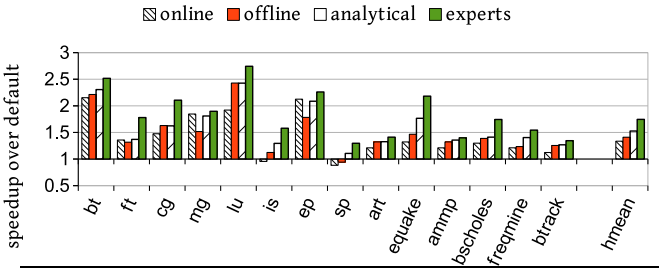


Figure 11. For targets executing with large workloads and low frequency hardware changes, our approach improves 1.74x over default, 1.31x over online, 1.23x over offline and 1.13x over analytic.

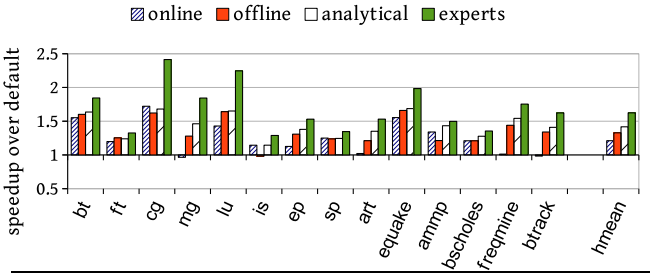


Figure 12. For targets executing with large workloads, high frequency hardware changes, this approach improves 1.62x over default, 1.34x over online, 1.22x over offline and 1.15x over analytic.

7.2 Dynamic Environment

Result 2: *Mixture of Experts significantly improves programs across diverse dynamic environments.*

Figure 8 summarises the results averaged across all benchmark programs for different workload and hardware settings. On average, online, offline and analytic approaches improve performance by 1.23x, 1.33x and 1.39x respectively. The mixture of experts approach outperforms all these by achieving 1.66x mean (1.54x median) speedup improvement.

The default policy performs poorly due to increased resource contention. The online technique adapts to the changes in the system by changing the thread number in response to the observed execution time. But this reacts slowly to the changes and hence achieves marginal improvement. The offline technique improves over the online scheme but it is limited by its workload training and cannot adapt to new environments. The analytic model performs well with workload change but is unable to adjust to the changing hardware resources. The mixture of experts approach immediately detects these changes and selects the best expert that is more specialized in the observed system state. It achieves significant improvement over these existing techniques.

Breakdown by Scenario: Here we examine in more detail the results on a per benchmark basis. We present results averaged over all the workloads on each workload and hardware setting.

7.2.1 Small Workload

Here the resource contention is minimal, however the changing number of processors limits the amount of computing resources.

Low frequency hardware change: Figure 9 shows the speedup for each policy averaged across all workload programs when the change in hardware is low. Here we improve performance 1.5x over OpenMP default and outperform all other schemes. Online improves over offline for `bt, ep` but it performs worse than the default for `sp`. Analytic approaches the performance of the mixture approach in some cases `mg, art, btrack` and `cg, equake` but in each case it is outperformed by the mixture of experts.

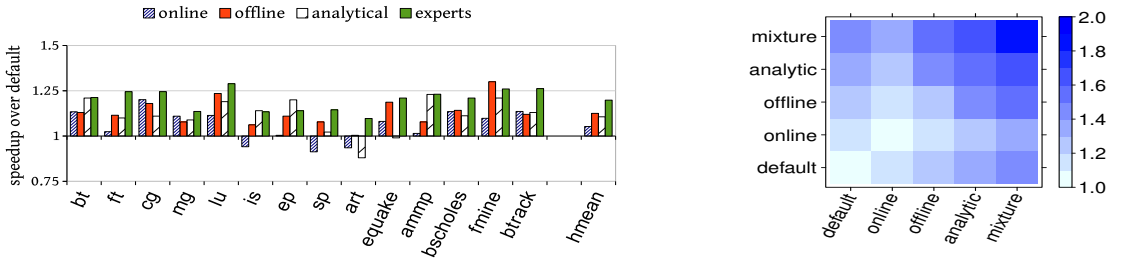
High frequency hardware change: With a high frequency hardware change, the amount of available resources changes at a faster rate. The results for this setting are shown in Figure 10. where we improve performance by 1.51x over the OpenMP default and outperforms the other techniques. The online technique slows down certain programs e.g., `ft, sp, art`. The offline approach never slows down any target program performing better than the online scheme. In all cases our approach achieves the best performance improvement.

7.2.2 Large Workload

With large workloads, the contention for system resource is greater. Variation in the available processors compounds this effect.

Low frequency hardware change: Figure 11 shows the results for this scenario. On average our approach achieves performance improvement of 1.74x over the default. Online improves `bt, ep` but slows down `is, ep`. Offline policy improves `bt, lu, cg, ep` but is ineffective for `is, sp, freqmine, btrack`. Analytic improves across all programs, but is outperformed by mixture of experts technique in all cases. `bt, lu, cg, equake` benefit a lot from our approach.

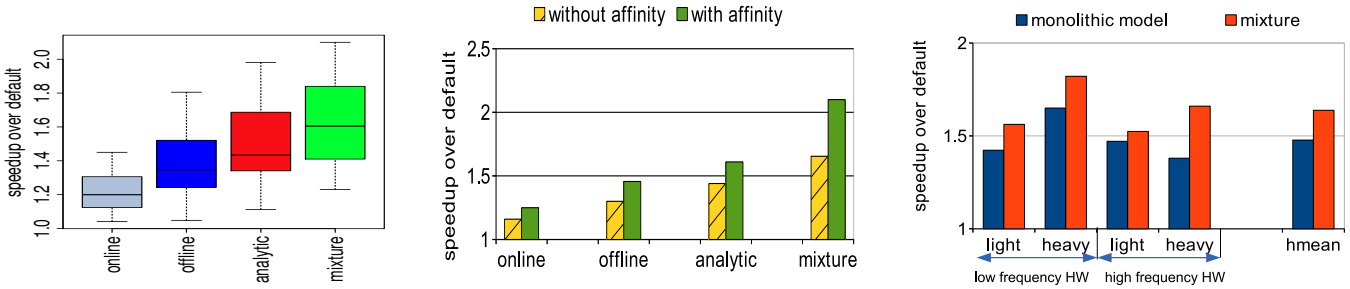
High frequency hardware change: Here we improve 1.62x over default, 1.34x over online, 1.22x over offline and 1.15x over analytic. Programs such as `cg, lu, equake, freqmine` benefit significantly from the mixture of experts. Offline and analytic improve over online across all programs except `sp`. Figure 12 shows the speedup results in detail.



(a) Impact on workloads

(b) Adaptive workloads

Figure 13. (a) Effect of policies on external workloads. The mixture approach never degrades workloads, improving workload by 1.19x on average. (b) Evaluation of pairs-of programs where both target and workload programs co-execute with smart policies. The mixture policy gives a combined improvement averaged across all pairs of 1.81x over the default.



(a) Evaluation on a real-world casestudy

(b) Thread affinity

(c) Monolithic model vs. mixture of experts

Figure 14. (a) In a live system, based on Figure 1, mixture improves by, on average, 1.32x, 1.21x, 1.19x over the online, offline and analytic models. (b) Impact of affinity scheduling on thread selection policies averaged over all benchmarks and workloads for small workloads scenario. Our approach gives a 2.1x average speedup and shows largest improvement of 28% using affinity scheduling. (c) Evaluation of monolithic model vs mixture of experts. The mixture approach improves 1.22x over the former.

7.3 Impact on Workloads

Result 3: *Mixture of Experts never slows down co-executing workloads.*

Any optimization scheme improving the target program performance should ideally exert minimal impact on the co-executing workloads. Figure 13(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 result is primarily due to a reduction in system-wide contention benefiting target and workload.

7.4 Adaptive Workloads

Result 4: *Mixture of Experts creates a win-win situation for target and 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 13(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 policies. 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.

7.5 Evaluation on a Real-world Case Study

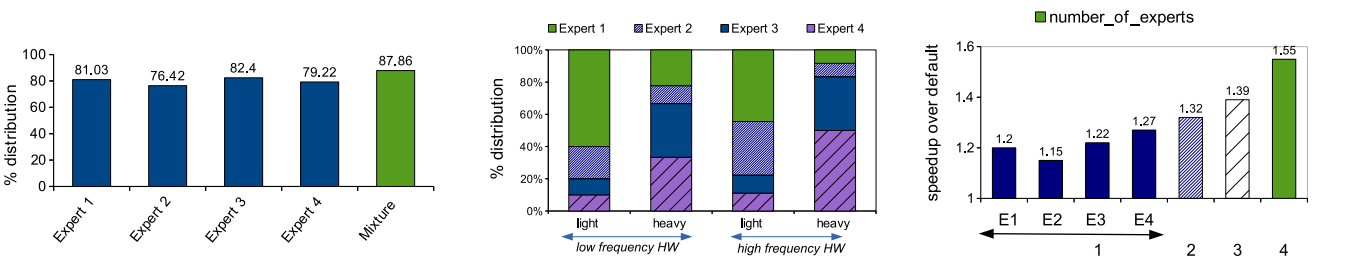
Result 5: *Mixture of Experts continuously adapts program parallelism to rapidly changing conditions.*

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 6 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 14(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, 1.19x over online, offline and analytic.

7.6 Thread Affinity

Result 6: *Mixture of Experts improves affinity scheduling.*

Associating threads to cores via affinity scheduling can improve performance as it may reduce memory traffic. Here we combine affinity scheduling with each of the thread selection policies. We ran all the benchmarks with multiple workloads in the small work-



(a) Efficiency of environment predictor (b) Distribution of experts (c) Effect of increasing number of experts

Figure 15. (a) Environment predictor accuracy of experts normalized to actual environment. (b) Distribution of the number of times an expert is chosen across each scenario. (c) Effect of increasing number of experts on program speedup averaged across all target programs. A mixture of 4 experts outperforms the best single expert by 1.22x.

load scenario described in Section 7.2.1, the scenario likely to benefit most from thread scheduling. The results averaged across the target benchmarks are shown in Figure 14(b). All schemes show improvement with affinity scheduling but our approach gives the largest improvement of 26%, giving an overall speedup of 2.1x.

7.7 Generic vs. Experts

Result 7: *Mixture of Experts yields better performance over one generic model composed of individual experts.*

Here we evaluate the performance of the mixture of experts policy comparing it against a single aggregate model with the same total training data. Figure 14(c) shows the speedup comparison using such a single model against the mixtures. The mixture of experts gives a 22% improvement over an aggregate model. This is due to the failure of the one size fits all approach of the aggregate model.

8. Analysis

We first analyse the accuracy of the experts and study how often they are selected. We also investigate how the number of experts impacts performance.

8.1 Environment Predictor Accuracy

The efficiency of the mixture of experts approach relies on the environment prediction capabilities of individual experts. Figure 15(a) shows how accurate these values are for each expert. Y-axis shows the normalized difference between observed and predicted environment averaged across all experiments. It can be seen that all experts accurately predict the future environment between 79% and 82% of the time. So individually there are highly accurate. When combined in a mixture model this accuracy increases to 87%.

8.2 Frequency of Expert Selection

Here we analyze how frequently the experts are selected by the mixture approach. If one expert were to dominate one or more scenarios, then having a mixture may be of little benefit. If, however, the frequency that an expert is selected is independent of scenario, then this undermines the need for online selection. Figure 15(b) shows the normalized frequency distribution of how many times each expert is selected in each of the four scenarios. As expected, one particular expert dominates each scenario: expert 1 is used 60% of the time for small workload, low frequency scenario, while expert 4 is preferred in the large workload, high frequency setting. Surprisingly, all experts are selected as the best at some point in each scenario. For instance, experts 1 and 2 are almost evenly chosen in the small workload, high frequency setting. This means that experts can be effectively used in scenarios that they have not been specifically trained for.

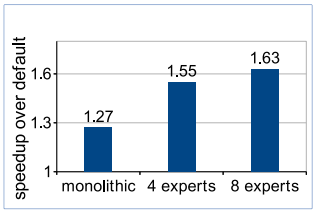


Figure 16. Increased granularity of number of experts

8.3 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 15(c) shows the average performance achieved across all benchmark programs in this scenario using a varying number of experts. Individually, each expert gives low performance. As expected, from Figure 15(b) 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.

8.4 Experts of Finer Granularity

Here we study how the number of experts impacts performance. We build 8 experts by further splitting the training programs based on scaling behavior and compare against the monolithic and 4 experts approach The results averaged across all programs for the scenario in Section 7.2.1, are presented in Figure 16. It can be observed that an increased number of experts further benefits the programs with 8 experts improving by 1.63x and 4 experts by 1.55x. This is probably due to the more specialized experts, capturing environment changes more precisely.

8.5 Distribution of Thread Numbers

Figure 17 shows the distribution of thread numbers predicted by individual experts and the mixture. It can be seen that the range of thread numbers vary with different expert across different evaluated scenarios. E^1 predicts larger thread numbers while E^4 predicts smaller numbers due to their training environments. These are consistent across the evaluated scenarios where as E^2 and E^3 are sensitive and vary in their predictions depending on the scenarios. The mixture *M* attempts to pick the best expert in all cases.

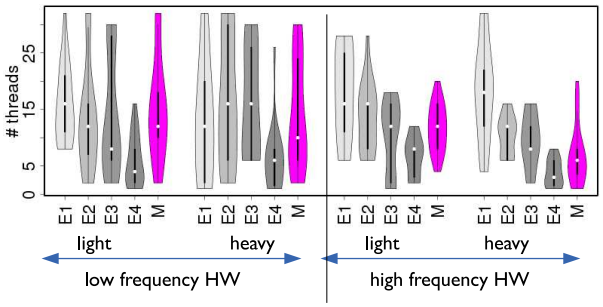


Figure 17. Thread number distribution

9. 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.

References

[1] NAS 2.3. `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

[2] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. CGO '11.

[3] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. PLDI, 2009. doi: 10.1145/1542476.1542481.

[4] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: http://doi.acm.org/10.1145/209936.209958.

[6] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and piranha. *IEEE Computer*, 28(1):40–49, Jan 1995. doi: 10.1109/2.362631.

[7] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. ICS '06, pages 157–166, New York, NY, USA, 2006. ACM. ISBN 1-59593-282-8. doi: http://doi.acm.org/10.1145/1183401.1183426.

[8] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998. ISSN 1070-9924. doi: 10.1109/99.660313.

[9] T. Dey, W. Wang, J. W. Davidson, and M. L. Soffa. Resense: Mapping dynamic workloads of colocated multithreaded applications using resource sensitivity. *ACM Trans. Archit. Code Optim.*, 10(4):41:1–41:25, Dec. 2013. ISSN 1544-3566. doi: 10.1145/2555289.2555298.

[10] N. U. Edakunni, G. Brown, and T. Kovacs. Boosting as a product of experts. *CoRR*, abs/1202.3716, 2012.

[11] M. K. Emani, Z. Wang, and M. F. O'Boyle. Smart, adaptive mapping of parallelism in the presence of external workload. In *CGO*, pages 1–10. IEEE, 2013.

[12] H. Hoffmann. Coadapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *ECRTS*, pages 223–232, July 2014. doi: 10.1109/ECRTS.2014.32.

[13] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. Seec: A framework for self-aware computing. 2010. URL `http://hdl.handle.net/1721.1/59519`.

[14] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. ASPLOS XVI, pages 199–212, New York, NY, USA, 2011. ACM. doi: 10.1145/1950365.1950390.

[15] H. Hoffmann, M. Maggio, M. Santambrogio, A. Leva, and A. Agarwal. A generalized software framework for accurate and efficient management of performance goals. EMSOFT '13, 2013.

[16] S. Ioannidis and S. Dwarkadas. Compiler and Run-Time Support for Adaptive Load Balancing in Software Distributed Shared Memory Systems. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 107–122. Springer Berlin Heidelberg, 1998.

[17] R. A. Jacobs, M. I. Jordan, S. J. Nowlan, and G. E. Hinton. Adaptive mixtures of local experts. *Neural Comput.*, 3(1):79–87, Mar. 1991. doi: 10.1162/neco.1991.3.1.79.

[18] M. Jordan and R. A. Jacobs. Hierarchical mixtures of experts and the em algorithm. In *IJCNN*, volume 2, pages 1339–1344 vol.2, Oct 1993. doi: 10.1109/IJCNN.1993.716791.

[19] T. Lattimore, K. Crammer, and C. Szepesvári. Optimal Resource Allocation with Semi-Bandit Feedback. *CoRR*, abs/1406.3840, 2014. URL `http://arxiv.org/abs/1406.3840`.

[20] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. ISCA, 2010. doi: 10.1145/1815961.1815996.

[21] S. Long and M. O'Boyle. Adaptive java optimisation using instance-based learning. ICS '04, 2004. doi: 10.1145/1006209.1006243. URL `http://doi.acm.org/10.1145/1006209.1006243`.

[22] Parsec. Parsec 2.1. `http://parsec.cs.princeton.edu/`.

[23] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: The degree of parallelism executive. PLDI '11, New York, NY, USA, 2011. ACM. doi: 10.1145/1993498.1993502.

[24] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: A System for Flexible Parallel Execution. PLDI '12, pages 133–144, 2012. doi: 10.1145/2254064.2254082.

[25] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007. ISBN 9780596514808.

[26] SpecOMP. SPECOMP 3.0. `http://www.spec.org/omp/`.

[27] S. Sridharan, G. Gupta, and G. S. Sohi. Holistic Run-time Parallelism Management for Time and Energy Efficiency. ICS '13, pages 337–348, 2013. doi: 10.1145/2464996.2465016.

[28] S. Sridharan, G. Gupta, and G. S. Sohi. Adaptive, Efficient, Parallel Execution of Parallel Programs. PLDI '14, pages 169–180, 2014. doi: 10.1145/2594291.2594292.

[29] K. Streit, C. Hammacher, A. Zeller, and S. Hack. Sambamba: a runtime system for online adaptive parallelization. CC'12, Berlin, Heidelberg, 2012. Springer-Verlag. doi: 10.1007/978-3-642-28652-0_13.

[30] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multithreaded workloads on CMPs. ASPLOS XIII, New York, NY, USA, 2008. ACM. doi: http://doi.acm.org/10.1145/1346281.1346317.

[31] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *ISCA*, pages 283–294. IEEE, 2011.

[32] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736036.