

# A Workload-Aware Mapping Approach For Data-Parallel Programs

Dominik Grewe  
dominik.grewe@ed.ac.uk

Zheng Wang  
jason.wangz@ed.ac.uk

Michael F.P. O’Boyle<sup>\*</sup>  
mob@inf.ed.ac.uk

Institute for Computing Systems Architecture  
School of Informatics  
University of Edinburgh  
Scotland, United Kingdom

## ABSTRACT

Much compiler-orientated work in the area of mapping parallel programs to parallel architectures has ignored the issue of external workload. Given that the majority of platforms will not be dedicated to just one task at a time, the impact of other jobs needs to be addressed. As mapping is highly dependent on the underlying machine, a technique that is easily portable across platforms is also desirable.

In this paper we develop an approach for predicting the optimal number of threads for a given data-parallel application in the presence of external workload. We achieve 93.7% of the maximum speedup available which gives an average speedup of 1.66 on 4 cores, a factor 1.24 times better than the OpenMP compiler’s default policy. We also develop an alternative cooperative model that minimizes the impact on external workload while still giving an improved average speedup. Finally, we evaluate our approach on a separate 8-core machine giving an average 1.33 times speedup over the default policy showing the portability of our approach.

## Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Compilers, Optimization*; D.1.3 [Programming Techniques]: Concurrent Programming—*Parallel Programming*

## General Terms

Experimentation, Languages, Performance

## Keywords

Compiler optimization, performance modeling, adaptive optimization, parallel programming, machine learning, predictive modeling, artificial neural networks

---

<sup>\*</sup>Member of HiPEAC

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HiPEAC 2011 Heraklion, Crete, Greece

Copyright 2011 ACM 978-1-4503-0241-8/11/01 ...\$10.00.

## 1. INTRODUCTION

Multi-core platforms are now common-place. While they provide the potential for energy-efficient scalable performance, effectively programming them remains a significant challenge, because the efficient mapping of such parallelism to the underlying hardware is critical for performance. Platforms are likely to change many times in an application’s life-cycle, so automatic methods that can systematically remap an application to new hardware are highly attractive.

Given the challenges faced in programming multi-cores, the majority of compiler research has made the necessary simplifying assumption that there is only one application to be executed on the target platform. While this may be true in certain, dedicated supercomputing environments, it is unlikely that an application can always guarantee splendid isolation on all multi-core platforms. As it is the optimizing compiler’s role to maximize the performance of any application, the potential impact of external multi-core workload on performance cannot be ignored. Depending on the situation we may wish to maximize performance without regard to external tasks or optimize program performance more cooperatively. It should work well when there are no competing tasks and adapt its mapping policy to varying degrees of external workload. Traditionally, the compiler assumed that such issues were handled by a run-time policy maker such as the operating system. However, the OS has little knowledge about the programs’ internal structure and is primarily concerned with fairness or throughput.

Mapping tasks to parallel machines is an extensively studied subject. There is a significant body of work that uses analytic approaches to statically map applications to classes of architectures [23, 24]. While accurate such approaches make simplifying assumptions about program/hardware interaction and are not portable across platforms. Others have looked at learning approaches to enable portability [33, 18, 9]. While portable, such approaches fail to consider external competition for resources. Dynamic scheduling approaches, in contrast do implicitly consider external workload, however they are reactive rather than predictive and can incur considerable run-time overhead [6, 8, 26, 32, 12]. All these approaches are concerned with how to best allocate global resources to meet competing requirements. Instead we are concerned with how to optimize and map our application given potentially restricted resources.

This paper develops an automatic portable compiler-based technique to determine the mapping of a parallel program

to a new architecture in the presence of external workload. The compiler automatically generates a lightweight run-time predictor that is integrated into the program to choose the best number of threads based on a description of the program and the current workload. Rather than developing a hand-crafted technique which requires updating whenever the system is upgraded, we develop an automatic technique based on predictive modeling with neural networks [5]. We build two separate models: one that maximizes application performance regardless of external workload and a more cooperative model that optimizes performance with minimal disruption to external tasks. On a four-core Intel Xeon platform we achieve 93.7% of the maximum performance achievable for OpenMP programs giving more stable performance and an average 1.24 times speedup over the compiler’s default run-time policy. We further evaluate our approach on a separate 8-core Intel Xeon machine and show an average 1.33 times speedup over the default, illustrating the portability of our approach.

The contributions of this paper are as follows:

- We develop a machine-learning based model that accurately predicts the right number of threads for an application with external workload.
- We show that such an approach works well across applications and workloads and outperforms existing approaches.
- We show that our approach is portable and can achieve significant performance improvement and reduce the impact on other applications.

The rest of this paper is structured as follows. In the next section a motivating example is given which is followed by a description of the setup of experiments in section 3. Section 4 explains the structure of the prediction approach presented in this paper. This is followed by a description of how the approach is evaluated and in section 6 the results of different mapping strategies are shown. Finally, a review of related work and concluding remarks are presented.

## 2. MOTIVATION

This section provides a motivating example showing that the best number of threads for a single parallel application varies dramatically depending on the application and external workload. A static policy that picks the same number of threads regardless of workload will give poor performance.

To illustrate this consider the graph in figure 1a. Figure 1 shows the speedup (y-axis) of the `fft` program as a function of its number of threads (x-axis) without any external workloads (solid, black line) as well as for varying external workloads on a 4-core machine. The workload consists of 0 - 4 other programs selected from the benchmark suites described in the next section.<sup>1</sup>

In the case of 1 workload program, the execution time of `fft` with just 1 thread allocated to it, is unsurprisingly equivalent to that of the sequential program. Executing `fft`

<sup>1</sup>For the purpose of illustration we only show single threaded external workloads here. In other words each of the 1 to 4 workloads consists of a single thread. In our evaluation we consider a much greater range of external workloads (see section 3).

with 2 threads reduces execution time with the best performance found with 3 threads. The single workload thread uses one core while `fft` uses the remaining 3. This leads to a speedup of 2.3 over the sequential run-time. The OpenMP compiler’s default policy selects 4 threads for `fft` leading to a slowdown of 1.6 over sequential execution. Consider now the case when we run the same `fft` program with 2 external workload programs as shown by the line labeled “2 workload programs”. Here the best performance is achieved using two threads, which leads to a speedup of 1.7 over sequential execution. Any further number of threads leads to a slowdown. The default OpenMP policy leads to a slowdown of 2.3. If the workload consists of more than two programs, it is best to run with just one thread.

From this example it may be tempting to conclude that creating  $\max(P-t, 1)$  threads, where  $P$  is the number of processors and  $t$  is the number of workload-threads, is the optimal solution when optimizing an application, i.e. 4 threads in total is best for a 4-core machine. Although such a simple adaptive technique works for this program and workload type, it does not hold in other cases. Consider the graph in figure 1b which shows the same experimental setup as figure 1a but this time we are evaluating the performance of `lmsfir` with external workload. For `lmsfir` the best performance regardless of workload is achieved by large numbers of threads. The shortest run-times are observed when seven or eight threads are used. `lmsfir` benefits from multithreading where the latency tolerance advantages outweigh cache pollution. These are just two examples of a variety of possible behaviors exhibited by parallel programs in the presence of workload. Furthermore, when considering multithreaded workload programs, as we do in this paper, more complex behavior can be observed.

The results shown here indicate that it may be worth parallelizing if there is other workload running on the machine. However, determining the best number of threads is non-obvious and depends on the program and workload characteristics as well as the underlying hardware and operating system. We therefore want a system that finds the best parallelism mapping in any workload situation. As a compiler is likely to have access to detailed information about its target application but only have lightweight information on workload at run-time, we develop a technique that fits these constraints.

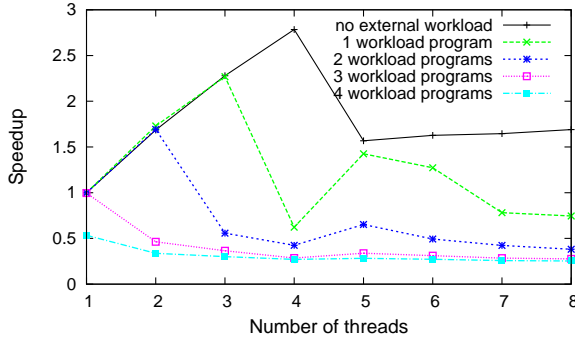
As the optimal mapping varies across programs and will vary across different platforms we wish to develop a technique that can *automatically adapt* to any hardware and run-time system. We therefore develop a predictive modeling based approach which learns the best mapping automatically without any hard-wired model. Although the examples in this section ignore the impact on external workload, this may be a critical consideration in certain settings. We therefore develop an alternative model that minimizes external workload impact.

## 3. EXPERIMENTAL SETUP

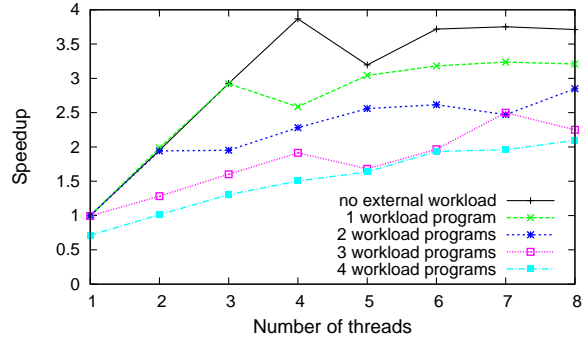
This section briefly describes the experimental setup, platform and benchmarks used throughout this paper.

### *Platform.*

Experiments were carried out on 4-core machines comprising of two dual-core Intel Xeon 5160 processors running at 3GHz and with a total of 8GB of main memory. The op-



(a) Performance of `fft` with and without workload.



(b) Performance of `lms` with and without workload.

Figure 1: The speedup of the `fft` and `lmsfir` benchmarks on a 4-core machine versus the number of threads in the presence of 0 to 4 external workload programs. On average the best number of threads achieves a speedup of 3.27 and 1.30, respectively, over the default policy, i.e. using 4 threads.

erating system used was 64-bit Scientific Linux 4 with kernel 2.6.9-78. Programs were parallelized with OpenMP [10] and compiled using the Intel ICC compiler version 10.1 with compiler flags `-xT -axT -ipo -openmp`. To show portability, an 8-core machine based on two quad-core Intel Xeon X5450 processors with a total of 16GB of main memory was also used.

### Benchmark Applications.

We evaluated our approach using all of the programs from the NAS parallel benchmark (NPB) and UTDSP benchmark suites [17] and all C benchmarks from the SPEC OMP benchmark suite [3] with the train data sets. This gives a total of 23 benchmarks. UTDSP consists of programs that perform standard Digital Signal Processing (DSP) functions. The NPB benchmark suite focuses on kernel and application programs that perform tasks typically found in the area of computational science. The SPEC OMP benchmarks are also primarily numerical benchmarks.

### Collecting Benchmark Execution Times in the Presence of Workload.

All benchmarks were executed with 1 to 8 threads on all workloads and the total run-time (wall clock time) was measured in order to collect the data needed for training and evaluation. This is the wall clock time of just the application under consideration. When considering the impact on workload, the elapsed wall clock time for all workload programs to finish is recorded. Each experiment was performed 20 times with the average time recorded. In this paper there was minimal variation across runs (less than 1%).

### Workloads.

The workloads were generated by randomly selecting sets of programs from the benchmark suites described above. For each hardware platform, this selection was only done once and used to evaluate all techniques discussed in this paper, ensuring a fair comparison. We evaluated different combinations of programs and thread sizes to ensure a variety of workloads with different characteristics in terms of resource usage and scalability. In order to make the experiments fair, the maximum number of workload threads was restricted to

the maximum number of cores on each platform: 4 and 8 respectively.<sup>2</sup> In cases of more than 1-thread workloads, all combinations of programs and thread numbers were considered. For instance with a 4-thread workload, we generated workloads consisting of: 4 programs, each with one thread; 3 programs, one of which has 2 threads the others having 1 thread; all the way to one workload program with 4 threads. For each application, we evaluated 54 distinct workloads. In each experiment, the workload programs were started simultaneously. After a short warm-up period the target program was executed.

## 4. PREDICTION APPROACH

The optimal number of threads for a program given an external workload is highly dependent on the underlying architecture and run-time system. Any small change in either of them, e.g. the number of cores or the scheduling algorithm, can affect the behavior of a parallel program and thus its optimal configuration in the presence of workload. So rather than hand-craft a heuristic, we use supervised learning [5] to develop an entirely automatic technique that is easily adaptable to any system without prior knowledge of the system’s characteristics.

In our approach we build a predictor for choosing the number of threads to use for a parallel program based on the characteristics of the program and the current workload situation. Formally, we are looking for a function  $f(\vec{p}, \vec{w}) \rightarrow \hat{t}$ , where  $\vec{p}$  is a vector of program features,  $\vec{w}$  is a vector of workload features and  $\hat{t}$  is the predicted optimal number of threads. The optimal number of threads depends on what our objective is. In this paper we consider two objectives (i) maximize application performance regardless of workload (ii) maximize application performance without adversely affecting workload. This is described in detail in section 4.2.

The structure of our approach is as follows. At first, training data is generated by running several programs with different workloads to determine the best thread configuration for that program/workload. These configurations with the

<sup>2</sup>With larger numbers of workload threads, the OpenMP compiler’s default policy rapidly degrades. We therefore limit their number to prevent artificially boosting our performance result.

corresponding program and workload features are used to train the model. All this is performed off-line “at the factory” and only needs to be repeated when the architecture or run-time system changes. A call to the model is inserted at the beginning of each program and is used to predict the optimal configuration given the program and the current workload situation.

A crucial part of building any prediction model is choosing the right features. They have to provide enough information about the program and workload to allow the model to make an accurate prediction. The next section describes the features used to create our predictor. Section 4.2 describes the algorithm that forms the basis for our predictor and the subsequent section explains how the model is trained and evaluated.

## 4.1 Feature Selection

### 4.1.1 Program Features

A compiler will have more detailed knowledge of its target application program than of the dynamic external workload. The speedups of an application in an *unloaded* environment provide a good insight into its behavior in a loaded one. Consider the graphs in figure 1 which show the speedup of the `fft` and `lmsfir` benchmark with and without external workload. The two programs exhibit a noticeably different scaling behavior which is reflected by the behavior in the presence of workload. One obvious difference is the behavior on an “over-saturated” machine, i.e. when more threads are spawned than there are cores. For some programs, performance degrades dramatically (see figure 1a), whereas other programs cope well with this situation (figure 1b). We implement multiple schemes that trade off feature collection cost and accuracy of the program characterization. In our cheapest scheme we only use two runs on a single-core processor with one and two threads, respectively. This is useful if no access to an unloaded multi-core machine is available. If, on the other hand, an unloaded machine is available at compile time, we collect up to six features requiring five runs on the machine, namely:  $s_5/s_4, s_6/s_5, s_6/s_4, s_8/s_4, s_4, s_8$  where  $s_i$  is the speedup with  $i$  threads over sequential execution on an unloaded machine.

### 4.1.2 Workload Features

While a compiler may have significant off-line knowledge of its application program it is unlikely to have significant information about the dynamic run-time workload. Workloads are therefore characterized by two extremely simple features: the total number of threads of all workload programs  $t$  and the number of programs the workload consists of,  $p$ . For example a workload consisting of a program having two threads together with a single-threaded program is characterized by the tuple  $(t, p) = (3, 2)$ . These features are easily accessible by querying the operating system and it is thus a realistic assumption to have this information available. In isolation, these two features do not provide enough information to make an accurate prediction as shown in section 2. However in combination with the scaling features of the application program the model is able to predict with high accuracy as shown in section 6.

## 4.2 Creating the Model

We use a Multi-Layer Perceptron (MLP), a form of arti-

ficial neural network, to model the problem because of its flexibility in modeling complex situations [33, 5]. In our case the model consists of three layers of nodes: an input layer, a hidden layer and a final output layer. These models have been shown to be able to approximate any function arbitrarily well [5].

The input to our model is a vector of features describing both the program and workload characteristics as described previously. Inside the model, the weighted inputs are passed through hyperbolic functions to account for non-linearity between in- and outputs. As output we get eight probabilities, each predicting the likelihood of a certain number of threads to be optimal. The model’s weights are determined by an optimization process which adjusts the model to the training data by minimizing the prediction error on these data. We used the scaled conjugate gradient method, because of its fast convergence rate compared to other optimization approaches [19]. It is important to note that only training data was used to create the models and no data from experiments involving the target program was used.

### 4.2.1 An Example Prediction

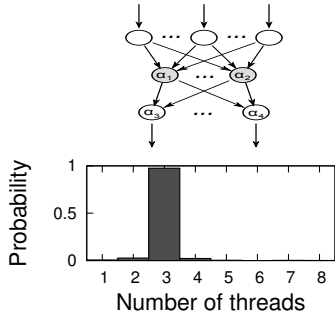
Figure 2 shows two example usages of our prediction model. In figure 2a the optimal number of threads for `fft` in an environment with one sequential workload program is predicted. The program and workload features are passed into the predictor and the output is a probability distribution over the number of threads. In this case, the predictor is almost 100% certain that using 3 threads is optimal. The speedups shown in figure 1a confirm that 3 threads is indeed the optimal choice for `fft` in this situation.

Figure 2b shows a prediction for the `lmsfir` benchmark also in an environment with one sequential workload program. In this case the output is not as unambiguous as before, with the predictor saying that both 7 and 8 threads have a high probability of being optimal. However, as can be seen in figure 1b, this is not a problem as both numbers provide near-optimal performance and picking one over the other doesn’t make a big difference.

### 4.2.2 Objective Function

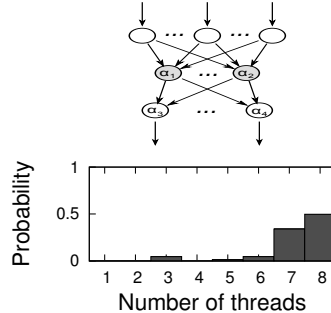
Predictive models are optimized towards a certain goal or *objective function*. This allows for creating models with different goals without having to change the underlying methodology. In this paper we created two models: The first predictor only cares about the *performance* of the program to be optimized, regardless of any slowdowns this may entail for the remaining workload. This is implemented by choosing the optimal number of threads simply as the thread number which provides the shortest execution time. In some situations, however, a more *cooperative* technique that also makes sure that the external workload’s performance is not dramatically impacted may be desirable. We thus also build a predictor that incorporates the impact on the workload into the objective function and is less radical with respect to the external workload. We therefore collected not only the run-time of the program to be optimized but also the workload program’s run-times. Now the optimal thread number was chosen to be the one that provides the fastest execution *while not slowing down the workload by more than 5%* with respect to sequential execution of the program to be optimized.

(0.56, 1.03, 0.59, 0.61, 2.78, 1.69, 4, 1)



(a) prediction for `fft` on one sequential workload program

(0.96, 1.16, 0.96, 0.96, 3.87, 3.71, 4, 1)



(b) prediction for `lmsfir` on one sequential workload program

Figure 2: Example predictions. Input features describing both the program and workload characteristics are passed into the neural network, which produces a probability distribution that predicts the optimal number of threads.

### 4.3 Training Cost

Like all predictive modeling approaches, the largest cost is associated with the collection of training data and the creation of the model. The collection of training data involves individually executing the benchmark programs in the presence of separate external workloads with different numbers of threads. In this paper 23 benchmarks and 58 sample workloads were used and we evaluated thread numbers between 1 and 8. Therefore, a total of  $23 \times 58 \times 8 = 10672$  experiments were required to collect the training data. However, this cost is paid just once at the factory, i.e. when the compiler is developed and does not add to compilation nor execution time. Clustering techniques can be used to significantly reduce training cost [30] and sampling can further reduce the cost on machines with large numbers of cores. However, this is beyond the scope of this paper.

#### 4.3.1 Prediction Overhead

At run-time, the neural network model is evaluated using the current workload situation and the program characteristics as inputs. This process is described in section 4.2 and mainly requires two matrix-vector multiplications with matrices of size  $8 \times h$  where  $h$  is at most 10, as well as  $h$  evaluations of the tanh function and 8 evaluations of the softmax function. These operations can be computed in a few micro-seconds and thus have essentially no impact on performance.

### 4.4 Deployment

Once the predictive model has been created it is used for choosing the optimal number of threads for a particular program in any workload situation. Because the workload features can only be obtained when the program is running, the predictor is part of the program. Once the program is executed, the workload information is collected and used as input to the model together with the previously gathered program features. The output of the model is the predicted optimal number of threads for this situation and determines how many threads are going to be created for this program. In our experiments the number of threads to use for a parallel region is dynamically set by calling a library function the cost of which is included in our experimental results.

## 5. METHODOLOGY

Section 3 described how the training data was collected and in section 4 the process of creating the model was shown. In this section we explain how the model is evaluated and introduce other mapping strategies to which it is compared in section 6.

The standard leave-one-out cross-validation methodology is used to evaluate the model. One program out of the  $N$  benchmarks is removed and the remaining  $N - 1$  programs are used to train the machine learning model as described in section 4.2. This model is then used to predict the optimal number of threads for the removed program in various workload situations. It is important to note that neither the left-out benchmark nor workload is used in the training phase. Hence, when a prediction is made, the test program has not influenced the prediction model and is *unseen*. We also evaluated changing the input data of the benchmark programs but didn't notice any difference in the program behavior.

We develop two models: *performance* is a model that attempts to maximize performance without regard to workload performance. Our second model *cooperative* attempts to maximize performance subject to not degrading workload performance by more than 5% with respect to a sequential execution of the program (see section 4.2.2 for more details).

There are two further mapping techniques which we compare our models against. The first one is ICC's OpenMP *default* scheme whose policy is always to use as many threads as there are cores.<sup>3</sup> We have also implemented a simple mapping technique, *simple adapt*, that adapts to the external workload: On a  $P$ -processor machine with  $t$  external workload-threads,  $\max(P - t, 1)$  threads are created. This method was motivated by the behavior of the `fft` benchmark in section 2. Unlike our model, *simple adapt* doesn't take program characteristics into account, but bases its prediction solely on workload information.

## 6. RESULTS AND EVALUATION

This section evaluates our predictive modeling approach when applied to benchmarks in the presence of workload.

<sup>3</sup>Note that this default behaviour is not part of the OpenMP specification, but used in many implementations such as the ICC and GCC OpenMP implementations.

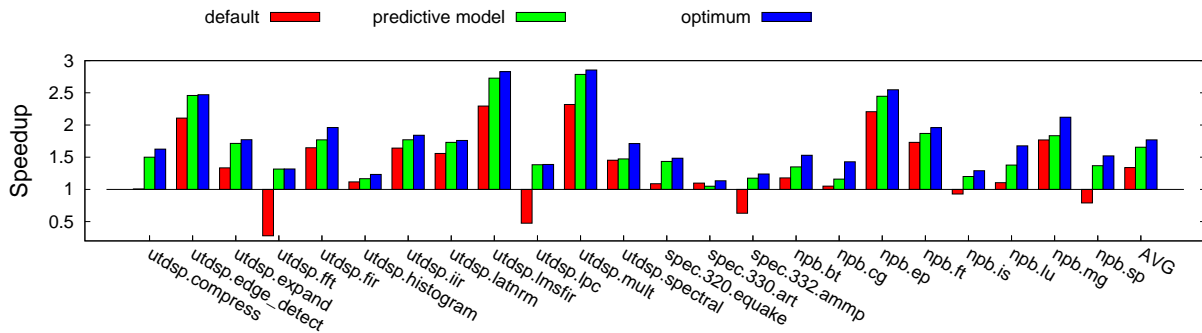


Figure 3: Speedup of mapping policies over sequential run-time across benchmarks averaged across external workloads. Our predictive model achieves an average speedup of 1.66 compared to 1.34 of the compiler’s OpenMP default policy. It outperforms the default policy for all but one benchmark.

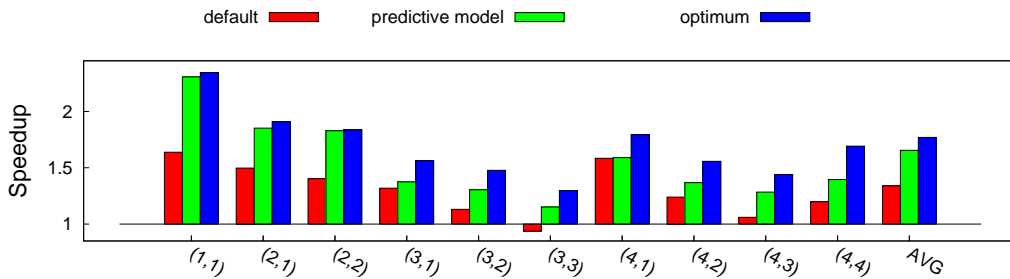


Figure 4: Speedup over sequential run-time across workloads averaged across all programs. A tuple  $(t, p)$  describes a workload consisting of a total of  $t$  threads made up from  $p$  programs ( $t \geq p$ ). Our predictor always outperforms the compiler’s OpenMP default policy.

We first evaluate the speedups of our performance-oriented model (*performance*) against the OpenMP default policy (*default*). Speedup results are first given on a per-program basis, then on a per-workload basis. In each case we show the performance of optimal thread selection as an upper-bound. We then compare the performance of our model to *simple adapt* described above in section 5. This is followed by an evaluation of the impact on the workload of different mapping strategies. Here we also consider the more *cooperative* prediction model. These predictors use the full set of features as described in section 4.1.1, but we also evaluate prediction models which use less costly features in this section. Finally, we evaluate the portability of our approach on a different platform with a larger number of cores.

## 6.1 Performance Evaluation

### 6.1.1 Evaluation by Program

Figure 3 shows the speedup over sequential execution for our (*performance*) predictive model as well as the performance of the OpenMP default policy. The results are presented on a per benchmark basis and averaged across all external workloads. In all but one case our predictive approach gives higher speedup results than the OpenMP *default* policy. While for some benchmarks, e.g. `spectral`, the performance improvement of our approach over the default is only marginal, the improvement can be as high as 4.7 times for benchmarks such as `fft`. On average, we achieve a speedup of 1.66 over the sequential run-time, which corresponds to a factor 1.24 times improvement over the OpenMP default

policy. Using the default policy in the presence of workload leads to slowdowns of as much as 3.5 for five of the 23 programs. In contrast, our technique improves performance for all benchmarks by up to a factor of 2.79.

Although our approach performs well against the default scheme, there may still be significant room for improvement. In order to provide an upper-bound on the performance achievable, we evaluated all possible thread numbers up to 8 for each program and workload and found the best speedup. Of course such an upper-bound would be impractical to find in practise but is a useful comparison for our experiments. Comparing our approach to that of the optimum, we achieve on average 92.3% of its performance (1.63 vs 1.77). This shows that not only do we significantly outperform OpenMP, we are not far from the best possible thread performance.

### 6.1.2 Evaluation by Workload

Although we perform well on a per-program basis, it may be the case that our scheme fails for certain types of workloads which may be significant in practice. So, rather than focusing our attention on program performance, in this section we evaluate the performance of our approach on a per-workload basis. Figure 4 shows the speedup over sequential execution of our approach and the default OpenMP approach on a workload by workload basis. The workloads are characterized by a tuple  $(t, p)$  where  $t$  denotes the total number of threads and  $p$  the number of programs in the workload. To aid side-by-side comparison we also show the performance of the optimal thread selection as an upper bound.

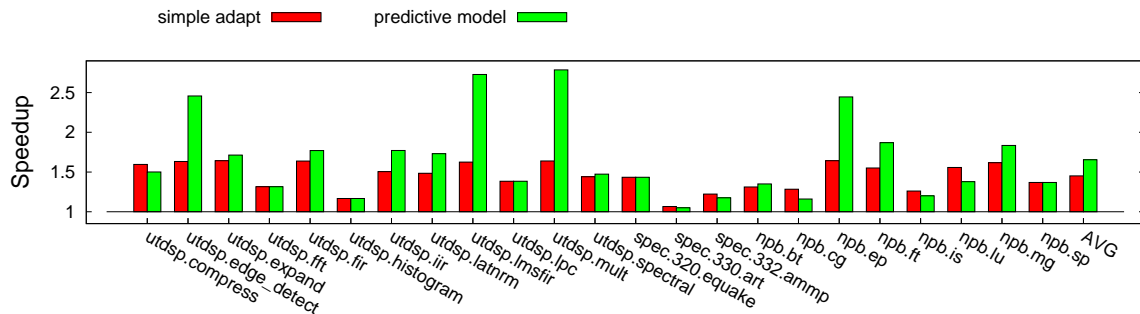


Figure 5: Speedup over sequential run-time of the predictive model compared to *simple adapt*. Although *simple adapt* approaches and sometimes even slightly improves upon our predictor for some benchmarks, it misses out on significant potential speedups for other applications.

Across the workloads our approach outperforms the default. Especially for small workloads with up to two threads, our model achieves significantly better results and performance is improved for 94% of all workload/benchmark pairs. This is significant as small occasional external workloads is a likely scenario when trying to optimize an application. For larger workloads consisting of a single program, i.e. (3, 1) and (4, 1), the default policy approaches the performance of our model. For the remaining workloads, however, the predictive model achieves better results in the majority of situations. If we compare our approach to the optimum, the greatest room for improvement is for (4, 4) workloads. Larger amounts of training data for this workload configuration will improve the accuracy of our technique.

### 6.1.3 Comparison to Simple Adapt

Previous sections have focused on comparing our predictive model against the static OpenMP default policy. Here we briefly consider a simple adaptive approach introduced in section 5, namely *simple adapt*. Figure 5 shows the speedups averaged across workloads for each benchmark for both mapping techniques. Although *simple adapt* approaches and sometimes even slightly improves upon our predictor for some benchmarks it misses out on significant potential speedups for other applications. *Simple adapt* achieves good performance for those benchmarks that improve performance with a reduced number of threads, e.g. `fft` or `compress`. However, for benchmarks whose optimal performance is achieved with a higher number of threads, such as `lmsfir` or `mult`, it fails to achieve good performance. Hence, an approach solely based on workload characteristics cannot achieve good speedups in all situations. On average, *simple adapt* achieves a speedup of 1.45 compared to 1.66 of the predictive modeling technique.

## 6.2 Impact on Workload Performance & the Cooperative Predictor

Our goal is to optimize the performance of a *single* program in the presence of external workload rather than optimizing *overall throughput*. However, there may be scenarios when the external workload should not be drastically affected. We therefore investigate how workload is affected in our approach compared to the OpenMP default strategy. A more cooperative model that takes workload into account (see section 4.2) is also evaluated.

To measure the impact on the workload, we computed the workload performance when the target program is executed with just one thread vs the number selected by either the *performance* or *default* model. Using a single thread generally has the least impact on the workload programs.

Figure 6a shows the *slowdown* of the external workload for each approach. When using the *performance* model the slowdown is 4.5%. The OpenMP default policy slows down workload performance by as much as 8.6%. Our predictive model is thus surprisingly less aggressive with respect to the workload than the default policy. This is because in many situations the number of threads is reduced to achieve a better performance and thus the impact on the workload is reduced as well, e.g. `fft`. For programs that achieve their best performance with more threads than there are cores, e.g. `utdsp.edge_detect` or `npb.ep`, using the predictive model results in a greater slowdown.

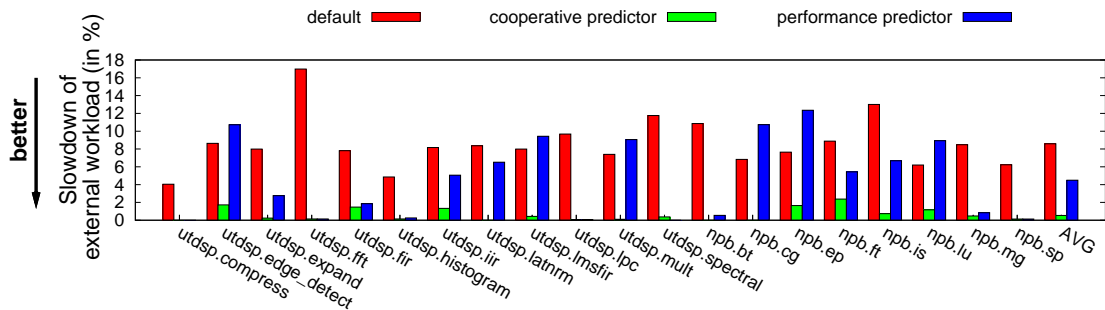
### The cooperative predictor.

The cooperative predictor minimizes impact on workload for all kinds of workloads with an average slowdown of 0.5%. This comes at the cost of lower speedups as can be seen in figure 6b. However, the cooperative predictor still outperforms the OpenMP default with an average speedup of 1.59 over sequential execution compared to 1.40 of the default policy.

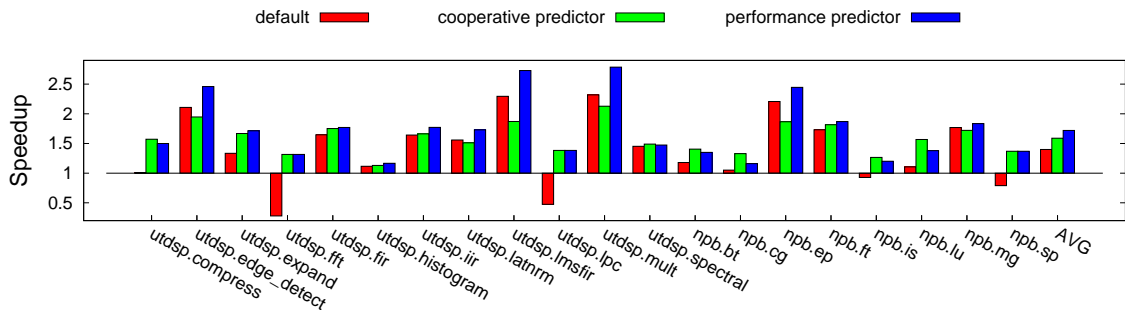
## 6.3 Reducing Feature Collection Cost

As mentioned in section 4.1.1 we evaluated several models based on different sets of features. The full set of features involves running the program five times (one sequential and four parallel runs) on an unloaded multi-core machine (this scheme is called *original* in this section). Because this may be too costly in some situations, we also implemented a model whose program input features are based on only two parallel runs rather than four, namely with 4 and 6 threads (*reduced*). However, it may be the case that it is impossible to run the program on an unloaded machine at all, because it simply may not be available. In this case some of the characteristics of the programs' behaviour can be captured by running the program with multiple threads on just a single core. We evaluated this idea by using only the ratio between the program performance with 1 and 2 threads on a single core as a program input feature to our model (*single-core*).

Figure 7 shows the performance of these approaches av-



(a) The slowdown of the *cooperative* and the *performance* predictor on the workload compared to the compiler’s OpenMP default policy. On average, both predictors are less aggressive than the default policy and the *cooperative* model’s impact is significantly smaller for all benchmarks.



(b) The speedup of the *cooperative* and the *performance* predictor compared to the compiler’s OpenMP default policy. The *cooperative* model improves performance over the default.

Figure 6: The impact on external workload (a) and the performance (b) of the *cooperative* and the *performance* prediction model.

eraged over all programs and benchmarks, as well as the performance of the default strategy, simple adapt and an oracle predictor (optimum). We also show the deviation from the optimum which captures how consistent each approach performs across the benchmarks.

Simple adapt clearly outperforms the default strategy. However, its deviation from the optimum can be very high for some programs as was shown in section 6.1.3. The same is true for the default. Both approaches work in some cases, but perform badly in others. All of our prediction approaches, on the other hand, provide a more consistent performance across all programs. They also give an improved performance on average. Our original approach achieves the highest average speedup 1.66 and is also the most consistent approach. Using a reduced set of features leads to a slightly degraded performance of 1.63 and also has a higher variation. Using only single-core features, our model still outperforms simple adapt with an average speedup of 1.58 compared to 1.45 and it is also significantly more consistent. However, its performance is not as good as in the other models. Hence, there is a trade-off between the cost of the feature collection and the accuracy of the resulting model.

## 6.4 Adapting to Larger Numbers of Cores

To show our approach is not limited to a particular platform, we retrained our model on a new 8-core platform as described in section 3. Once again we randomly sampled workloads this time up to 8 threads/8 programs in size, built a new model and evaluated our technique against the com-

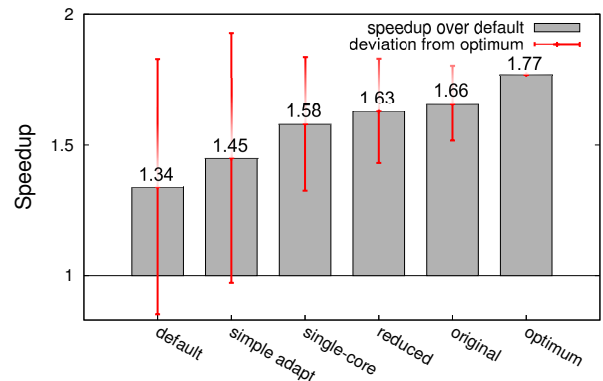


Figure 7: Average speedup over sequential run-time of predictive model with various program feature sets compared to other mapping strategies. The red lines show the performance deviation compared to the optimum.

piler’s OpenMP default policy, which always uses 8 threads on an 8-core machine, on all 23 benchmarks. We reused the program features collected on a 4-core machine, which can be useful if access to a larger unloaded machine is not available. Figure 8 shows that the default policy achieves an average speedup of 2.54 over the sequential execution on an 8-core machine. The prediction model, however, achieves a speedup of 3.39 over sequential execution and thus yields a

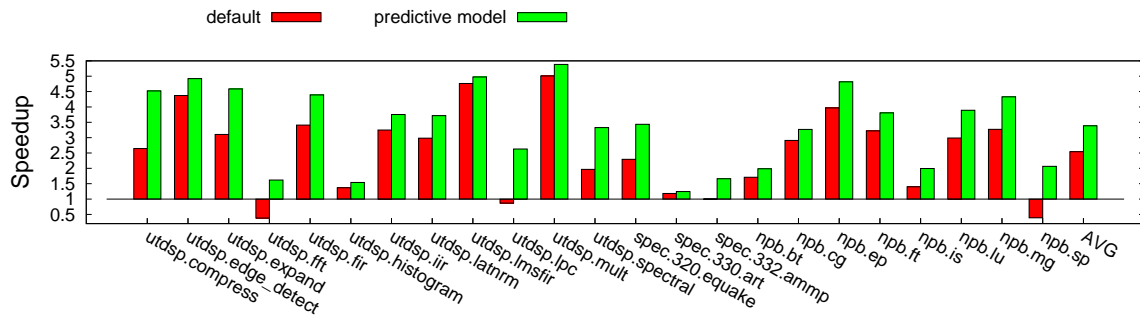


Figure 8: The speedup over sequential execution of the default strategy and the prediction model on an 8-core machine. The predictive model outperforms the OpenMP default for all but one benchmark and achieves a 1.30 times faster execution on average.

1.3 times faster execution than the default strategy. Once again it outperforms OpenMP in all but one case. This shows that our predictive modeling approach is able to port to larger scale platforms automatically.

## 7. RELATED WORK

The mapping of parallel programs is an extensively researched area. There has been significant work in compiler-directed mapping of a single program without external workload [23, 24, 29, 33]. In [23, 24] a technique for mapping programs described as acyclic graphs is presented. In [29] a model-driven approach is used to map parallel programs to distributed systems. Each set of mappings is evaluated using an analytic model, the best being selected at compile time. Carpenter et al. describe an approach for mapping streaming applications to heterogeneous architectures in [7]. Both [18, 33] use predictive modeling to determine the best number of threads for parallel loops. Curtis-Maury et al. [9] build a regression model based on hardware performance counters to determine the best parallel mapping in terms of power and performance. All of these approaches suffer from the inability to model external workload.

To overcome this limitation, others have looked at dynamic mapping at run-time when the system load is known [8, 15, 16]. Corbalan et al. in [8] use speedup at run-time to dynamically allocate processors to programs. However, they only consider programs containing loops that are executed at least dozens of times, in order to collect information about speedups with different configurations and to gradually find the optimal mapping. Our approach works for any kind of data-parallel program, because we don't rely on speedup estimations at run-time and determine the optimal number of threads without any on-line profiling. In [15] a technique for run-time mapping parallel programs to simultaneous multi-threading processors is introduced but assumes no other workloads. Autopin [16] dynamically tries different preset mappings, the best one is chosen. However, this technique is only relevant if the number of program threads is smaller than the number of processing cores and is not suited to external workload. A similar approach is investigated in [12]. Polychronopoulos et al. [22] describe multiple scheduling algorithms for parallel multi-programmed environments. In contrast to our approach they require kernel-level execution and ignore characteristics of the individual programs.

In the area of simultaneous multi-threading processors,

mapping of competing applications is important, because of the high level of resource sharing of co-scheduled programs [31]. Snaveley et al. [26, 27] monitor the performance of programs under different schedules. Based on this information, they predict the schedule that yields optimal overall throughput. In [32] this technique is extended by considering unbalanced schedules. Acosta et al. [1] periodically measure the IPC of programs to find good co-schedules on SMT architectures. In contrast to these dynamic techniques, approaches to improve SMT performance using program analysis [25, 14] and probabilistic models [11] have been proposed. However none of these approaches considers how to improve the performance of an application when given limited resources due to competing jobs.

Predictive modeling is a technique used widely in compiler and architecture design. In [13], ANNs are used to predict the performance of parallel programs, while Barnes et al. [4] use regression to predict the scalability of parallel programs. Predictive Modeling has been used for instruction scheduling by Moss et al. [21], loop unrolling [20, 28] and to speedup iterative compilation Agakov et al. [2]. It is a useful technique as it is by construction portable. None of these prior works has considered its use in mapping programs in the presence of workload.

## 8. CONCLUSION AND FUTURE WORK

This paper has developed an automatic and portable approach to effectively map parallel applications to multi-cores in the presence of external workload. Using an off-line predictive modeling approach based on program and workload features we achieve 93.7% of the maximum speedup available across benchmarks and workloads outperforming the compiler's default OpenMP strategy which occasionally incurs program slowdowns. Our approach gives an average speedup of 1.66 on 4 cores, a factor 1.24 times better than the default. Furthermore, we showed that the slowdown of the external workload is reduced on average compared to the default policy. We also built a cooperative predictor that reduces the impact on the workload to a minimum (0.5%) while still outperforming the default policy by a factor of 1.14. Finally, we evaluated our approach on an 8-core machine giving an average 1.3 times speedup over the default policy. For situations when the full set of features is too costly, a cheaper model that trades off accuracy for feature collection cost but still achieves good performance can be employed.

Future work will investigate more dynamic workloads and applications. We will also investigate incorporating more complex workload features that are externally observable such as cache behavior into our models. Models taking energy consumption into account will also be considered. Finally we will explore the use of such an approach in a heterogeneous multi-core setting.

## Acknowledgements

This work has made use of the resources provided by the Edinburgh Compute and Data Facility (ECDF). (<http://www.ecdf.ed.ac.uk/>). The ECDF is partially supported by the eDIKT initiative (<http://www.edikt.org.uk>).

## 9. REFERENCES

- [1] C. Acosta, F. Cazorla, A. Ramírez, and M. Valero. Thread to core assignment in smt on-chip multiprocessors. In *SBAC-PAD*, 2009.
- [2] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O’Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *CGO*, 2006.
- [3] V. Aslot, M. Domeika, R. Eigenmann, G. Gaertner, W. Jones, and B. Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *WOMPAT*, 2001.
- [4] B. Barnes, B. Rountree, D. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A regression-based approach to scalability prediction. In *ICS*, 2008.
- [5] C. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, Inc., New York, NY, USA, 1995.
- [6] T. Brecht and K. Guha. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 1996.
- [7] P. Carpenter, A. Ramírez, and E. Ayguadé. Mapping stream programs onto heterogeneous multiprocessor systems. In *CASES*, 2009.
- [8] J. Corbalán, X. Martorell, and J. Labarta. Performance-driven processor allocation. In *OSDI*, 2000.
- [9] M. Curtis-Maury, J. Dzierwa, C. Antonopoulos, and D. Nikolopoulos. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *ICS*, 2006.
- [10] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 1998.
- [11] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. In *ASPLOS*, 2010.
- [12] P. K. F. Hölzenspies, J. Hurink, J. Kuper, and G. Smit. Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc). In *DATE*, 2008.
- [13] E. Ipek, B. de Supinski, M. Schulz, and S. McKee. An approach to performance prediction for parallel applications. In *Euro-Par*, 2005.
- [14] Y. Jiang, K. Tian, and X. Shen. Combining locality analysis with online proactive job co-scheduling in chip multiprocessors. In *HiPEAC*, 2009.
- [15] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive execution techniques for smt multiprocessor architectures. In *PPoPP*, 2005.
- [16] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. autopin - automated optimization of thread-to-core pinning on multicore systems. In *HiPEAC*, 2008.
- [17] C. Lee. UTDSP benchmark suite. <http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html>.
- [18] S. Long, G. Fursin, and B. Franke. A cost-aware parallel workload allocation approach based on machine learning techniques. In *NPC*, 2007.
- [19] M. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 1993.
- [20] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. In *AIMSA*, 2002.
- [21] J. Moss, P. Utgoff, J. Cavazos, D. Precup, D. Stefanovic, C. Brodley, and D. Scheeff. Learning to schedule straight-line code. In *NIPS*, 1997.
- [22] E. Polychronopoulos, D. Nikolopoulos, T. Papatheodorou, X. Martorell, J. Labarta, and N. Navarro. An efficient kernel-level scheduling methodology for multiprogrammed shared memory multiprocessors. In *PDCS*, 1999.
- [23] G. Prasanna and B. Musicus. Generalized multiprocessor scheduling using optimal control. In *SPAA*, 1991.
- [24] G. Prasanna and B. Musicus. Generalized multiprocessor scheduling for directed acyclic graphs. In *SC*, 1994.
- [25] S. Sarkar and D. Tullsen. Compiler techniques for reducing data cache miss rate on a multithreaded architecture. In *HiPEAC*, 2008.
- [26] A. Snaveley and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, 2000.
- [27] A. Snaveley, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. In *SIGMETRICS*, 2002.
- [28] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *CGO*, 2005.
- [29] A. Sussman. Model-driven mapping onto distributed memory parallel computers. In *SC*, 1992.
- [30] J. Thomson, M. O’Boyle, B. Franke, and G. Fursin. Reducing training time and confidence calculation using clustering in a machine learning compiler. In *LCPC*, 2009.
- [31] D. Tullsen, S. Eggers, and H. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ISCA*, 1995.
- [32] M. D. Vuyst, R. Kumar, and D. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors. In *IPDPS*, 2006.
- [33] Z. Wang and M. O’Boyle. Mapping parallelism to multi-cores: A machine learning based approach. In *PPoPP*, 2009.