Mapping Parallelism to Multi-cores: A Machine Learning Based Approach

Zheng Wang  Michael F.P. O’Boyle

Member of HiPEAC
School of Informatics, The University of Edinburgh, UK
jason.wangz@ed.ac.uk, mob@inf.ed.ac.uk

Abstract

The efficient mapping of program parallelism to multi-core processors is highly dependent on the underlying architecture. This paper proposes a portable and automatic compiler-based approach to mapping such parallelism using machine learning. It develops two predictors: a data sensitive and a data insensitive predictor to select the best mapping for parallel programs. They predict the number of threads and the scheduling policy for any given program using a model learnt off-line. By using low-cost profiling runs, they predict the mapping for a new unseen program across multiple input data sets. We evaluate our approach by selecting parallelism mapping configurations for OpenMP programs on two representative but different multi-core platforms (the Intel Xeon and the Cell processors). Performance of our technique is stable across programs and architectures. On average, it delivers above 96% performance of the maximum available on both platforms. It achieve, on average, a 37% (up to 17.5 times) performance improvement over the OpenMP runtime default scheme on the Cell platform. Compared to two recent prediction models, our predictors achieve better performance with a significant lower profiling cost.

Categories and Subject Descriptors D.3 [Software]: Programming languages; D.3.4 [Programming languages]: Processors—Compilers, Optimization

General Terms Experimentation, Languages, Performance

Keywords Compiler optimization, Performance modeling, Machine learning, Artificial neural networks, Support vector machine

1. Introduction

Multi-core based processors are widely seen as the most viable means of delivering performance with increasing transistor densities (13). However, this potential can only be realized, in the long-term, if the application programs are suitably parallel. Applications can either be written from scratch in a parallel manner, or, given the large legacy code base, converted from an existing sequential form. Regardless of how applications are parallelized, once the program and hardware. Such models are unable to adapt to different architectures, and the compiler writer has to spend a non-trivial amount of effort in redeveloping a model from architecture to architecture. Online learning (15; 5) is an alternative approach that attempts to overcome these limitations. It is accurate and does not use prior knowledge of the hardware, compiler or program, instead relying on a large amount of profiling of the target program. Although useful for hardware design space modeling (15), it is impractical for compiler based mapping. Thus existing approaches are either restricted in their portability or require excessive profiling.
Our scheme automatically learns from prior knowledge and maps a given parallel program to hardware processors. We demonstrate our technique by building machine learning (ML) based predictors to select mapping configurations (thread numbers and scheduling policies) for OpenMP programs. The predictors are first trained \textit{off-line}. Then, by using code, data, and runtime features extracted from low-cost profiling runs, they predict optimal parallelism configurations for a \textit{new, unseen} program across input data sets.

This paper is organized as follows: section 2 demonstrates that it is a non-trivial task to map a given program to the underlying architecture. Section 3 describes our ML-based predictors and section 4 describes how our predictors are trained and used. Section 5 and 6 discuss our experimental methodology and results. Section 7 discusses related work and is followed by conclusions in section 8.

2. Motivation

This section illustrates that selecting the correct number of threads and how they are scheduled to a multi-core has significant impact on performance. In figure 1 a primary parallel loop in FT (from the NAS parallel benchmark suite) is shown. Despite the simplicity of the code the parallelism mapping decisions are non-trivial.

Consider the graph in figure 2. The x-axis shows the number of processors employed on a SMP platform with 2 Quad-core Intel Xeon processors for an OpenMP program, while the y-axis shows the speedup obtained for this loop. Each of the two lines shows the speedup obtained from the program presented in figure 1 but for two different thread scheduling policies. We see from this figure that the CYCLIC scheduling policy performs better than the BLOCK scheduling policy in some cases. When running this program with 8 processors, the BLOCK scheduling policy – a compiler and OpenMP runtime default scheme, results in performance degradation with a factor of 1.2 times compared with the CYCLIC scheduling policy. The reason is that, for this example, the CYCLIC scheduling policy achieves better load balancing and data locality with a large number of processors. Furthermore, although the CYCLIC scheduling policy exhibits better scalability than the BLOCK scheduling, the performance improvement is small when using more than 6 processors.

Consider now the graph in figure 3. Once again the x-axis shows the number of processors this time for the Cell platform for the same OpenMP program. The y-axis shows the speedup obtained. It illustrates that selecting the right number of threads number is important, if we are to obtain any speedup. In stark contrast to figure 2 there is no performance improvement available when using any number of Synergistic Processing Elements (SPEs) along with one Power Processing Element (PPE). This is because the communication cost is too high to offload the computation to the SPE, a disjoint memory processing unit that has a high communication cost for moving data from the global shared memory to its local memory in a Cell processor. The best parallelism configuration is to spawn an additional worker thread on the PPE (the 1-processor configuration) which supports two SMT threads. Figure 3 also shows that mapping parallelism across platforms is important. Applying the Xeon-optimal mapping scheme (using the maximum number of processors with the CYCLIC scheduling policy) on the Cell platform results in a reduction in performance equals to 75% of the sequential version.

This simple example illustrates that selecting the correct mapping scheme has a significant performance impact, and the optimal mapping scheme may vary from architecture to architecture. Prior research have already revealed that selecting an optimal mapping scheme for a parallel program has large performance gains(28). Therefore, it is crucial for the compiler to find a good mapping scheme for a parallel program, and we need an automatic and portable solution for parallelism mapping.

3. Predictive Modeling

To provide a portable, but automated parallelism mapping solution, we use machine learning to construct a predictor that, once trained by training data (programs) can predict the behavior of an unseen program. This section describes how mappings can be expressed as a predictive modeling problem.
Figure 4. Our ANN model predicts program scalability accurately and then selects the best thread number.

Figure 5. The SVM model constructs hyper-planes for classifying scheduling policies.

3.1 Characterization
The key point of our approach is to build a predictor that can predict scalability and scheduling policies with low profiling cost. We propose two predictors: a data sensitive (DS) predictor and a data insensitive (DI) predictor for tackling programs whose behavior is sensitive to input data sets, and for those who are not. The only difference between these two predictors is in the number of profiling runs needed for extracting the data features (as discussed in section 4.3).

A formal representation of the parallelism mapping problem is described as follows. Let $x$ be a program’s feature, $s$ be the performance curve (scalability) and $p$ be the best scheduling policy. We wish to build a model, $f$, which predicts the scalability, $s$, and the scheduling policy, $p$, i.e., $f(x) = (s, p)$. The closer the performance of the predicted result $(\hat{s}, \hat{p})$ to the performance of the best configuration $(s, p)$ is, the better the model will be. This problem can be further broken down into two sub-problems.

The first is to determine the scalability of a program, and then decide the number of threads allocated to it. The second is to group program features that have similar characteristics into scheduling policy groups. We use two standard machine learning techniques to build two predictors to solve these two problems.

3.2 Model Construction
We use a feed-forward Artificial Neural Network (ANN)(4) to solve the scalability fitting problem, and use a multi-class Support Vector Machine (SVM) model(3) to solve the scheduling policy classification problem.

The ANN model predicts the scalability of a program and selects the optimal thread number for it (as illustrated in figure 4). It is a multi-layer perception which has 2 hidden layers with 3 hidden neurons (nodes) for each layer. The training algorithm for this ANN model is Bayesian regularization backpropagation(4). We use the artificial neural network because of its proven success in modeling both linear and non-linear regression problems, and it is robust to noise(4).

The SVM model attempts to construct hyper-planes in the multi-dimensional feature space, to separate between those occasions when a scheduling policy $p$ is best used and those occasions when it is not. Figure 5 illustrates how this separation could look like in a hypothetical program space. We use a multi-class SVM model with the Radial Basis Function as the kernel because of its proven success in handling both linear and non-linear classification problems(4).

There are alternative approaches to build a performance prediction model, such as analytical(5) and regression-based(2) schemes. We compare our predictors with these two techniques in section 6.

3.3 Training
We use an off-line supervised learning scheme whereby we present the models with pairs of program features and desired mapping decisions, and the models learn how to map parallelism from empirical evidence. Figure 6 illustrates the model training procedure. The collected data is processed into features (as discussed in section 4) as $x$ for the ANN model and the SVM model. Each model finds a class of functions that closely fits the training examples. More precisely, the ANN model finds a function $f$ to take the program features $(x)$ as input and predicts the program scalability $(s)$, i.e., $x \rightarrow s$. The SVM model constructs hyper-planes to separate the classification problem $x \rightarrow p$. Since the training data gathering and the model training procedures are performed off-line, the training process can be done "at the factory" before the compiler is shipped to users.

3.4 Deployment
Once the predictor is trained, it can be used to predict an "unseen" program. Figure 7 illustrates how we use the model for prediction. For a new, previously unseen application, the following steps need to be carried out.
4. Extracting Program Features

The primary distinctive characteristic of our approach is that it uses code, data and runtime features as means to build an accurate predictor. This section describes how essential program characteristics (or features) are extracted from profiling runs.

4.1 The Feature Extractor

Table 1 describes program features used in our predictors. Our code and data feature extractor is built on LLVM(17). The source to source instrumentation used in our feature extractor has lower overhead (less than 25%) than dynamic binary profiling approaches, which may result in 10 to 100 times slowdown(25). In this paper, we run the binary of the instrumented serial code to capture the code and data features. These features could also be extracted using a number of other approaches, such as the dynamic instrumentation approach(23), without affecting the prediction accuracy of our approach.

4.2 Code Features

Code features are derived directly from the source code. Rather than using this high level information directly, we post-process and group these features into four separate groups to characterize a program, as shown in figure 8.

4.3 Data Features

Some dynamic features are dependent on input data sets, such as the L1 data cache and the branch miss rates. Counters are inserted into the serial code to record the loop iteration count of the profiled input data set. Through this profiling run, we also record the L1 data cache and the Branch miss rates from hardware performance counters that are widely supported in modern processors.

For the DI predictor, the feature extractor predicts the potential dynamic features based on features extracted from the smallest input data set. It predicts program behaviors for a larger loop iteration count by using features extracted from the smallest input data set and prior knowledge that learned from training data. The DI predictor assumes that the program behavior does not have too much variability across data sets at a fine-grain parallel level. Of course, there may be cases where the behavior of a program is dependent on the input data set. Hence, we also propose the DS predictor that can handle the problem in this situation.

The DS predictor uses one profiling run on the sequential program for each data set in order to extract data set information. This translates into profiling with 7% and 1% of the total number of possible parallelism configurations on the Intel platform and the Cell platform respectively. It is obvious that the DS predictor has better performance than the DI predictor because it extracts the data set features more accurately (as presented in section 6).

4.4 Runtime Features

As any "block-box" system, our predictors need to understand how the compiler parallelizes a program. They get this information by investigating the execution time of a parallelized program. We obtain the execution time by using one profiling run with the parallel version of the program. This profiling run can be done with arbitrary number of processors that allows the user to find a tradeoff between hardware resources and the profiling overhead. In this paper, the feature extractor parallelizes the program with the CYCLIC scheduling policy. On the Intel platform, this parallelized program is profiled with the maximum number of processors that has the least profiling overhead for most cases. On the Cell platform, the parallelized program is profiled with a SPE, because the SPE is a critical resource in the Cell processor.

4.5 Summary

In summary, our ML-based predictors use profiling information to characterize code, data and runtime features of a given program. The feature extractor needs several profiling runs for a program to extract these features. For a previously unseen program, we firstly instrument the serial code for counting operations, and then we perform a profiling run for this sequential instrumented version to obtain the code and data features. Secondly, the program is parallelized and profiled with the smallest input data set in order to extract the runtime features.

A surprising result is that the DI predictor produces accurate results by only profiling with the smallest input data set. This is because, as found in our experiments, at a fine-grain level, the parallel program behaviors does not exhibit significant variability across data sets. Some programs are sensitive to input data sets, and they can be handled by the DS predictor at the cost of an additional profiling run with each input data set. Figure 9 shows the number of profiling runs needed by each predictor.

5. Experimental Methodology

This section introduces platforms, compilers, and benchmarks used in our experiments as well as the evaluation methodology.

5.1 The Experimental Platform

In order to show that our approach works across different types of multi-cores, we targeted both a shared memory and a distributed memory platforms. The first target is a shared memory homogeneous machine with two quad-core Intel Xeon processors, supporting up to 8 threads. The second, in contrast, is a QS20 Cell blade.
We evaluated our predictors on 20 programs from three benchmark suites: NPB(10), UTDSP(18), and Mibench(12), as shown in table 3. These parallel workloads represent widely used computation kernels from embedded, high performance, and commercial applications. We omitted those programs from UTDSP and Mibench which have loop carried dependence on their primary time-consuming loops. Most programs in Mibench have function-pointers and pointer indirection that could not be handled by the IBM Xlc compiler properly. This is due to the multiple address spaces of the Cell processor, which prevents us from compiling the programs on the Cell Blade Server.

Table 3. Programs used in our experiments.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Program</th>
<th>Benchmark</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mibench</td>
<td>stringsearch</td>
<td>Mibench</td>
<td>susan</td>
</tr>
<tr>
<td>UTDSP</td>
<td>lpc</td>
<td>UTDSP</td>
<td>histogram</td>
</tr>
<tr>
<td>NPB</td>
<td>CG</td>
<td>NPB</td>
<td>FT</td>
</tr>
<tr>
<td>NPB</td>
<td>FT</td>
<td>NPB</td>
<td>BFT</td>
</tr>
<tr>
<td>NPB</td>
<td>LMSFIR</td>
<td>NPB</td>
<td>MG</td>
</tr>
<tr>
<td>NPB</td>
<td>IIR</td>
<td>UTDSP</td>
<td>Coqeeze</td>
</tr>
<tr>
<td>UTDSP</td>
<td>detect</td>
<td>UTDSP</td>
<td>histogram</td>
</tr>
<tr>
<td>UTDSP</td>
<td>Compress</td>
<td>UTDSP</td>
<td>SPP</td>
</tr>
<tr>
<td>UTDSP</td>
<td>Lpc</td>
<td>UTDSP</td>
<td>multi</td>
</tr>
</tbody>
</table>

5.2 Benchmarks

We evaluated our predictors on 20 programs from three benchmark suites: UTDSP(18), the NAS parallel benchmark (NPB(10), and Mibench(12), as shown in table 3. These parallel workloads represent widely used computation kernels from embedded, high performance, and commercial applications. We omitted those programs from UTDSP and Mibench which have loop carried dependence on their primary time-consuming loops. Most programs in Mibench have function-pointers and pointer indirection that could not be handled by the IBM Xlc compiler properly. This is due to the multiple address spaces of the Cell processor, which prevents us from compiling the programs on the Cell Blade Server.

As for selecting the scheduling policy, these two models need to predict the execution time of a parallel program with different scheduling policies. This means they need to profile different parallel versions of a program that compiled with each scheduling policy respectively. By ranking the predictive execution time, these models are able to predict the optimal mapping scheme for a given program.
large number of input data sets. We present the prediction accuracy of each program as the prediction performance relative to the upper bound. For each program, we report the result by averaging performance for all input data sets. We exhaustively executed each program with all possible mapping schemes to empirically find the actual optimum. Therefore, the gap between the predicted optimum and the actual optimum is calculated based on the actual performance of programs.

6. Experimental Results

This section first evaluates the maximum performance achievable from selecting the best number of threads and scheduling policy and, as such, provides an upper-bound on performance with which to evaluate our approach. We then evaluate our ML-based predictors against the OpenMP runtime default scheme across data sets showing that it consistently outperforms it. Next, we compare our approach against two recently proposed prediction models on two platforms. Finally, we evaluate the profile overhead required by each of these approaches and show that our predictors deliver the best performance and reduce profiling costs by factors of at least 4 times.

6.1 Upper Bound Performance

Table 4 shows the upper bound speedup with the largest data set for each program relative to the sequential version on both the Intel and the Cell platforms. It some instances it is not profitable to parallelize so their upper bound speedups are 1. In other cases speedups up to 7.99 and 7.79 are available on the Intel and the Cell platforms respectively.

Selecting the right mapping scheme has a significant performance impact on a parallel program. On average, the worst scheduling policy results in 2.7 (up to 42) and 2.1 (up to 29) times performance degradation on the Intel and the Cell platforms respectively. Moreover, there is no single scheduling policy that consistently outperforms others for all programs across architectures. Considering the number of threads with scheduling policies together, on average, the worst mapping configuration results in 6.6 (up to 95) and 18.7 (up to 103) times performance degradation on the Intel and Cell platforms respectively. So, there is significant performance improvement available, but it is important not to select a bad mapping configuration.

6.2 Comparison with Default Scheme

In this section we compare our ML-based predictors with the OpenMP runtime default scheme on two different platforms.

6.2.1 Performance Comparison

Figure 10 shows that ML-based predictors not only have better mean performance but also have better stability across programs than the OpenMP runtime scheme across platforms. On average, the ML-based approach has better performance than the OpenMP runtime scheme, above 96% on both platforms compared to 95% and 70% for the OpenMP’s default scheme on the Intel and the Cell platforms respectively. This figure illustrates the advantage of our approach against a fixed mapping strategy – the ML-based approach adapts to programs, data sets and platforms.

Xeon Platform. On the Intel platform (as shown in figure 10(a)), the ML-based approach has slightly better performance than the OpenMP runtime scheme. This is mainly because the Intel icc compiler has been well optimized for OpenMP programs that used in our experiments on the Intel platform(27). Even though, the icc OpenMP runtime default scheme still results in great performance degradation in some programs (such as NPB.FT.L2, NPB.FT.L3, and NPB.MG.L1), in which it only delivers 57% to 75% performance relative to the upper bound. In contrast to the OpenMP runtime default scheme, the ML-based approach delivers stable performance across data sets. It achieves at least 95% performance to the upper bound for any program used in our experiments, which demonstrates the stability of our approach across programs.

Cell Platform. Figure 10(b) compares the performance of the OpenMP runtime scheme against our ML-based approach. Unlike the Intel platform, the OpenMP default scheme does not deliver high performance on the Cell platform. Performance of our ML-based approach on the Cell platform is as good as its on the Intel platform, which shows the stability of our approach across platforms. On average, the ML-based approach has better performance than the OpenMP runtime scheme, 96% and 97% for the two ML approaches compared to 70% for OpenMP’s default scheme. In one case (NPB.CGL.L3) the performance improvement is 17.5 times greater than the IBM Xlc OpenMP runtime default scheme. Although the DI predictor has slightly worse performance than the runtime default scheme in some programs, the performance gap relative to the OpenMP runtime default scheme is small (0.01%
architecture behaviors from the training data automatically. This results in better adaptation across data sets. 

**6.2.2 Stability Comparison**

Box-plots in figure 11 summarize the performance gap to the upper bound for the OpenMP default scheme and ML-based predictors across programs and data sets. The longer the “whisker” a model has, the less stable it has. The graph clearly shows that performance of the ML-based predictors is not only stable across programs and data sets, but also stable across architectures. It also shows that the DS predictor is stabler than the DI. This is because the DS predictor use extract profiling runs to obtain the data and dynamic features for a program with different input data sets that results in better adaptation across data sets.

Although our ML-based predictors were not aware of the homogeneous or heterogeneous characteristics of the underlying hardware when they were constructed, they learned the program and architecture behaviors from the training data automatically. This demonstrates the strength of an automatic model—it frees compiler developers from tremendous effort in tuning the heuristic from architecture to architecture (and from program to program) by learning based on empirical evidence.

**6.3 Comparison with Other Models**

This section compares our approach against the regression-based model and MMGP approach. In this experiment, we assumed that MMGP and the regression-based model always pick the best scheduling policy. This allows a fair comparison.

**Xeon Platform.** We only compare our approach with the regression-based model on the Intel platform, because MMGP does not target this platform.

According to figure 12, the ML-based predictors have consistently better performance and greater stability across programs compared to the regression-based model. On average, the ML-based predictors outperform the regression-based model which only delivers 80% performance to the upper bound. Despite assuming it always picks the best scheduling policy, the regression-based model slows down performance of some programs significantly. For example, it delivers only 3.3% performance to the upper bound for NPB.CG.L3. This is because the regression-based model can not choose the correct number of threads for these programs.

**Cell Platform.** On the Cell platform, we compare our approach with MMGP, a performance model particularly targets the Cell processor, and the regression-based model. Consider the graph in figure 13, where the DI predictor (the DS predictor but is not shown to aid clarity) has a better average performance as well as greater stability across the programs compared to both MMGP and the regression-based model.

In this experiment we assumed that both models can choose the best scheduling policy, which means that they should produce
better performance results than our predictors. However, on average ML-based predictors outperform these two models by delivering 96% (for the DI predictor) and 97% (for the DS predictor) of performance of the upper bound, while MMGP and the regression-based model have 76% and 69% of performance of the upper bound respectively. Moreover, our predictors enjoy stable performance for most programs except for one loop in FT and one loop in LU. The relatively low performance of these two loops is because of their unique characteristics compared to the training data sets, and this can be improved by continuous learning or using more training data sets. In contrast to our predictors, the regression-based model has its poorest prediction achieving just 4% of the upper bound, and MMGP’s poorest prediction achieving 18% of the upper bound.

6.4 Summary of Prediction Results

Section 6.2 and 6.3 demonstrate two advantages of our approach compared with hand-crafted heuristics (MMGP and the OpenMP runtime default scheme) and online learning (the regression-based model) approaches. Firstly, our approach has the best performance on average. Secondly, its performance is stable across programs, data sets and architectures.

6.5 Profiling Cost

Profiling cost is a critical metric in evaluating the effectiveness of any prediction model. This section discusses the profiling cost of the regression-based model, MMGP, and the ML-based predictors.

As shown in table 5, our approach has the lowest profiling cost among these models. The DI predictor profiles only 0.03% and 0.06% of all possible parallelism configuration on the Intel and the Cell platform respectively. The DS predictor needs additional profiling runs than the DI predictor and has a profiling overhead of 7.18% on the Intel platform and 1.16% on the Cell platform. However, when compared to the regression-based model, it still reduces the profiling overhead by factors of 4 and 12 times on the Intel and the Cell platforms respectively. Comparing to MMGP on the Cell platform, the DI predictor and the DS predictor significantly reduce the profiling overhead by factors of 29 and 512 times respectively.

Figure 14 shows the profiling cost per program for each model (the profiling cost of the DI predictor is very low and is not shown in figure 14(a) to aid clarity). The profiling cost of the DS predictor ranges from 0.2% to 13% on the Intel platform and ranges from 0.08% to 4.7% on the Cell platform. Overall, our predictors have lower profiling cost than the regression-based model and MMGP, which have to go through a large portion of all possible mapping configurations to make a prediction.

6.5.1 Summary of Profiling Cost

The ML-based approach has a fixed number of profiling runs regardless of the number of processors and scheduling policies it considers. This is an important advantage that guarantees the scalability of our approach in a large design space – introduced by future multi-

Table 5. Absolute profiling overhead for each model.

<table>
<thead>
<tr>
<th>Model</th>
<th>Intel</th>
<th>Cell</th>
</tr>
</thead>
<tbody>
<tr>
<td>DI predictor</td>
<td>4.79secs</td>
<td>4.80mins</td>
</tr>
<tr>
<td>DS predictor</td>
<td>13.10mins</td>
<td>1.75hours</td>
</tr>
<tr>
<td>Regression-based model</td>
<td>39mins</td>
<td>16.45hours</td>
</tr>
<tr>
<td>MMGP</td>
<td>N.A.</td>
<td>N.A.</td>
</tr>
</tbody>
</table>
Our two ML-based predictors are learnt with off-line training. In this paper, we used some program runs (627 runs on the Intel platform, and 1311 runs on the Cell platform) to collect training data and build our model. The total program runs and training process caused two days by using four machines. This is a one-off cost amortized over all future users of the system and represents less time than is usually needed to develop a hand-tuned heuristic.

7. Related Work

Task scheduling and performance modeling have an extensive literature. Prior research mainly focused on heuristics and analytical models(19; 10), runtime adaption(8), and online training models(15; 2) of mapping or migrating task(26) on a specific platform. Rather than proposing a new task mapping or performance modeling technique for a particular platform, this paper aims to develop a compiler-based, automatic, and portable approach that learns how to best use of existing compilers and runtime systems.

Task Mapping in Multiprocessors. Ramanujam and Sadayapan (26) used heuristics to solve the task mapping problems in distributed memory systems. Runtime list-based heuristics give high priorities to critical tasks in a task dependence graph(24). These approaches target on runtime scheduling of different tasks rather than processor allocation for a parallel task.

Analytical Performance Modeling. Gabriel et al (11) and Sharapov et al (14) present two systematic methods to predict application performance. Their approaches rely on low level hardware detail, which are only feasible to analyze on a limited set of applications and architectures.

Several analytical models for modeling parallel performance have been proposed, such as (19),(10), and (5). These models use low-level program and architecture information to model computational phases of a parallel program. However they require user-provided program and architecture characteristics, preventing their portability across platforms.

The OpenUH compiler(20) uses a static cost model for evaluating the cost for parallelizing OpenMP programs. This model is built on static analysis which does not adapt to input data sets, and targets a particular platform.

Runtime Adaption. Corbalan et al (8) measure performance and perform processor allocation for loops at runtime. The adaptive loop scheduler(28) selects both the number of threads and the scheduling policy for a parallel region in SMPs through runtime decisions. In contrast to these runtime approaches, this paper presents a static processor allocation scheme which is performed at compilation time.

Statistical Performance Prediction. Ipek et al(15) use an artificial neural network to predict a high performance application-
SMG 2000 with an error of less than 10%. However, their model needs a fairly large number of sample points for a single program, which introduces a high profiling cost. Recently, Bernhard et al. (2) demonstrate the regression-based model can be used to predict the scalability of an OpenMP program. Their model trained on a small subset of processors, predicts execution time for a large number of processors. In contrast to our technique, these models do not learn from prior knowledge which requires more expensive online training and profiling costs for a new program.

Adaptive Compilation. Cooper et al. (7) develop a technique to find a "good" compiler optimization sequence that reduces the code size. The shortcoming of this model is that it has to be repeated for each program to decide the best optimization sequence for that program. Cavazos et al. (6) use performance counters to generate compiler heuristics for predicting code transformation strategies for a sequential program. Recently, Long et al. (22) used machine learning to determine the thread number allocated for a parallel java loop on the runtime. This approach does not tackle with the portability issue and does not adapt to different input data sets. Furthermore, they do not perform any limit study.

In contrast to prior research, we built a model that learns how to effectively map parallelism to multi-core platforms with existing compilers and runtime systems. The model is automatically constructed and trained off-line, and parallelism decisions are made at compilation time.

8. Conclusions and Future Work
This paper has presented two portable machine learning (ML) based predictors for the compiler to map parallel programs to multi-cores. We demonstrated our predictors by predicting fine-grain parallelism mappings of OpenMP programs. By using code, data and runtime features, our predictors predict the optimal number of threads and the optimal scheduling policy for mapping a new, unseen program to the underlying hardware. In contrast to classical performance prediction approaches, our approach focuses on finding optimal parallelism mappings for programs across data sets. Our model is built and trained off-line, and is fully automatic. We evaluate our ML-based predictors by comparing them with two state-of-the-art performance prediction models: an analytical model and a regression-based model as well as the OpenMP runtime default scheme. Experimental results on two different multi-core platforms (Intel Xeon and the Cell processors) show that our predictors not only produce the best performance on average but also have the greatest stability across programs, data sets and architectures. Compared to two recent performance prediction models, our ML-based predictors reduce the profiling cost for a new program significantly by factors ranging from 4 to 512 times.

Future work will combine our predictors with code transformation and compiler options to optimize parallel programs. Furthermore, we will apply our predictors to other types of parallelism, such as software pipelining.

Acknowledgments
This research is partially supported by the European MILEPOST and SARC projects under the 7th Framework Programme. We would like to thank to Barcelona Supercomputing Center for the use of their Cell/B.E. machines.

References