Smart, Adaptive Mapping of Parallelism in the Presence of External Workload

Murali Krishna Emani Zheng Wang Michael F.P. O'Boyle

School of Informatics, The University of Edinburgh, UK

m.k.emani@sms.ed.ac.uk, zh.wang@ed.ac.uk, mob@inf.ed.ac.uk

Abstract

Given the wide scale adoption of multi-cores in main stream computing, parallel programs rarely execute in isolation and have to share the platform with other applications that compete for resources. If the external workload is not considered when mapping a program, it leads to a significant drop in performance. This paper describes an automatic approach that combines compile-time knowledge of the program with dynamic runtime workload information to determine the best adaptive mapping of programs to available resources. This approach delivers increased performance for the target application without penalizing the existing workload. This approach is evaluated on NAS and SpecOMP parallel benchmark programs across a wide range of workload scenarios. On average, our approach achieves performance gain of 1.5x over a state-of-art scheme on a 12 core machine.

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, Runtime adaptation, Machine Learning

1. Introduction

Multicore-based parallel systems now dominate the computing landscape from data centers to mobile devices. Their effective use is the main challenge for system designers. Robust compiler technology to exploit the potential of multicores is still lacking despite the large body of research in automatic parallelization, mapping, scheduling and memory hierarchy optimization.

While the plausibility of truly automatic discovery of parallelization remains contentious, there has been much success in compiler-directed mapping and scheduling of parallelism; determining the number of parallel threads and how parallel work is best assigned to them. However, one widely used simplifying assumptions is that the target machine is fully available with no competing workloads and that resources are static throughout the lifetime of the application. This assumption may remain true for high-performance computing applications but for the vast majority of platforms, it is not the case.

Given that resources such as the number of idle processors available varies and has a dramatic impact on the correct mapping and scheduling of work, entirely static compiler driven approaches are likely to fail. The reason for such a simplifying assumption is clear. At compile-time we do not know the dynamic runtime environment, so it seems reasonable for such an issue to be assigned to the runtime or operating system. which, unlike like the compiler, are concerned with other issues such as fair and secure managing of all jobs and resources, which may not be aligned with the compiler's objective. From a compiler's perspective, what is fair for all may not be best for the chosen program. Or put more constructively, the best compiler policy for an application needs to consider what the o/s behavior is

Dynamic runtime scheduling that takes into consideration other workloads is an obvious way to match program parallelism to available resources: expanding when there is space available, shrinking when resources are used up. However such approaches are generic, with no specific knowledge of the program structure. Ideally, we would like to combine all the knowledge a compiler has of a particular program with the runtime's knowledge of the dynamic environment to make the right mapping decisions as both the program and the environment evolve.

The tension between the program centric view of a compiler and the operating system comes to a head when a program tries to demand too many system resources and is penalized. A compiler that tries to maximize its program's performance may inadvertently adversely affect it. It seems sensible therefore for a compiler and runtime to improve its program's performance without adversely affecting the external workload.

CGO'13 23-27 February 2013, Shenzhen China.

978-1-4673-5525-4/13/$31.00 ©2013 IEEE. . . $15.00

This paper develops a dynamic mapping policy that takes static compiler information of a program and runtime system utilization to determine the best number of threads for an application. Specifically, we redesign the OpenMP runtime library that determines the number of threads allocated at the start of a parallel region. The new library examines the state of the runtime environment and combines this with compiler analysis of the program and uses a heuristic to determine the right number of threads. This heuristic is automatically generated using machine learning off-line at the factory based on synthetic workloads and artificial neural network. This avoids the pitfalls of using a hard-wired heuristic that requires human modification whenever the hardware changes. Our approach dynamically adjusts the number of threads used for an application as the external workload changes.

It aims to maximize the performance for the target program with minimal disruption to external tasks. We evaluated this approach in a live dynamic workload environment where workloads vary in both frequency and resource utilization. On a 12-core multi-core system, our approach delivers on average, across all workloads a 1.5x speedup over the state-of-the-art dynamic scheduling approach. Furthermore, it never degrades the performance of either the target or the external workload.

2. Example

This section provides an illustrative example showing that the best mapping of a parallel application depends on the dynamic system workload and that determining the best number of threads for the target application is non-trivial.

Consider the simplified scenario shown in figure 1(a) where the upper and lower diagrams describe the target and workload behavior over time. The diagrams describe a long running OpenMP program *LU* coscheduled with varying external workloads: W_1 with 8 threads and W_2 with 6 threads on a 12 core machine.

The upper diagram describes the behavior of *LU* over time using 2 distinct scheduling policies: the OpenMP default scheme and a state-of-the-art online adaptation technique [15]. OpenMP default scheme is to always allocate same number of threads as the number of cores, 12. This is unlikely to be ideal where it achieves a speed up of just 1.2. In fact any static scheme performs poorly. The best static number of threads for this workload scenario is 4 which gives just a 1.4x speedup.

The online adaptation approach performs much better, reacting to the workload, increasing or decreasing the number of threads of each parallel section of the target program until no further performance improvement is observed. As can be seen in figure 1(c), it achieves considerable improvement over the static scheme giving a speedup of 2.13x on average.

Figure 1(b) shows a zoom-in view for the first 4 seconds.

If we exhaustively replay this workload scenario and evaluate all possible number of threads at each dynamic invocation of a parallel section, we have a good measure of the ideal thread allocation policy representing the maximum available performance of any scheme. Figure 1(b) shows ideal target thread configuration for maximum achievable speedup. It can be observed that this speedup can be achieved by allocating different thread number for parallel sections depending on the existing workload and switch instantly to maximum available processors once workload ceases to execute. Figure 1(c) shows that if we were able to select the ideal number of threads over time, this will result in a significant 5x speedup.

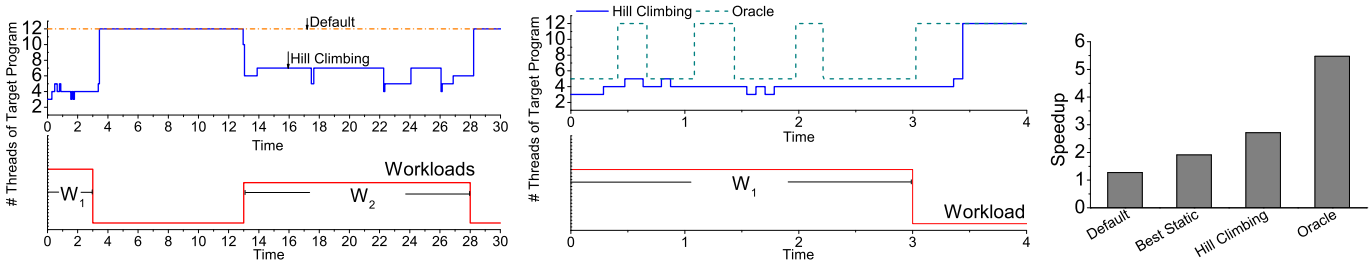
So, although the dynamic online approach achieves better performance than a static policy, there is still significant room for improvement. While it is able to exploit medium term variation such as free space on the machine, it is less able to perform well with the short term variation of coscheduled workloads. However, server utilization varies over very short time scales [13]. What is needed is a scheme that can quickly adapt to the dynamic behavior of external workload and to close the gap between state-of-the-art online dynamic schemes and the maximum available performance. This paper aims to develop such a scheme.

3. How Much Room for Improvement

Although the previous section shows that for one example current approaches can be improved, it is important to see if this is more widely the case. We conducted a limit study comparing existing approaches to an idealised oracle. To make the experiment tractable we have created a synthetic reproducible workload environment and evaluated the behaviour of scheduling policies on NAS OpenMP benchmarks. In our later evaluation (section 6) we consider a more dynamic setting, but this synthetic setting is useful as a limit study.

Figure 2 shows the average speedup achieved (and standard deviation) over sequential execution on a 12-core multicore system (see section 5) with each workload. The workloads consists of just one other program from the benchmark suite executed with 1 to 12 threads. So, each bar in the diagram therefore corresponds to the performance of the target program averaged over $8 \times 12 = 96$ distinct workloads runs with different scheduling policies, repeated 10 times. So it incurred an one-off cost of 8*8*12*12=9216 experiments.

To provide an upper bound on performance, we then evaluated all possible scheduling decisions (thread number) at each parallel region for each target program for all workloads. Clearly in a dynamic live environment, no scheduler could do this, but this exhaustive controlled experiment gives us an insight into the performance of current schemes. We compared this ideal to the default approach and a state of the art online gradient-based approach [15]. As can be seen from the figure 2, the default approach performs poorly having little improvement across all benchmarks, the one exception being *ep*. Given that on average that there will be 6 workload threads, the default policy saturates the machine causing slowdown. The hill climbing approach outperforms



(a) Reaction to Workload (Panoramic View) (b) Reaction to Workload (Zoom in View) (c) Achieved Performance

Figure 1: Target thread configurations implemented by different schemes in reaction to change in workload. The close view (b) compares default, hill climbing schemes with the best possible thread configuration. Corresponding speedup graphs (c) show the scope for improvement

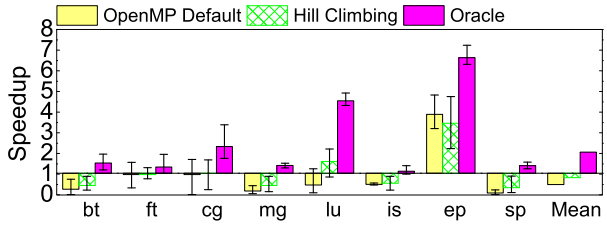


Figure 2: Performance comparison of existing approaches. There is significant room for performance improvement

the static default approach but on average gives no speedup due to it reacting too slowly to workload variation. The ideal scheme, however, delivers 2x speedup on average showing that even in the restricted case of running one program with just one external workload program there is significant room for improvement.

Given these results, the next section looks at developing a heuristic that can select the right number of threads to improve over the default and hill climbing approaches using program and runtime information.

4. Automatic Heuristic Generation

Our goal is to build a heuristic that chooses the right number of threads for each parallel section of the target program based on characteristics of the program and the runtime workload. Instead of hand-crafting a heuristic that is tightly coupled to a particular environment and workload setting, we use *supervised learning* to automatically generate a heuristic that can be ported to any hardware platform.

Figure 3 describes how our approach works. At *compile time*, the compiler extracts information about the program in the form of static program features (described in section 4.1). It then links the compiled program with a runtime library which consists of an automatically learned heuristic. For each parallel section, the compiler inserts a call to the built-in runtime where the static program features of that parallel section are passed as a parameter. At *execution time*, the runtime combines these program features with dynamic external workload information as inputs to an automatically

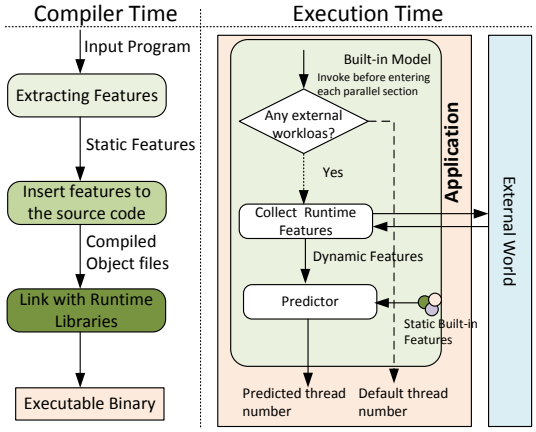


Figure 3: Apply our techniques at compilation and runtime.

learned heuristic that returns optimal number of threads for this parallel section.

Building and using such a heuristic follows a well-known three-step for supervised learning: (1) generate training data (2) train a model or heuristic (3)use the heuristic. We generate training data by exhaustively running each training program together with a workload program.

During the generation of training data we gather a set of *features* to characterize the target program and workload behaviors which are then used to train a heuristic which is later evaluated in an unseen setting.

4.1 Program and Workload Features

We use a set of numerical values to form a feature vector,v to represent the static program features and dynamic runtime workload.

Program Features The set of features used, described in table 1 consists of static instruction, memory and branch summary information where the corresponding values are normalized to the total number of instructions. Given that the regions of interest are always parallel, this information is sufficient to differentiate across programs for thread selection.

Type	Feature	Type	Feature
Static	Load/Store count	*Static*	Branch count
Static	Instruction count	*Dynamic*	Processors
Runtime	# Workload threads	*Runtime*	Run queue length
Runtime	ldavg-1	*Runtime*	ldavg-5

Table 1: List of features

Workload Features We use a set of Linux profiling tools to collect dynamic workload features. To characterize the runtime environment, we use three features from */proc* filesystem: *run queue*, *ldavg-1*, *ldavg-5*. The *run queue length* represents the number of processes waiting for scheduling in the Linux kernel. The *ldavg-n* is system load average calculated as the average number of runnable or running tasks over an interval of *n* ($n = 1, 5$) minutes. In essence, the runtime features capture the current system load and the impact of external workload. In addition to these metrics, our feature set also consists of the number of workload threads and the total number of physical processors. These 8 features are organized as a feature vector that is the input of our machine learning heuristic.

4.2 Generating Training Data

Unlike previous approaches where the model is trained for each target program [14], we train our heuristic using training data collected from a synthetic workload setting and apply the trained heuristic to various unseen dynamic runtime environments. Training data are generated by running the target training program together with a single workload application with varying number of threads. To find out the optimal mapping strategy for the target program under a given workload configuration, we exhaustively assign different thread counts for the target program and record the best performing thread setting. We extract runtime features during the training run. Those runtime features and the best-performing thread number, t_{best}, are put together to form the training data set, $\{v_i; t_{best,i}\}, i = 1, ..., N$.

Although producing training data takes time, it is only an one off cost incurred by our heuristic. The model is trained only once offline and frozen and no further learning takes place during program execution.

4.3 Building the Heuristic

Our heuristic is based on an artificial neural network [5]. We employ the standard Multilayer Perceptron with 1 hidden layer that learns by back propagation algorithm.

We supply the training algorithm with training data collected offline. Each such data item includes the static program features, the runtime features and the best mapping. The training algorithm tries to find a function f which, takes in a feature set, v, and gives a prediction, $\overrightarrow{t}$, that closely matches actual best mapping, t_{best} in the training data set.

4.4 Runtime Use

Once we have gathered training data and built the heuristic we can use it to select the mapping for any *unseen*, *new* program. During execution time, the library is called and checks whether there is a workload program running on the system. If any workload program is detected, runtime features from */proc* are collected and act as inputs to the neural network which outputs the optimal number of threads for the target program. The runtime uses this number of threads to execute the corresponding parallel region. If there is no workload, the target program runs with default configuration using all available physical threads.

Category	Number of programs	Total workload threads
Light	<2	<6
Medium	[2-5]	[6-12]
Heavy	>5	>12

Table 2: Workload Settings

5. Experimental Methodology

Hardware and Software Configurations Our experiments were carried on an Intel Xeon platform with two 2.4 GHz six-core processors (12 threads in total) and 16GB RAM. The operating system was Red Hat 4.1.2-50 running Linux kernel 2.6.18. The programs were compiled using *gcc* 4.6 with parameters "-O3 -fopenmp".

Benchmarks We used the NAS parallel benchmark suite [2] and all the C benchmarks of the SPEC OMP 2006 suite [3] to evaluate our approach. These programs are representative parallel programs, which are a good candidate for evaluating this work.

Workloads We classify workloads into three categories: *light*, *medium* and *heavy* depending on the number of external programs and the total number of threads used by them. Table 2 lists the setting of each category. In addition, our settings consist of two types of workload arrival patterns: low-frequency and high-frequency where workload programs arrive at 2, 10 second intervals respectively.

The combination of three types of workloads and two types of arrival patterns results in six workload scenarios.

5.1 Comparison

We compare our machine learning based approach with the OpenMP default scheme and state-of-the-art hill-climbing technique. **Default:** The OpenMP runtime selects the number of threads to be equal to the number of processors by default. **Hill Climbing:** A state-of-the-art online adaptation approach that uses hill climbing technique is proposed by Raman et al. [15].

5.2 Methodology

For each target program, we repeated each experimental run for 10 times and averaging the measured execution time. Each program ran with different workload programs and we report the average speedups across different workload program sets.

We evaluated our approach using standard leave-one-out *cross-validation* technique. This means we remove the program to be evaluated from the training program set and then

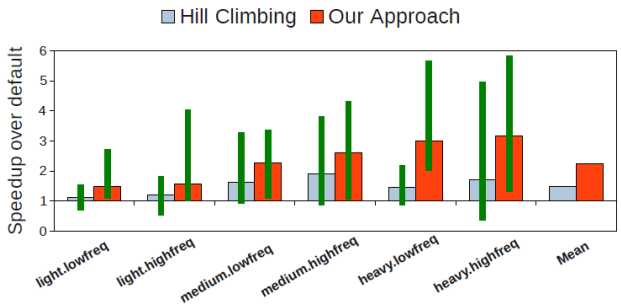


Figure 4: Comparison of speedups of hill climbing and our approach over the OpenMP default scheme under different workload scenarios averaged across all target programs. The min-max bars show the ***ranges*** across different target programs. On average our automatic approach outperforms the hill climbing scheme (2.25x vs 1.48x) and does not slow down the target program in any case.

build a model based on the *remaining* programs. This ensures the model always predicts on an unseen program. Unlike previous work where the model has to trained on each environment [9, 14], our machine learning model is trained on a static pair-wise workload setting and is evaluated on completely *new, dynamic* workload settings that have not been seen in the training stage.

6. Experimental Results

We first summarize the performance of our approach against alternative approaches across all workload settings. Next we give detailed comparison for each workload setting. Then, we evaluate our approach on a workload scenario that is derived from a large scale warehouse system. Finally we evaluate the prediction accuracy of our model by comparing its prediction to the oracle scheme using the training data.

6.1 Overall Results

Figure 4 shows the performance results on 6 different workload scenarios averaged across all benchmark programs. In a given workload setting, the speedup improvement varies for different programs. Hence, the min-max bars in this graph show the *range* of speedups achieved across various target programs.

Hill Climbing The hill climbing achieves a mean speedup improvement of 1.47x. For workload settings consisting of light programs, its improvement is small (less than 1.3x). For medium and heavy workload settings, it is able to greatly improve the OpenMP default scheme with speedups over 1.5x. However, by looking closely to the min-max bars, we can see that this approach may actually slowdown some target programs by giving a speedup below 1. This is in particular for high frequent workload programs owing to slow reaction to this environment.

Automatic Our approach not only gives better performance when compare to OpenMP default but also significantly outperforms the hill combing scheme across all workload scenarios. For light workload settings, it is not surprising that the OpenMP default scheme performs reasonably well. Under such a setting, our automatic approach gives the least improvement with a speedup of 1.5x. This still translates to 1.15 times of improvement over hill climbing. When considering medium and heavy workload settings, our approach has a clear advantage with speedups above 2.4x (up to 3.2x) over the OpenMP default scheme. This translates to a speedup over 1.36 (up to 2.3x) when comparing to the hill climbing approach. Overall, the automatic approach achieves a geometric mean speedup of 2.3x. This translates to a 1.5 times improvement over the 1.47x speedup achieved by hill climbing.

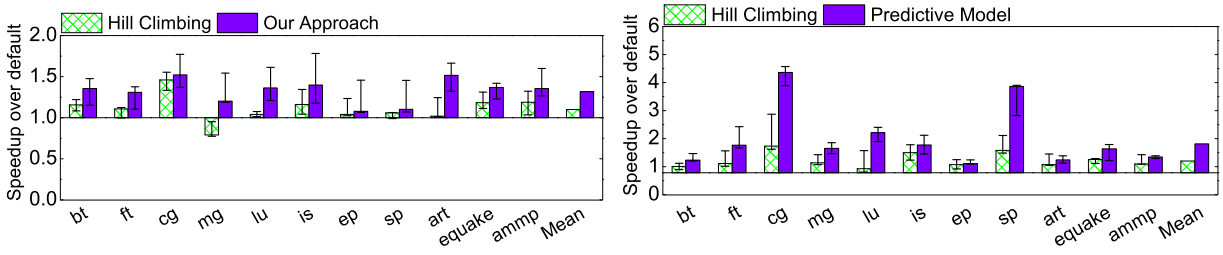
6.2 Detailed Comparison

Although our goal is to maximize the performance of the target program, it is also important to know the impact of the mapping decision on the external workload programs. Therefore we include the performance of workload programs in this section.

6.2.1 Light workload

Target Figure 5 shows the performance of the target program for low and high frequent workload arrival rates. For the *low frequent* arrival rate setting (figure 5 (a)) our approach achieves a geometric mean speedup of 1.32, which outperforms hill climbing by a factor of 1.2. For *ep* that scales very well on a multi-core, default scheme performs well and performance improvement obtained by both approaches is small. For benchmark *mg*, the hill climbing approach performs poorly by slowing down the program to 79% of the default performance. This is because *mg* consists of many short-run parallel sections which do not provide enough time allowing the hill climbing approach to find the optimal number from its starting point. For the *high frequent* arrival rate setting (figure 5 (b)), our approach achieves a geometric mean speedup of 1.82, leading to 1.5 times improvement over hill climbing. For benchmarks *cg* and *sp*, significant room for improvement was observed over OpenMP default and our approach achieves a speedup over 3.9.

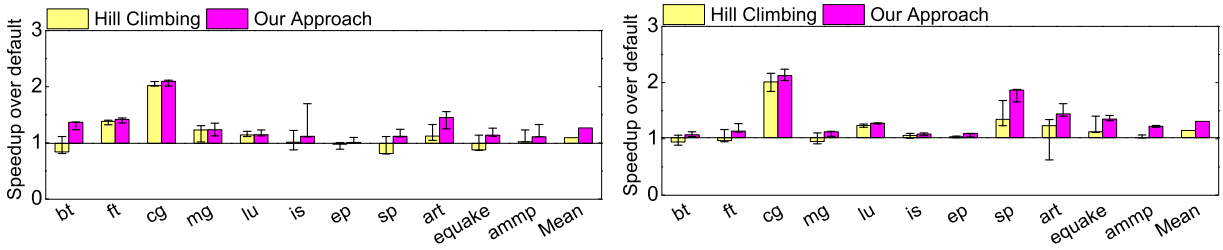
Workload Figure 6 shows performance of the workload programs when they run with the target program. where they can also benefit from the right remapping of the target program. In the *low frequent* arrival rate setting (figure 6 (a)), the hill climbing scheme and our approach achieve a mean speedup of 1.42 and 1.65 respectively. Both approaches are able to improve benchmark *cg* as well as its associated workloads with workload speedups above 2.0. For other target programs, such as *ep*, *is* and *ammp*, there is little room for improvement. In the *high frequent* arrival rate setting (figure 6 (b)), similar performance improvement (as the low frequent setting) was observed. It is important to not slow down



(a) Low frequent workload arrival rate

(b) High frequent workload arrival rate

Figure 5: Speedups of target programs with *standard deviation* for *light* workloads. Our approach achieves a mean speedup of 1.31 (vs 1.10 of hill climbing) and 1.82 (vs 1.21 of hill climbing) for low (a) and high frequent (b) arrival rates respectively.



(a) Low frequent workload arrival rate

(b) High frequent workload arrival rate

Figure 6: Speedups of associated workloads with *standard deviation* in *light* workload settings. Our approach achieves a mean speedup of 1.27 and 1.31 for low (a) and high frequent (b) arrival rates respectively.

the workload programs during this remapping. Our approach is able to do so while hill climbing gives poor performance for the workloads associated with those benchmarks by over saturating the system. In this particular setting, our approach delivers a mean speedup of 1.31 for workload programs, leading to a 114% of improvement over hill climbing.

6.2.2 Medium workload

Under a medium workload scenario, workload programs create moderate contention for resources. The performance of the default scheme can be further improved for both the target program and the associated workloads, as shown in figures 7 and 8.

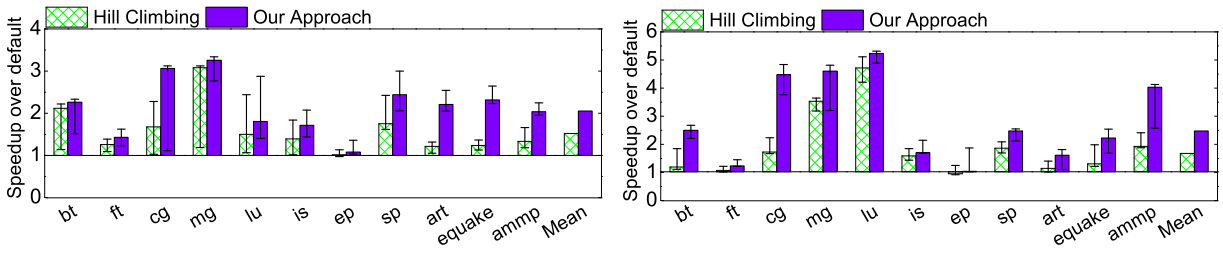
Target Significant performance improvement over the OpenMP default was observed under medium workload scenarios. The hill climbing achieves a mean speedup of 1.52 and 1.68 for the low and high frequent arrival rate settings respectively. Once again, our approach outperforms the hill climbing technique with a mean speedup of 2.05 and 2.47 for the low and high frequent arrival rate settings respectively. For benchmark *mg*, hill climbing performs better here than it does for the light workload setting, because the optimal number of threads for *mg* for this particular setting is close to the starting thread count picked by hill climbing, Hence, it can reach the optimal thread configuration quickly.

Workload As can been seen from figure 8, performance of the associated workloads can also be improved along with the target program which is more significant in a setting with a high frequent workload arrival rate (figure 8 (b)). This is due to the fact that the default scheme tends to cause harmful program contention by over subscribing hardware resources. Dynamic runtime schemes, on the other hand, achieve a better performance by eliminating the contention. Overall, hill climbing is able to improve the performance of workload programs over the default scheme with a mean speedup of 1.30 and 1.58 for the low and high frequent arrival rate settings respectively. Our approach outperforms hill climbing with a mean speedup of 1.60 and 1.92 for the low and high frequent arrival rate settings respectively.

6.2.3 Heavy workload

A heavy workload scenario depicts a realistic picture of many-core systems, data centers and more where multiple programs execute together creating extreme competition for resources. There is enormous scope for the dynamic scheme to improve over the static approach.

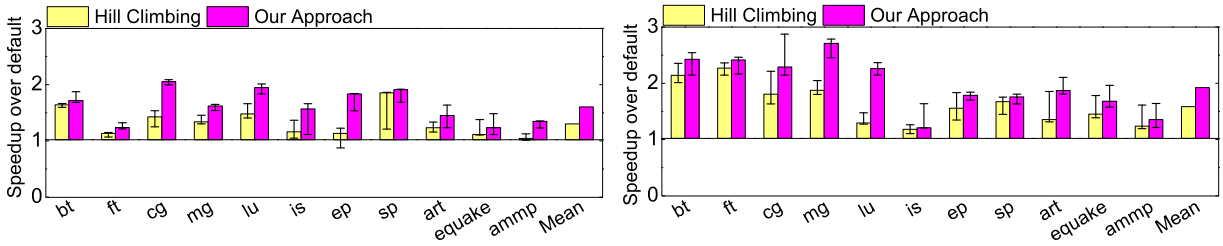
Target As observed the graphs in figures 9 (a) and (b) our approach achieves a geometric mean speedup of 2.9x and 3.2x for low and high frequent workloads respectively. For heavy workloads, the default scheme over saturates the system adding to the contention caused by the workloads. Our



(a) Low frequent workload arrival rate

(b) High frequent workload arrival rate

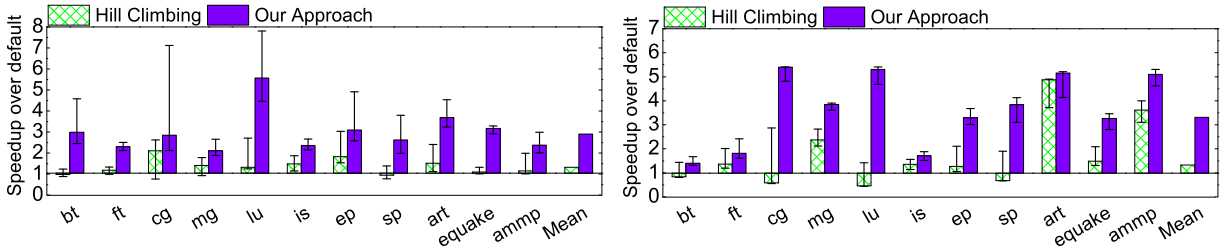
Figure 7: Speedups of target programs with *standard deviation* for *medium* workloads. Our approach achieves a mean speedup of 2.05 (vs 1.52 of hill climbing) and 2.47 (vs 1.68 of hill climbing) for low (a) and high frequent (b) arrival rates respectively.



(a) Low frequent workload arrival rate

(b) High frequent workload arrival rate

Figure 8: Speedups of associated workloads with *standard deviation* in *medium* workload settings. Our approach obtains a mean speedup of 1.65 and 1.31 for low and high frequent arrival rates respectively.



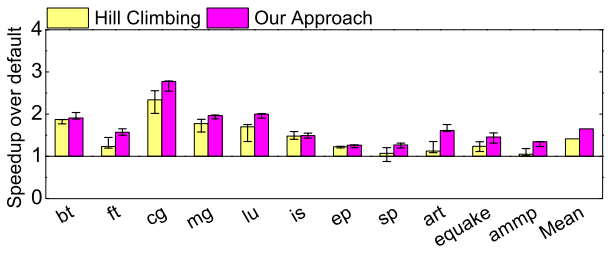
(a) Low frequent workload arrival rate

(b) High frequent workload arrival rate

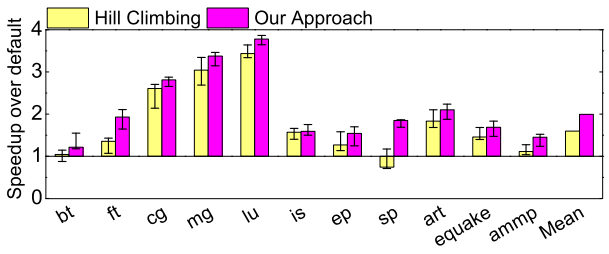
Figure 9: Speedups of target programs with *standard deviation* for *heavy* workload. Our approach achieves a mean speedup of 2.9 (vs 1.33 of hill climbing) and 3.2 (vs 1.59 of hill climbing) for low and high frequent arrival rates respectively.

approach eliminates this contention and achieves improvement as we select carefully optimal thread number suited for this heavy workload environment. The hill climbing method achieves a mean speedup of 1.33x and 1.59x which suffers mostly due to the time spent to reach optimal level. We obtain high speedup improvement for *lu*, *equake* where the hill climbing approach only manages to obtain marginal speedup. In high frequent workload settings, hill-climbing approach fails to adapt to the workloads and worsens performance of programs like *bt*, *sp*, *lu*, whereas we achieve significant improvement for these programs.

Workload We achieve an improvement in workload performance as well due to remapping strategy leading to reduced contention. From figures 10, our approach improves workload performance by 1.69x, 2.12x times in low and high frequent settings. Hill climbing approach improves workload performance by 1.46x and 1.76x times in low, high frequent workloads. This translates to 1.15x, 1.20x times improvement over hill climbing approach. Both approaches provide fair allocation of processors to workloads by which they execute faster by reducing the congestion in the system specially for programs *cg*, *lu* and *mg*.



(a) Low frequent workload arrival rate



(b) High frequent workload arrival rate

Figure 10: Speedups of associated workloads with *standard deviation* in *heavy* workload settings. Our approach achieves a mean speedup of 1.61 and 1.92 for low and high frequent arrival rates respectively.

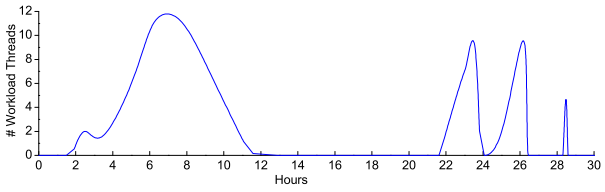


Figure 11: Workload distribution from a real world system.

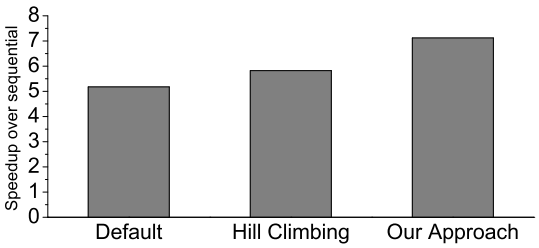


Figure 12: Performance with live system workload

6.3 Case Study Using Live System Workloads

To validate our approach in a real world setting, we selected a workload environment based on a sample of an in-house high performance cluster of computing systems [1]. The distribution of the arrival of jobs in this cluster and the number of requested processors over a period of 30 hours can be observed in figure11. We extracted jobs from a 15 minute snapshot this real-world workload from a log that recorded system activity over this period. This snapshot was selected to highlight variation in workload pattern.

Over this workload scenario, figure12 shows the speedup of one target program *lu*, with different schemes. It can be observed that our predictive model fares better than OpenMP default and state-of-the-art technique by 1.37 and 1.22 times performance improvement. This clearly shows that our model adapts well with the dynamic external workload programs in any computing environment.

6.4 Prediction Accuracy

It is important to know the prediction accuracy of our automatic approach. As it is impossible to collect the oracle performance in a live system, we evaluate the prediction accuracy of our approach using our training data where the oracle performance is known. Figure 13 compares our automatic approach against the oracle. The baseline is the default scheme. As can be seen from this diagram, the resulted performance of our model is very close to the oracle and existing gap can be enhanced by additional training benchmarks. On average, our approach achieves a speedup of 1.66, which translates to 81% of the oracle speedup of 2.06. This confirms that our model has a highly accurate prediction result.

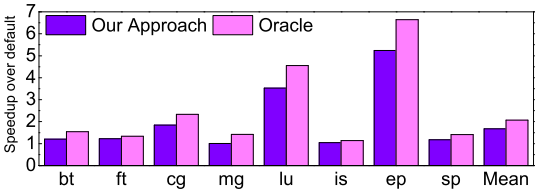


Figure 13: Speedups of our approach and the oracle on the training data. Our approach has a high prediction accuracy by achieving 81% of the oracle performance.

7. Analysis

In this section we give an insight into the source of performance improvement by our approach.

7.1 Insight into performance improvement

The performance gain from our approach can be accounted for (a) **changing directly the thread number only when needed and without delay**: In the presence of workloads, a direct change in number of threads of target program may be necessary in order to reduce the contention. Ideally an approach has to react quickly to this dynamic environment. Hill climbing changes thread numbers in incremental steps wasting time in reaching the optimal thread number. Our approach reacts quickly to the workload and changes directly to the optimal thread number as and when required.

Figure 14 shows the difference in thread numbers between successive calls to the OpenMP runtime library. The oracle stays at the same thread number 80% of the time and there are several instances where sharp change in thread number (-5, +9) is essential for optimal performance. Closely following the oracle, our approach retains the same thread number at around 70% of the time and there

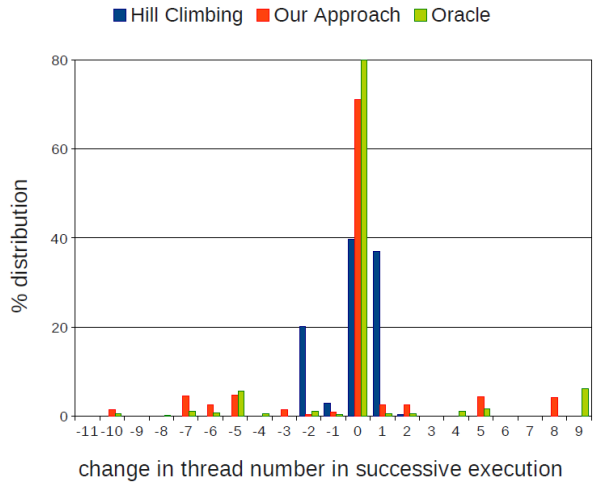


Figure 14: Distribution of thread number change in successive calls to runtime library

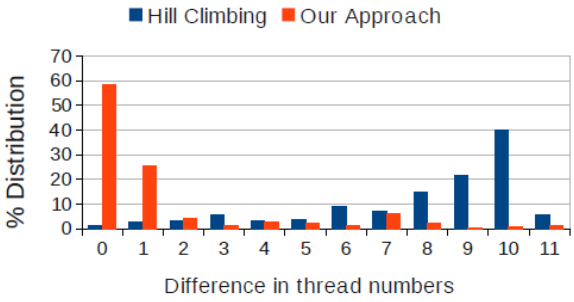


Figure 15: Distribution of difference in thread numbers between Hill Climbing and our approach relative to oracle

are significant instances where there is a direct thread number change (-7,-5,+5,+8). Hill climbing stays at the same thread number just 40% of the time and confines only to incremental changes in thread number.

(b) **Determining accurately a near optimal thread number**: An ideal approach would essentially assign thread numbers close to that of oracle. The smaller the difference, the better will be the performance. Figure 15 shows the difference in thread numbers between (i) oracle and our approach (ii) oracle and hill climbing technique. It can be observed that our model predicts thread numbers exactly as the oracle around 60% of the time, whereas the hill climbing techinque rarely assigns thread numbers the same as that of the oracle, around 1.3% of the time.

7.2 Importance of features

During the training phase, we collect all possible static code and dynamic runtime features. We then select a set of best features (table 1) based on information gain ratio. A hinton diagram (figure16) shows the importance of these features on the prediction accuracy of the model where the area of a square is a direct measure of impact created by that feature. We define *impact* of a feature f as normalized percentage value in difference in prediction accuracy between the model built using all selected features and a model built using selected features without f. Hence this impact value measures drop in prediction accuracy of the model when the specific feature is not included in the training set. It can be observed that chosen static and dynamic features are equally important for the model with load/store count and workload-threads having biggest impact among the selected features.

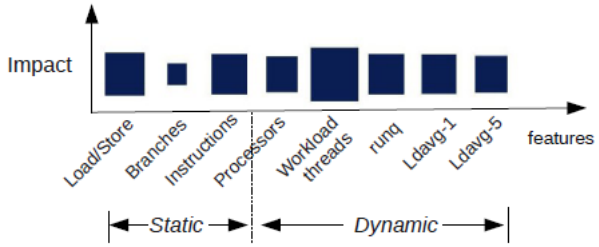


Figure 16: Hinton diagram showing impact of selected features on the model

7.3 Comparison with other learning algorithms

Here we compare our model built using different classification algorithms namely artificial neural networks (ANN), decision trees (C4.5) and support vector machines (SVM). Prediction accuracy for models built using ANN, decision tree using J48 algorithm, SVM were 81.58, 74.73, 78.65 respectively. Training time was least for decision tree model and highest for SVM. Based on the prediction accuracy values and time taken to train the model, we chose ANN to build our predictive model.

8. Related Work

There is an extensive body of research work in mapping and scheduling parallel programs.

Automatic Heuristic Generation The work presented by Grewe et al. is the most similar in spirit to our work [9]. Their model predicts the number of threads to use when launching an application. This, however, is an entirely static approach which gives very poor performance in a dynamic environment as presented in section 2. Moore and Childers use linear regression-based utility models to predict the thread counts for parallel programs [14]. They build a model for each application using profiled data. Machine learning is also shown promising in code optimization by assuming the program runs only on an unloaded machine [6, 19, 21**?** , 22] without considering the external workloads.

Dynamic Job Scheduling Previous work investigates hardware and OS based approaches to schedule tasks on SMT processors. Symbiotic job scheduling tries to find the best mix of jobs [8, 18] on SMT processors. Essentially, this is a fine grain scheduling technique that targets CPU time allocation. Throughput-Driven Fairness (TDF) scheduling policy [17] aims to maximize the system throughput while providing uniform-level of performance to the software, cre-

ating an illusion that the processor operates at a single frequency. Bhadauria and McKee proposed local search heuristics to determine runtime program schedules for energy [4] assuming programs to be executed known ahead of time.

Online Adaptation Curtis-Maury et al. build a regression model based on hardware performance counters to determine the best number of threads allocated to a single parallel programs for energy-efficiency [7]. This technique only works for a single application. Mars et al. describe a contention aware execution runtime called CAER that detects online hardware resource contention. CAER throttles down the execution of some applications on the contended resource [12]. The scheduling policies are not automatically generated and no compiler knowledge is used.

Compiler Assisted Mapping Compiler-based techniques have been applied to assist runtime scheduling [11, 20]. Flexstream [10] is a compiler and runtime framework that dynamically reconfigures a streaming program. However, it does not consider the impact on the external workloads. Parcae is a dynamic tuning framework [15, 16]. Our approach significantly improves the scheduling policy of Parcea by using compiler knowledge for runtime decision and to predict the best number of threads to use without iteratively trying different threads configurations.

9. Conclusion and Future Work

This paper has presented a predictive modeling based approach to determine the best mapping of an application in the presence of dynamic external workloads. This approach brings together static compiler knowledge of the program and dynamic runtime information to reconfigure and optimize an application in a dynamic environment. Our approach aims to maximize performance of the target program with minimum impact on the external workloads. Using predictive modeling techniques, our approach provides a significant performance improvement over existing techniques. We evaluated our approach in a live multi-program environment where workloads vary in both frequency and resource utilization. On a 12-core platform, our approach provides a significant performance improvement over the state-of-the-art online adaptation approach by a factor of 1.5x while having no impact on the external workloads. Future work will consider exploiting our techniques on heterogeneous many-core environments where processors have asymmetric computation capabilities.

References

[1] Edinburgh compute and data facilities. `http://www.ed.ac.uk/schools-departments/information-services/services/research-support/research-computing/ecdf/`.

[2] NAS parallel benchmarks 2.3, OpenMP C version. `http://phase.hpcc.jp/Omni/benchmarks/NPB/index.html`.

[3] SPECOMP Benchmark suite. `http://www.spec.org/omp/`.

[4] M. Bhadauria and S. A. McKee. An approach to resource-aware co-scheduling for cmps. In *ICS '10*, pages 189–199.

[5] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006.

[6] K. E. Coons, B. Robatmili, M. E. Taylor, B. A. Maher, D. Burger, and K. S. McKinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *PACT '08*, 2008.

[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. In *ICS '06*.

[8] S. Eyerman and L. Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. In *ASPLOS '10*, pages 91–102.

[9] D. Grewe, Z. Wang, and M. F. P. O'Boyle. A workload-aware mapping approach for data-parallel programs. In *HiPEAC '11*, pages 117–126.

[10] A. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications for heterogeneous architectures. In *PACT '09*, pages 214–223.

[11] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *MICRO 42*, pages 45–55, 2009.

[12] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention aware execution: online contention detection and response. In *CGO '10*, pages 257–265.

[13] D. Meisner and T. F. Wenisch. Dreamweaver: architectural support for deep sleep. In *ASPLOS '12*, pages 313–324.

[14] R. W. Moore and B. R. Childers. Using utility prediction models to dynamically choose program thread counts. In *ISPASS '12*, pages 135–144.

[15] A. Raman, A. Zaks, J. W. Lee, and D. I. August. Parcae: a system for flexible parallel execution. In *PLDI '12*, pages 133–144.

[16] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism orchestration using dope: the degree of parallelism executive. In *PLDI '11*, pages 26–37, 2011.

[17] K. Rangan, M. Powell, G.-Y. Wei, and D. Brooks. In *HPCA'11*, pages 3–14.

[18] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-IX*, pages 234–244, 2000.

[19] G. Tournavitis, Z. Wang, B. Franke, and M. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. In *PLDI '09*.

[20] M. Voss and R. Eigenmann. Adapt: Automated de-coupled adaptive program transformation. In *International Conference on Parallel Processing*, pages 163–170, 2000.

[21] Z. Wang and M. O'Boyle. Partitioning streaming parallelism for multi-cores: a machine learning based approach. In *PACT '10*, .

[22] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *PPoPP '09*, .