Program Affinity Performance Models for Performance and Utilization

Ryan W. Moore
Computer Science Department
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
Email: rmoore@cs.pitt.edu

Bruce R. Childers
Computer Science Department
University of Pittsburgh
Pittsburgh, Pennsylvania 15260
Email: childers@cs.pitt.edu

***Abstract*—Multithreaded applications have a wide variety of behavior, causing complex interactions with today's chip multiprocessor machines. Application threads may have large private working sets, and may compete for cache space and memory bandwidth. These threads benefit from large private caches. Other threads may share data or communicate, and thus, execute more quickly if using shared caches. Many applications fall somewhere in between, requiring careful thread-to-core assignments to maximize performance. Yet because of the large number of thread-to-core assignments on today's chip multiprocessors, it is time and energy prohibitive to exhaustively try and determine the best assignment. In this paper, we present and demonstrate application performance models that predict application performance given a proposed thread-to-core assignment. We show how these models can be quickly built and used to select thread-to-core assignments for multiple programs and to improve system utilization.**

I. Introduction

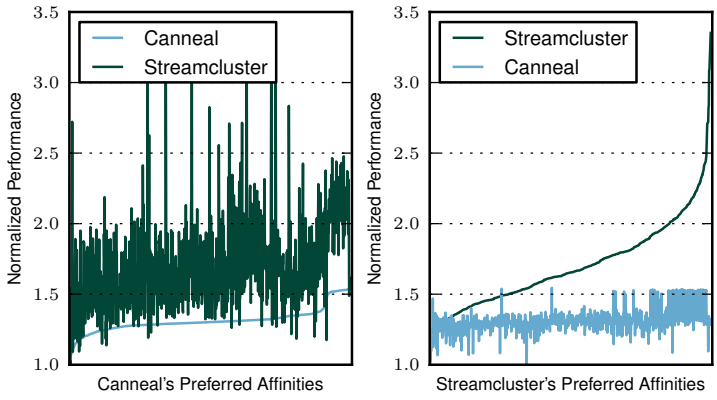
Manufacturing technology has provided huge increases in transistor density. However, additional transistors have provided little benefit to single-threaded performance due to limited instruction-level parallelism (ILP) and thermal constraints [1]. Instead, processor designers have used additional transistors to put multiple cores within one processor.

The memory and cache hierarchy of multicore systems is particularly interesting. Multiple levels of cache are shared, but only between certain cores. The distribution of DRAM across processors causes non-uniform access times. Communication links between processors may even be non-uniform.

The behavior of multithreaded applications further complicates our understanding of, and therefore prediction of, program throughput (i.e., *performance*). An application may share data between threads, and benefit from cores sharing one or more caches. Some applications, however, do not share data and therefore, benefit from being spread across caches. Many applications fall somewhere in between: Their threads may simultaneously compete for cache space while also benefiting from cache sharing. To maximize performance, thread-to-core mappings must be carefully selected.

A carefully selected thread-to-core mapping (i.e., affinity) can improve performance greatly, as seen in Figures 1a and 1b. The figure shows *canneal* and *streamcluster*'s performance across different thread-to-core mappings (affinities) [2]. The

978-3-9815370-2-4/DATE14/©2014 EDAA



(a) Canneal (b) Streamcluster

Fig. 1: Program performance across affinities (16 threads)

graphs show each program's performance when launched under a specific thread-to-core mapping (i.e., x-axis point). Programs are not executed simultaneously. *Canneal* can achieve a 50% performance gain compared to the worst affinity choice. *Streamcluster* can achieve a 240% gain! The figures also show that the best affinities for one program are not necessarily the best affinities for another.

Table I shows the number of affinities for various thread counts and available sockets for an multicore, multiprocessor machine (described in Section III). We only consider non-isomorphic affinities. With 24 application threads and four available sockets there are 1,979 affinities to select from! Furthermore, the number of available affinities greatly increases if a corunner's affinity choices are also considered. Clearly, an exhaustive evaluation of all affinities in order to find a program's best affinity is too costly.

This paper presents preliminary work in thread-to-core performance models. These models can be used to quickly evaluate thread-to-core mappings' effects on program performance. Using models, the scheduler can maximize performance and improve utilization without having to evaluate mappings online.

This paper makes the following contributions: 1) Demonstration of the importance of thread-to-core mappings, 2) Techniques to build models that predict program performance

Number of Affinities				
Thread Count	**1 Sockets**	**2 Sockets**	**3 Sockets**	**4 Sockets**
2	2	3	3	3
4	3	8	10	11
6	4	16	27	32
8	3	26	57	80
10	2	34	105	174
12	1	38	168	339
14	n/a	34	231	585
16	n/a	26	280	912
18	n/a	16	300	1,282
20	n/a	8	280	1,632
22	n/a	3	231	1,884
24	n/a	1	168	1,979

TABLE I: Non-isomorphic affinities for one application

$$P_{app,tc}(aff) = \beta_0 F_0(aff) + \beta_1 F_1(aff) + \beta_2 F_0(aff) F_1(aff) + \beta_3 {F_1}^2(aff)$$

Fig. 2: Example model

Experimental Machine Properties	
Processor	AMD Opteron 6164 HE @ 1.7 GHz
Sockets	4
L3s per socket	2
Cores per L3	6 (1 context per core)
RAM	64 GiB
OS/Compiler	Linux 2.6.39, GCC 4.6.3

TABLE II: Evaluation machine

given an affinity, and 3) An evaluation of the models and demonstration of their ability to improve performance through the proper selection of thread-to-core mappings.

II. Approach

Given the large number of possible affinities and the effect of affinity on program performance, a smart approach to performance modelling is necessary. Our approach builds models that, given a candidate affinity, predict program performance. Performance is normalized to the worst observed performance (throughput) of the program. Models are program-specific and thread-count specific. The large number of affinity choices precludes using online information (e.g., cache miss rates). Instead, we train models offline and use statically derived aspects about affinities, *features*, to make predictions (e.g., average number of threads sharing an L3).

Models are built in two steps. First, feature selection determines which features are most useful to predict performance. Second, programs are profiled under a particular thread count and performance models are constructed. We discuss these steps next, followed by a discussion of how to use the models.

A. Feature Selection

The choice of features directly affects model quality. Features need to capture the aspects about affinity that contributes to or detracts from program performance. This offline process only needs to be done once per machine. The features selected will be used in subsequent models on that machine.

Features are functions that take an affinity and return the *feature value* for that affinity. For a subset of programs, we gather performance across a subset of affinities. Regression models are built from this data with different combinations of features. Models are compared based on how well they predict the profiled performances.

Feature selection is an automated process. Features are statically derived from affinity. Therefore, we can consider new features and combinations of features without increasing profiling time. We generated a large number of features using a computer architecture expert. The feature generation process will select the features that work best (discussed in Section III).

B. Model Training

Models are trained on a subset of affinities. To ensure that models make accurate predictions in such a large space, affinities are selected such that they uniformly cover the range of features. The optimal set of these training affinities is computationally infeasible to find[1].

Instead, we randomly select sets of affinities. The set that has the highest average distance from one another is used for training. These affinities approximate a uniform distribution over the training space.

Program performance is normalized by the worst observed performance. The models predict program speedup relative to this performance. Least squares multivariate linear regression is used to find the importance of each feature and its relationship to program performance (i.e., feature coefficients). An example model is shown in Figure 2. The figure shows that this program's performance is affected by two features: F_0 and F_1. The magnitude of a coefficient β_i reflects how important a feature is to the program's performance.

C. Model Usage

After training, the model is ready for program performance prediction. The time to consult a model is nearly instantaneous, as models are simply a linear combination of products between feature values and coefficients. Feature values do have to be computed for each evaluated affinity. However, this process is fast due to the static nature of the features (e.g., number of L3s used by an affinity).

The operating system (or equivalent) can consult program models and determine which affinity is best for performance objectives. For example, the OS scheduling policy might try to improve average performance. Another policy might give higher priority to a specific program, preferring affinities that improve the program's performance to the detriment of corunners.

III. Preliminary Results

We implemented our models and feature selection on the machine described in Table II. We evaluated *blackscholes*, *bodytrack*, *canneal*, *facesim*, *streamcluster*, and *swaptions* from PARSEC [2]. We additionally experimented with *detbfs*, *knn*, and *raycast* from PBBS [3]. When applicable, the `pthreads` version of each program was used.

[1] For a affinities and p points, there would be $\binom{a}{p}$ combinations to consider.

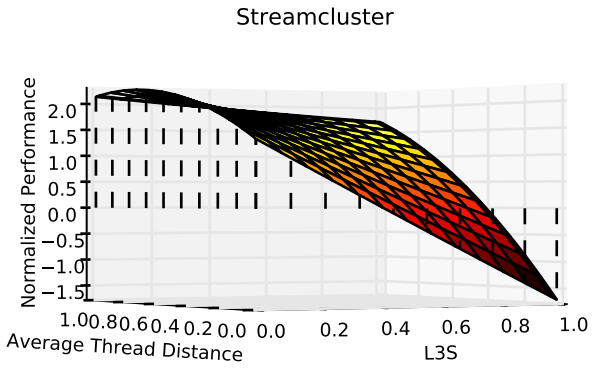


Fig. 3: Example model with two normalized features

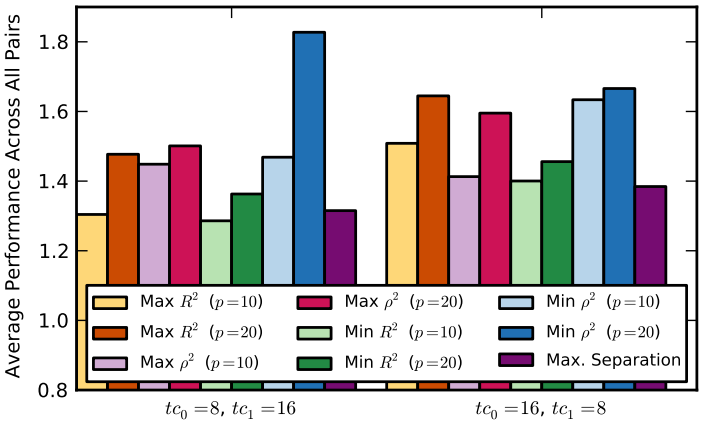


Fig. 4: Performance using models versus best static policy

For feature selection, we profiled programs run with 8 threads on a variety of affinities. After building a model for each program, we compare the predictions with the observed performance according to the R^2 and ρ^2 values. R^2 measures linear correlation. Spearman's rank correlation coefficient (ρ) is used to measure the dependence between two variables (i.e., to what extent one is a function of the other).

To aggregate model quality during feature selection, we consider the highest, smallest, and average correlation measure (R^2 or ρ^2) across models. Feature selection chooses the set of features that maximizes the aggregate measures. Features selected via each method are shown in Figure III.

Due to space limitations, we cannot fully describe each feature. Some common features include *bandwidth facilitation*^{2}, *socket imbalance*^{2}, and *L3s*. *Bandwidth facilitation*^{2} measures the average number of threads on each socket. Socket loads are squared to more heavily account for sockets with more active threads. *Socket imbalance*^{2} measures the average difference between the number of threads on each L3 of a socket. Differences are squared to place more emphasis on larger imbalances. Finally, *L3s* is the number of L3 caches whose cores have at least 1 thread assigned.

A. Model Construction

Models can be trained on as many points as desired. In our setup, the time (minutes), T, to train p training points is given by $T(p) = (1+SerialSetupTime)\times p$ where, for the evaluated programs, $SerialSetupTime$ ranges from 0 seconds (i.e., the program instantly starts its parallel work) to 2 minutes. We let the program execute its parallel section for 1 minute, observe progress, and kill the program. This process is automated. We consider models built with 10 and 20 training points. In many contexts (e.g., scientific computing), the cost of training is small compared to the benefits that the models provide over multiple uses of a program. Once the training data is gathered, the time required to build a model is minimal. Regression is fast and the number of training points is relatively small and with few dimensions (10s of points, 5 dimensions).

Figure 3 shows an example model for *streamcluster* (16 threads). We show the model with two features due to the complexity of visualizing models with additional dimensions. The figure shows *streamcluster*'s performance as the number of L3s is varied along with the average thread distance. Average thread distance is how "far" threads are away from each other. Threads sharing an L3 are closer than threads on a different L3 on the same socket, which are further away than threads on separate sockets. The model predicts that decreasing the number of L3s in use while increasing the average thread distance improves *streamcluster*'s performance. These objectives can conflict for certain affinities. Intuitively, the models predict that threads for *streamcluster* should be packed onto as few L3s as possible and that the L3s used should span sockets. Such configurations decrease communication cost between threads while allowing for increased memory bandwidth. Some points on the surface do not correspond to valid affinities (e.g., those points that correspond to a negative performance).

B. Model Evaluation

We now evaluate the use of the models for improving system-level performance objectives. In these experiments, we construct an OS policy to maximize the average program performance. Each program is weighted equally. We execute pairs of programs on two of the available four sockets of our evaluation machine, in order to force 100% utilization. For each pair of programs, we consider two cases: the alphabetically first program has 8 threads ($tc_0 = 8$) and the second program has 16 ($tc_1 = 16$), and the reverse case.

We additionally compare the models against a program-agnostic scheduling policy: *max. separation*. *Max. separation* was the best program-agnostic allocation policy. It isolates programs from one another, causing them to avoid sharing L3s and sockets.

The results are shown in Figure 4. We evaluate models built with 10 training points ($p = 10$) and 20 training points, ($p = 20$) and four sets of features (max R^2, max ρ^2, min R^2, min ρ^2), for eight comparisons in total. These four feature sets proved to be the best performing sets.

For the "max" features, the feature selection maximizes the single largest R^2 or ρ^2 value across program models. In effect, this metric chooses the features that resulted in the best single model. For the "min" feature sets, the feature selection tried to maximize the minimum R^2 or ρ^2 value across program

Maximized Objective Metric	Feature 1	Feature 2	Feature 3	Feature 4
average R^2	mean thread distance	bandwidth facilitation^2	L3s	socket imbalance^2
average ρ^2	L3s	footprint and communication	socket load	sockets
max R^2	mean thread distance	L3s	normalized average L3 MiB/thread	socket imbalance^2
max ρ^2	bandwidth facilitation^2	L3s	socket imbalance	socket σ
min R^2	mean thread distance	density^2	L3s	socket σ
min ρ^2	density^2	median L3 MiB/thread	median thread distance	max non-empty socket imbalance

TABLE III: Evaluated features chosen by various objective metrics

models. This feature selection goal chose the features that most helped the worst performing model.

Although preliminary, our models regularly outperform the best static policy. The choice of features greatly impacts the ability of the models to make accurate predictions. Additional training points result in better performance, as is to be expected: A higher budget (p) results in a more accurate model.

The best set of features to use varies depending on program thread count. This phenomenon is a result of changing cache and memory subsystem behavior as the program uses different thread counts. For some thread counts, communication speed may be a bottleneck. In others, cache space is the bottleneck. Consequently, the features used to predict performance might also change to reflect the causes of program slowdown. Still, the models worked across the thread counts.

By using the models, the evaluated policy increases performance by taking advantage of per-program performance nuances. As demonstrated in Figure 4, the models are accurate enough to capture trends in program behavior and inform allocation decisions. A program-agnostic policy, like *max. separation*, cannot consider program-specific behavior. Some pairs of programs prefer to be spread out across L3s and sockets, whereas others perform best when packed tightly together on the same sockets.

We also examined the mean-squared error of model predictions for all affinity interleavings of two programs on the evaluated system (two sockets). The models perform well, predicting program trends.

For $p = 20$, model predictions had an average MSE of 0.318. For $p = 10$, average MSE was over 2.036 due to the min ρ^2 feature set having poor accuracy for some workloads. The average MSE was 0.187 if the ρ^2 models are ignored for $p = 20$. As demonstrated, models are accurate enough to guide decisions and outperform the best static policy.

IV. Related Work

Wang et al. use a compiler-based approach to determine program affinity [4]. Our approach is not compiler-specific. Klug et al. select program affinities by evaluating over several affinities at runtime [5]. Radojkovic et al. had a try-and-evaluate solution to selecting program affinities, but for network-based applications [6]. Tam et al. use specialized hardware performance counters to determine program affinity settings [7]. Our work does not rely on specific counters.

Tang et al. use an adaptive approach to choose co-running programs' affinities [8]. Their work is fairly coarse-grained and for systems with few cores (i.e., they evaluate up to 4 cores). Their technique selects whether or not threads are distributed across LLCs and whether or not threads share the front-side bus. Our techniques are designed for higher thread counts (e.g., we evaluated up to 16) and far more thread-to-core mappings.

V. Conclusion

We presented initial work to automatically model program performance across affinities. The models rely on statically derived information, allowing for the fast examination of program performance across the large number affinities possible on modern multicore, multiprocessor machines. We also demonstrated that the models can be used to improve program performance.

Future work includes considering interference between programs, as well as models to predict the performance of unseen programs. We also plan to consider input effect on performance and program phases. We will investigate models that work across thread counts to reduce training costs.

Acknowledgment

This research was partially supported by National Science Foundation grant CCF-0811352.

References

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The case for a single-chip multiprocessor," in *Proc. of the seventh Int'l Conf on Architectural support for programming languages and operating systems*, ser. ASPLOS VII. ACM, 1996.

[2] C. Bienia *et al.*, "The parsec benchmark suite: characterization and architectural implications," in *PACT*, 2008.

[3] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *Proc. of the 24th ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '12. ACM, 2012.

[4] Z. Wang and M. F. O'Boyle, "Mapping parallelism to multi-cores: a machine learning based approach," in *Proc. of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, ser. PPoPP '09. ACM, 2009.

[5] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "autopin automated optimization of thread-to-core pinning on multicore systems," in *Transactions on High-Performance Embedded Architectures and Compilers III*, ser. Lecture Notes in Computer Science, P. Stenstrm, Ed. Springer Berlin Heidelberg, 2011, vol. 6590.

[6] P. Radojković, V. Čakarević, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero, "Thread to strand binding of parallel network applications in massive multi-threaded systems," in *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ser. PPoPP '10. ACM, 2010.

[7] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: sharing-aware scheduling on smp-cmp-smt multiprocessors," in *Proc. of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07. ACM, 2007.

[8] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. Soffa, "The impact of memory subsystem resource sharing on datacenter applications," in *Comp. Arch. (ISCA), 2011 38th Ann. Intl'l Symp. on*, 2011.