

# Pruning the Optimization Search Space Using Architecture-aware Cost Models<sup>\*</sup>

Apan Qasem Ken Kennedy

Department of Computer Science  
Rice University  
Houston, TX  
{qasem,ken}@cs.rice.edu

**Abstract.** In recent years, a number of strategies have emerged for empirically tuning applications to different architectures. Although quite successful for certain domains, empirical tuning is yet to gain wide acceptance as a viable strategy in high-performance computing. The principal bottleneck in this regard is the prohibitively large search space that needs to be explored in order to discover the best program variants for different architectures. Although there have been some efforts at using cache models in pruning the search space for kernels, the optimization search space of whole applications still remains mostly intractable. In this paper, we propose a novel search space pruning strategy. Our approach is to identify architecture-dependent parameters within compiler cost models and search for the best values of those parameters. We have implemented this strategy for exploring the search space of loop fusion and tiling for whole applications. Preliminary experiments suggest that this approach of tuning for architecture-dependent model parameters is highly effective in reducing the size of the optimization search space while incurring only a small performance penalty.

## 1 Introduction

Over the last several decades we have witnessed tremendous change in the landscape of computer architecture. New architectures have emerged at a rapid pace and at the same time, the complexity of microprocessor architecture has grown consistently. The changing nature of the processor architecture and its ever increasing complexity, has made retargeting of applications a major concern for high-performance computing. The advent of each new architecture and even a new model of a given architecture has required retargeting and retuning of applications at considerable cost. To address this issue, many strategies for automatic tuning have been proposed [9, 4, 5, 8]. Although the empirical approach has been quite successful in generating highly-tuned domain specific libraries, its application in tuning general scientific programs has been limited. The chief obstacle

---

<sup>\*</sup> This material is based on work supported by the Department of Energy under Contract Nos. 03891-001-99-4G, 74837-001-03 49, 86192-001-04 49, and/or 12783-001-05 49 from the Los Alamos National Laboratory.

in this regard is the prohibitively large search space that needs to be explored to find optimal transformation parameters. For example, Zhao et al. [13] show that the search space for fusing  $n$  statements into  $m$  loops without any reordering can be as large as  $\binom{n-1}{m-1}$ . Clearly, exploring such a large space is infeasible for a general-purpose compiler. Recent papers advocate model-guided tuning as a means of pruning this enormous search space [11, 6, 3]. In the model-guided approach, analytical models are used to restrict the search space to regions that are likely to contain mostly good values.

In this paper, we propose a new approach to pruning the optimization search space. In our strategy, we move away from the search space of parameterized transformations and instead focus on the search space of architecture-dependent parameters embedded within the cost models. As we know, the profitability of many program transformations are sensitive to certain machine parameters. For example, tile sizes are constrained by the capacity of the target cache. Compiler cost models use these architectural parameters as a means for picking the best transformation parameters. However, in most cases these parameters are difficult to determine accurately. For example, the fraction of cache we can exploit depends on the size and associativity of the cache, the number of different arrays we access in the program and also the size of each of those arrays. A static model that attempts to capture all these parameters is unlikely to be totally accurate for all architectures. The goal of our tuning strategy is to correct for these inaccuracies in the cost model. We use empirical search to find the best estimates of the machine parameters which in turn deliver the best set of transformation parameters.

Our pruning strategy reduces the size of the search space in two ways. Firstly, we can use a single parameter to capture the effects of multiple transformations which reduces search space dimensionality. For example, we can use the estimate of the cache size parameter to tune both loop fusion and tiling parameters. Secondly, for transformations that can have different parameters for different loops (i.e. tiling), we can again use just a single parameter to tune each of the loops in the program. Thus, the search space we explore does not grow with program size. For large applications with many loop nests, this property can be very effective in limiting the size of the search space.

## 2 Related Work

In recent years, there has been a flurry of work in empirical tuning that aim to improve the tuning process using machine learning, statistical methods and heuristic search strategies. However, relatively few of these have addressed the issue of using compiler models in pruning the optimization search space.

Knijnenburg et al. [6] introduced the notion of search space pruning using compiler models in the context of iterative compilation. In their work, they examine the effects of cache models on empirically tuning tiling and unroll factors. They use static models in combination with a cache simulator to filter out bad candidates with high cache miss rates from the parameter search space. Their

results show that the use of cache models can indeed speedup the tuning process significantly without a high sacrifice in performance. An interesting and important aspect of this work is the use of *slack factors* to estimate the capacity of set-associative caches. These *slack factors* are determined experimentally and then used as a fixed value during the tuning process. In our work, it is these *slack factors* that form the basis of our search space. As we show in Section 3, having the *slack factors* integrated into the tuning process can significantly reduce the optimization search space.

Yotov et al. [11] show that analytic modeling alone can deliver performance that is comparable to that of ATLAS. In subsequent work, they have shown that modeling, combined with local search and model refinement is highly effective in generating optimized code for BLAS on different architectures [12]. Chen et al. [3] combines analytical models with empirical search to automatically tune dense matrix computations to two different architectures. They use static models to generate a parameter search space that is likely to contain the optimal parameter value. By combining their cache-conscious models with empirical search, they are able to achieve performance comparable to that of ATLAS on the matrix multiply kernel. The search process is about 2-4 times faster than that of ATLAS. Most recently, Agakov et al. [1] have used predictive modeling techniques to focus search strategies to more profitable regions within the search space. Their approach was highly effective in reducing the tuning time for a large search space of  $82^{20}$  points.

Our approach to search space pruning is distinct from previous work mentioned in this section, in that we aim to tune architectural parameters integrated in our cost model rather than the space of transformation parameters.

### 3 Approach

Our approach to defining and pruning the optimization search space is best described as a three-step process. In this section, we describe each step in some detail.

***Step 1: Identify architectural resources that affect profitability.***

Most transformations - particularly those worth tuning for - are considered to be architecture sensitive; that is, their profitability depends on certain parameters of the target architecture. In most cases, the architectural parameters will impose constraints on the transformation parameters. For example, the profitability of unroll-and-jam is constrained by the register pressure of the unrolled loop [2]. The literature on code-improvement transformations is replete with many such examples. The first step in our tuning process is to identify key architectural resources that affect the profitability of the transformations in question and then determine the relationship between the architectural parameters and the transformation parameters.

***Step 2: Construct parameterized models to estimate available resources.***

Once the architectural resources have been identified, we need a mechanism to estimate the amount of resource that is available to the program. The amount of resource that can be exploited by a program is determined by a host of factors. For example, the fraction of cache we can exploit depends on the size and associativity of the cache, the number of different arrays we access in the program and also the size of each of those arrays. Hence, the second step in our tuning process is to construct models that estimate the amount of resource that is available. For each resource  $R$ , we construct a function that computes the *effective size* of  $R$ .

$$R' = \text{EffectiveSize}(r_1, r_2, \dots, r_n) \text{ s.t. } R' \leq R$$

where  $r_1, r_2 \dots r_n$  are parameters that determine the effective size of  $R$ .

Again, there is published work describing many such models. However, the key issue in using these models is finding a suitable parameterization, so that we can expose the relevant parameters for tuning through empirical search. We accomplish this by introducing the notion of a *tolerance term*. We derive tolerance terms for each of the machine parameter estimates such that there is a linear relationship between the tolerance term and the architectural resource. For example, we use the cache miss rate of the program as tolerance for estimating the effective cache capacity.

$$L'_k = \text{EffectiveCacheSize}(s_k, a_k, T)$$

where  $L'_k$  is the effective size of the cache at level  $k$ ,  $s_k$  and  $a_k$  refer to the size and associativity of the cache and  $T$  is a tolerance term that corresponds to the miss rate at  $L_k$ .

***Step 3: Search for best estimates using tolerance values.***

Our search space is the Cartesian product of the sets of tolerance values used in estimating each architectural parameter. As such, any multidimensional search strategy used to explore the search space of transformation parameters can be effective in exploring the search space. However, a natural choice for exploring this search space turns out to be a sequential search. For each tuning parameter in the search space, we start off conservatively with a low tolerance value and increase the tolerance at each subsequent iteration. We stop the iterative process either when performance degrades or when we have reached the availability threshold of a particular resource. The rationale behind choosing a sequential search is the following: since at each step we allow the program to consume more of a particular resource, at some iteration we will reach a threshold value where the program will have consumed too much of that resource. From that point on consuming more of that particular resource will only further degrade performance. As we shall see in the experimental results in Section 5, this simple search strategy works fairly well for tuning loop fusion and tiling parameters.

## 4 An Example

In this section, we demonstrate the effectiveness of our strategy using a simple example. For this example, we only consider fusion of innermost loops and tuning

of the effective register set parameter. We first explain the fusion parameter search space, then the search space for the effective register set and then present results from an experiment comparing the two search spaces.

If reordering of loops is not allowed, the number of different ways to fuse  $k$  loops is  $2^{k-1}$ . Thus, the number of points in the fusion search space of  $k$  loops is  $2^{k-1}$ . We can represent the search space of different fusion configurations using Gray Code ordering. In a Gray Code ordering a fusion configuration is represented using a bit pattern where each bit corresponds to an edge between two fusible loops. A bit is set if the corresponding adjacent loops are fused. For example, if we have eight fusible loops then we need bit strings of length seven where bit string 0000000 corresponds to no loops being fused and 1111111 corresponds to all loops being fused.

A key consideration for fusing loops at the innermost level is the register pressure of the fused loop. If the number of registers required to execute the fused loop is more than the number of available registers then fusion is unlikely to be profitable because of register spills. For this reason, a compiler cost model for fusion will usually impose the following constraint for fusing loops:

$$\text{Register Pressure}(\text{Loop}_{\text{fused}}) < \text{Effective Register Set}$$

where *Register Pressure* is the number of registers required to execute the fused loop and *Effective Register Set* is the number of physical registers available to the loop at runtime.

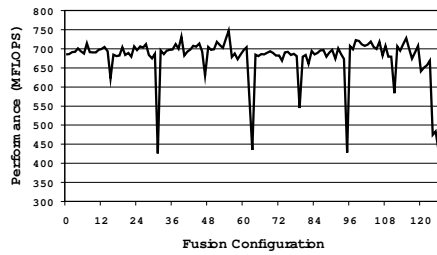
Although there are several algorithms that can estimate the register pressure of a loop nest, the actual number of physical registers available at runtime is usually much more difficult to determine accurately. Clearly, the above constraint in the cost model will be ineffective if we do not have an accurate estimate of the effective register set. Conversely, the constraint will produce the best results when we have the best estimate for the effective register set. Hence, in our empirical tuning framework, we search for the best estimate of the effective register set with the assumption that the best estimate will generate the best fusion configuration.

The number of available registers at runtime is a subset of the actual number of physical registers in the program. Hence, we estimate the effective register set using the following formula:

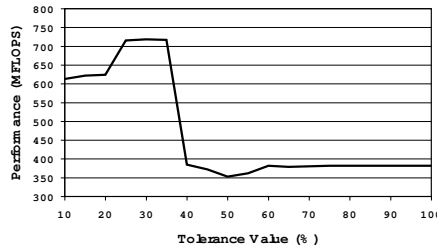
$$\text{Effective Register Set} = \lceil T \times \text{Register Set Size} \rceil, \quad 0 < T \leq 1$$

Here,  $T$  is the *tolerance term* that determines the fraction of the physical register set that is available to us at runtime. Thus, at each step in our search process we generate and evaluate a fusion configuration where the register pressure of each fused loop is less than the effective register set for some value of  $T$ . For example, if  $T = 0.5$  and the number of physical registers is 64 then we will generate a fusion configuration where the register pressure of each fused loop is

less than 32.<sup>1</sup> Thus, the search space for tuning the register set parameter is a set of tolerance values. The number of points in the search space is determined by how finely we wish to tune the parameter. For example, if we increase our tolerance by 0.05 at each step then we will have just 20 points in the search space. Note, that if increasing our tolerance does not result in a larger effective register set or a different fusion configuration then that point in the search space does not need to be evaluated. Thus, the number of points in the search space is bounded above by the size of the register set of the target platform and in practice, the number of points that need to be evaluated is likely to be much smaller than this upper bound.



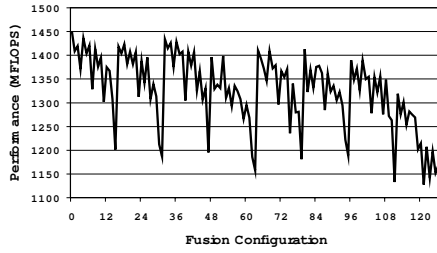
**Fig. 1.** Performance curve for fusion configuration search space on Opteron



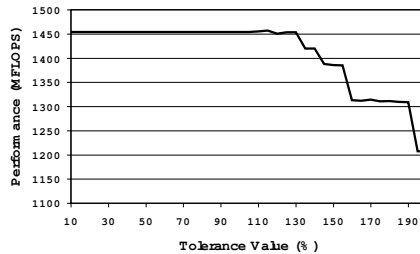
**Fig. 2.** Performance curve for effective register set search space on Opteron

To compare the two different search spaces we perform a simple experiment with the `advect3d` kernel from the NCOMMAS [10] weather-modeling application. The `advect3d` kernel has a total of 24 loops divided into eight loop

<sup>1</sup> Note, for any given tolerance term there can be multiple fusion configurations. Our static cost model determines which of those configurations will be picked for evaluation.



**Fig. 3.** Performance curve for fusion configuration search space on Pentium 4



**Fig. 4.** Performance curve for effective register set search space on Pentium 4

nests which are perfectly nested. All loop nests are fully fusible. For this experiment, we consider fusing only the innermost loops without any reordering. As explained previously, the fusion search space for `advect3d` contains  $2^{8-1} = 128$  points. The size of the search space of the register set parameter is dependent on the tolerance value increments and the number of physical registers in the target platform. We present performance results for these two search spaces on two platforms: a 2 GHz Opteron with 32 floating-point registers and a 2.4 GHz Pentium 4 with 8 floating-point registers. For both platforms, we increase tolerance by 5% at each step. Hence, for both platforms, the register set search space contains 20 points. However, since the number of registers on Pentium 4 is less than 20, the number of points that result in different fusion configurations is bounded above by the number of physical registers.

The performance of all possible fusion configurations on the Opteron is shown in Fig. 1. As expected, the performance line is very jagged with many peaks and valleys. The performance curve for the effective register set search space on the same platform is shown in Fig. 2. As explained earlier, this search space is much smaller than the search space of fusion configurations. However, the important thing to note here is that the performance line for this search space is relatively *smooth*. Not only that, the performance line follows a specific pattern. Initially, when we increase the tolerance from very low values (i.e. 10%) performance keeps

increasing. Then, when  $T = 35\%$ , there is a big drop in performance. According to our search heuristic,  $T = 35\%$  represents the threshold point and no further exploration of the search space is necessary. Indeed, we observe that none of the points beyond this threshold produce better performance. Hence, we could stop our search after evaluating just seven points in this search space. Another issue to note, is the leveling-off of the tail-end of the performance curve. This happens because all eight loops in `advect3d` are fused at the 55% tolerance level and the fusion configuration does not change for any value of  $T$  beyond that point. Hence, even if we were doing an exhaustive search we would not need to evaluate this portion of the search space.

The performance curves for `advect3d` on Pentium 4 are presented in Figs. 3 and 4. We notice very similar results on this platform as well. A jagged performance line for the fusion configuration search space and a smooth line for the search space of the effective register set parameter. Since Pentium 4 has so few floating-point registers, only a single pair of loops is fused when we increase our tolerance to a 100%. This explains the long flat segment at the beginning of the performance line in Fig. 4. Our search heuristic does not evaluate points beyond the 100% threshold. Hence, the search on this platform stops at  $T = 100\%$  after fusing just one pair of loops. To verify that this conservative approach is indeed the right one, on this platform, we forced the search strategy to evaluate points beyond  $T = 100\%$ . As the results in Fig. 4 show, going beyond the maximum threshold and trying to fuse more loops makes the performance worse. Thus, for this platform it is best to stop at  $T = 100\%$ .

## 5 Experimental Results

We have implemented our search space pruning strategy for two transformations: loop fusion and tiling [7]. Our search space consists of tolerance values of two architecture-sensitive model parameters: *Effective Register Set* and *Effective Cache Capacity*. Our testing platform is a MIPS Origin r12000. We select four programs that exhibit opportunities for loop fusion and tiling: `advect3d`, an advection kernel for weather modeling, `erle`, a differential equation solver, `liv18`, a hydrodynamics kernel from Livermore loops, and `mgrid`, a multi-grid solver from SPEC 2000. For each program we run two sets of experiments: one using a sequential search on the pruned search space (`model-based`) and another using a multi-dimensional direct search strategy on the *un-pruned* search space (`direct`).

Performance results from four applications on the MIPS are presented in Fig. 5. The results show that `model-based` is able to find values that are very close to the values found by `direct`. The performance gap is never more than 5%. On the other hand, in terms of tuning time we pay a high premium when we apply direct search. Fig. 6 shows that on average `direct` requires about four times as many program evaluations as `model-based`. In the context of empirical tuning, where number of program evaluations is the principal bottleneck, this savings in tuning time will be significant for any decent-sized application. In such

cases, the savings in tuning cost will make the small sacrifice in performance worthwhile.

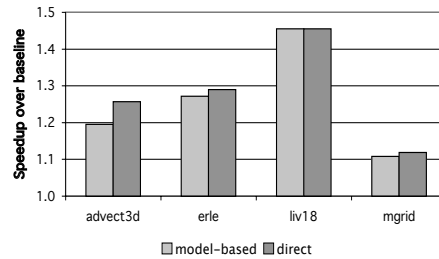


Fig. 5. Performance comparison: model-based vs direct

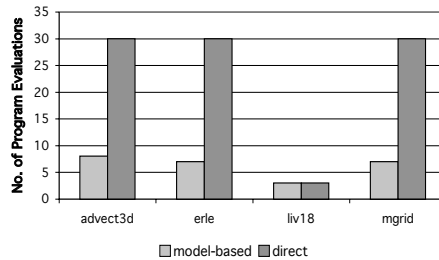


Fig. 6. Tuning time comparison: model-based vs direct

## 6 Conclusions

In this paper, we have presented a method of pruning the optimization search space by searching for architecture-dependent model parameters. Preliminary experimental results suggest that this approach can be highly effective in reducing the size of the search space while incurring only a small performance penalty.

**Acknowledgement.** We would like to thank the anonymous reviewers for their constructive comments and helpful suggestions in improving the quality of this paper.

## References

1. F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. O'Boyle, J. Thomson, M. Toussaint, and C. Williams. Using machine learning to focus iterative optimization. In *International Symposium on Code Generation and Optimization, 2006. (CGO 2006)*., New York, NY, 2006.
2. S. Carr. *Memory-Hierarchy Management*. PhD thesis, Dept. of Computer Science, Rice University, Sept. 1992.
3. C. Chen, J. Chame, and M. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization*, San Jose, CA, 2005.
4. K. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21<sup>st</sup> century. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, Oct. 2001.
5. G.G.Fursin, M.F.P.O'Boyle, and P.M.W.Knijnenburg. Evaluating iterative compilation. In *Proceedings of the Fifteenth International Workshop on Languages and Compilers for Parallel Computing*, College Park, Maryland, July 2002.
6. P. Knijnenburg, T. Kisuki, K. Gallivan, and M.F.P.O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurrency and Computation: Practice and Experience*, 16:247–270, 2004.
7. A. Qasem and K. Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *Proceedings of the 20th ACM International Conference on Supercomputing*, June 2006.
8. S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. August. Compiler optimization-space exploration. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, San Francisco, CA, 2003.
9. C. Whaley and J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of SC'98: High Performance Networking and Computing*, Orlando, FL, Nov. 1998.
10. L. J. Wicker. NSSL collaborative model for atmospheric simulation (NCOMMAS). <http://www.nssl.noaa.gov/~wicker/commas.html>.
11. K. Yotov, X. Li, G. Ren, M. Cibulskis, G. DeJong, M. Garzaran, D. Padua, K. Pingali, P. Stodghill, and P. Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the SIGPLAN '03 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.
12. K. Yotov, K. Pingali, and P. Stodghill. Think globally, search locally. In *Proceedings of the 19th annual international conference on Supercomputing (ICS06)*, 2005.
13. Y. Zhao, Q. Yi, K. Kennedy, D. Quinlan, and R. Vuduc. Parameterizing loop fusion for automated empirical tuning. Technical report, Lawrence Livermore National Laboratory, Dec. 2005.