A Systematic Approach to Model-Guided Empirical Search for Memory Hierarchy Optimization*

Chun Chen, Jacqueline Chame, Mary Hall, and Kristina Lerman

University of Southern California/Information Sciences Institute
4676 Admiralty Way, Suite 1001, Marina del Rey, CA 90292
{chunchen,jchame,mhall,lerman}@isi.edu

Abstract. The goal of this work is a systematic approach to compiler optimization for simultaneously optimizing across multiple levels of the memory hierarchy. Our approach combines compiler models and heuristics with guided empirical search to take advantage of their complementary strengths. The models and heuristics limit the search to a small number of candidate implementations, and the empirical results provide accurate feedback information to the compiler. In previous work, we propose a compiler algorithm for deriving a set of parameterized solutions, followed by a model-guided empirical search to determine the best integer parameter values and select the best overall solution. This paper focuses on formalizing the process of deriving parameter values, which is a *multi-variable* optimization problem, and considers the role of AI search techniques in deriving a systematic framework for the search.

1 Introduction

Since the development of the earliest optimizing compilers, it has been well understood that compiler optimization is a challenging problem with a variety of tradeoffs. As architectures and applications become increasingly complex, statically predicting the impact of individual compiler optimizations and the aggregate impact of a collection of optimizations is becoming increasingly difficult.

Currently, optimization of high-end computing applications is done manually in an ad-hoc manner. A recent strategy to address this complexity and improve performance employs *empirical optimization*, to systematically evaluate a collection of automatically-generated *code variants* and *parameter values* [8, 4]. Code variants, in this context, are alternative but equivalent implementations of the same computation. For a particular variant, there may additionally be optimization parameters such as unroll factors and tile sizes. Rather than estimating performance through analysis, implementation variants are actually *executed on the target architecture* with representative input data sets across different parameter values so that performance can be measured and compared. However a recent paper [9] showed that the model-driven approach on Matrix Multiply can yield comparable performance with ATLAS [8], suggesting that the compiler-derived model may be able to limit the search space.

In a previous paper, we demonstrated that combining the strengths of models with empirical search can yield better performance than either ATLAS or hand-coded BLAS [2]. For memory hierarchy optimization, finding a set of variants and parameters that result in high

* This work has been supported by NSF grant ACI-0204040.

```
DO K = 1,N
  DO J = 1,N
    DO I = 1,N
      C[I,J] += A[I,K]*B[K,J]
```

(a) Original Matrix Multiply

```
new P[TK,TJ]
new Q[TI,TK]
DO KK = 1,N,TK
  DO JJ = 1,N,TJ
    copy B[KK..KK+TK-1,JJ..JJ+TJ-1] to P
    DO II = 1,N,TI
      copy A[II..II+TI-1,KK..KK+TK-1] to Q
      DO J = JJ,min(JJ+TJ-1,N),UJ
        DO I = II,min(II+TI-1,N),UI
          load C[I..I+UI-1,J..J+UJ-1] into registers
          DO K = KK,min(KK+TK-1,N)
            prefetch P's
            multiply Q's and P's to registers
          store C[I..I+UI-1,J..J+UJ-1]
```

(b) Optimized Matrix Multiply

Fig. 1. Matrix Multiply

performance is difficult because of the complex tradeoffs among memory hierarchy levels. In addition, the search space is difficult to model analytically since performance can vary dramatically with problem size and optimization parameters. Empirical results can help the compiler tune the accuracy of its models and select the best candidate implementations. A purely empirical approach is not practical in general because the search space of possible variants and their parameters is prohibitively large. A compiler's understanding of the impact of code transformations on performance can be used to limit the search space and rule out the vast majority of inferior implementations.

This paper explores the parameter search in an effort to develop a systematic and generalizable approach that goes beyond our memory hierarchy optimization strategy. Realizing that many compiler optimizations require some sort of heuristic-based search, we consider the suitability of AI search techniques. Although compiler researchers have begun to apply elements of AI to their work [3, 6, 7], no principled methodology yet exists. We believe a formal framework will enable compiler developers and application programmers to move away from ad-hoc approaches toward a principled process of design.

The remainder of the paper is organized as follows. Section 2 illustrates the problem of optimizing for multiple levels of the memory hierarchy and describes our framework. Section 3 formalizes the search problem as a multi-variable optimization problem. Finally, Section 4 concludes the paper.

2 Guided Empirical Search for the Memory Hierarchy

To achieve high application performance on today's architectures with deep memory hierarchies, it is essential to consider the overall performance impact of individual optimizations. In [2] we propose an approach for simultaneously optimizing across all levels of the memory hierarchy using a combination of compiler analysis, architecture models and a guided empirical search for optimization parameters.

Before describing our approach, we illustrate the tradeoffs among memory hierarchy optimizations using matrix multiply as an example. Figures 1 (a) and (b) show the original matrix multiply and a parameterized code variant derived by applying several optimizations to the original version (loop interchanging, unrolling, tiling, data copying and data prefetching). Table 1 shows the performance of the optimized variant for four sets of optimization parameters.

$Set = \{T_I, T_J, T_K, P_{P,K}\}$	Loads	L1 misses	L2 misses	TLB misses	Cycles
$set1 = \{1, 32, 64, 0\}$	4.20 B	142 M	21.6 M	0.231 M	10.2 B
$set2 = \{8, 256, 256, 0\}$	4.08 B	319 M	7.19 M	4.42 M	9.70 B
$set3 = \{16, 512, 128, 0\}$	4.11 B	182 M	8.01 M	2.78 M	9.47 B
$set4 = \{16, 512, 128, 4\}$	5.12 B	188 M	8.04 M	2.78 M	9.18 B

Table 1. Performance variation with optimization parameter values

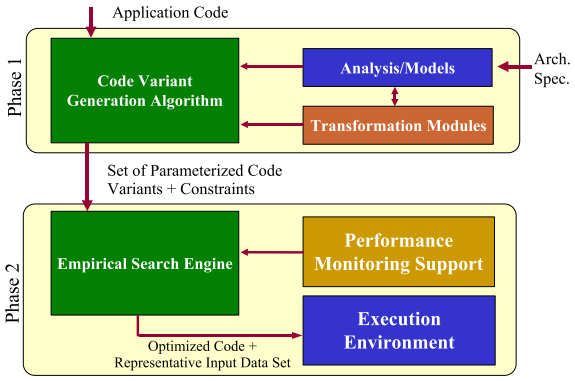

Fig. 2. Optimization framework

Each row in Table 1 corresponds to a set of integer parameter values $\{T_I, T_J, T_K, P_{P,K}\}$[1] where T_I, T_J, T_K are the tile sizes of loops `I`, `J` and `K` and $P_{P,K}$ is the prefetch distance of array `P` in loop `K`. For each set of parameters, the optimized code variant was executed on an SGI Octane R10000 and the performance data was obtained using the performance monitoring interface PAPI [1]. The same matrix size, larger than the second-level cache, was used in all experiments.

The first parameter set achieves the lowest number of L1 misses by tiling loops `J` and `K` with sizes 32 and 64 ($T_I = 1$ indicates that loop `I` is not tiled, and $P_{P,K} = 0$ indicates that no prefetches are inserted for array `P`). The tile sizes in $set2$ result in lower L2 misses but higher L1 misses than those of $set1$. $set4$ achieves the best performance by balancing locality between the L1 and L2 caches, even though it has the highest number of loads and none of the best L1 or L2 misses. In addition, prefetching with distance `4` achieves an extra 3% reduction in total cycles with respect to $set3$.

This example illustrates that achieving the best performance requires exploiting reuse in all levels of the memory hierarchy, trading off best performance at any particular level for locality at all levels.

The remainder of this section presents a summary of our framework, which is organized into two main phases (Figure 2). In the first phase the compiler generates a set of parameterized code variants. The second phase is a search among parameter values for each code variant, guided by models and heuristics.

[1] For simplicity, the unroll factors of this optimized variant are the same for all sets of parameters and are not shown in the table.

Transformations	Definition	Goal	Variants	Parameters
Loop permutation	Change the loop order	Enable U&J and Tiling + Reduce TLB misses	Different loop orders	-
Unroll and Jam	Unroll outer loops and fuse inner loops	Reuse in registers	-	Unroll factors
Scalar replacement	Replace array accesses with scalar variables			
Tiling	Divide iteration space into tiles	Reuse in cache	-	Tile sizes
Data copying (w/tiling)	Copy subarray into contiguous memory space	Avoid conflict misses + Avoid TLB thrashing	Yes/no on specific data structures	-
Prefetching	Prefetch data into cache before actual references	Hide memory latency	-	Prefetch distances

Table 2. Transformation variants and parameters

Phase 1: Generate Parameterized Variants using Models. The code generation algorithm systematically applies individual transformations based on analysis and models (the details of the algorithm can be found in [2]). The compiler uses dependence analysis to determine the legality of code transformations, locality analysis to evaluate data reuse and select specific locality optimizations, register reuse analysis to estimate register pressure, etc. The models include register, cache and TLB models and also incorporate various heuristics for those optimizations. Along with each code variant, the compiler generates a set of constraints for the optimization parameters, which are used in the second phase to guide and prune the search. Table 2 shows the code transformations and parameters used by the algorithm. The fourth column indicates whether a transformation results in more than one code variant. For example, for loop permutation the algorithm may generate multiple code variants, each with a different loop order, if it cannot decide which order is best statically. Other transformations, such as loop tiling, do not increase the number of variants, but result in code variants with unbound parameters, as illustrated in the table's last column.

Phase 2: Search for Parameter Values. In this phase, a guided empirical search performs a series of experiments to derive the best parameter values for each code variant. In addition, code transformations that depend on parameter values are applied during this phase. The resulting code variants are then compiled and executed on the target machine. The search engine uses metrics collected by performance monitoring tools to evaluate the quality of a code variant with a given set of parameter values. Currently we use PAPI to collect performance data, and use processor cycles as the performance evaluation function for the search.

In [2] we use compiler domain knowledge about specific optimizations to search the parameter space efficiently. In the next section we discuss how to approach the search for parameter values systematically and explore this problem in a broader context.

3 Systematically Searching the Parameter Space

The goal of this section is to provide insight into a systematic solution to Phase 2, searching for integer parameter values of code variants to select the best variant and parameter set. Before discussing search techniques, we describe aspects of the search that can be captured

by search algorithms to expedite the search and lead to high-quality solutions. We use the memory hierarchy optimization problem to make the discussion more concrete. The search for a set of parameter values leading to the best performance can be expressed as a function of several features, which are specified by the compiler:

$$Search = \{Parameters,\ Constraints,\ Dependence,\ Ordering,\ Starting\ Points\}$$

Set of parameters. In the case of memory hierarchy optimization, let us assume we are optimizing a single n-deep loop nest.[2] Then the following set of parameters is associated with each variant:

- $U_{L1}, \ldots, U_{Ln}$: *unroll factors* for each loop in an n-deep loop nest.
- $T_{L1}, \ldots, T_{Ln}$: *tile sizes* for each loop in an n-deep loop nest.
- $P_{A1,L1}, \ldots, P_{A1,Ln}, \ldots, P_{Am,Ln}$: *prefetch distances* for arrays $A1$ through Am within the loop nest.

Set of constraints on integer values. Phase 1 provides a set of constraints for each unbound parameter of a code variant.[3] For example, when unroll-and-jam is applied to multiple loops the unroll factors should be such that reuse is maximized while satisfying the register capacity constraints. In general, a constraint on unroll factors due to register capacity can be expressed as $a_1 * U_1 + a_2 * U_2 + \ldots + a_n * U_n \leq R$, where a_i is a constant that depends on the loop nest and R is register file size. Similarly, tile sizes should be such that the tile footprint fits in cache, and a constraint on tile sizes can be expressed as an inequality involving the product of the tile sizes of each loop. These constraints prune off uninteresting portions of the search space, and keep the search focused on the area of the search space most likely to achieve the best results.

Dependence between parameters. Parameters that appear on a same constraint are considered interdependent and are evaluated as a set. Unroll factors of multiple loops may appear in a same constraint due to register capacity, and are considered interdependent. Similarly, tile sizes of multiple loops may appear in one or more constraints related to cache capacity. Unroll factors and tile sizes are considered independent from each other, based on the knowledge that reuse in registers and caches are complementary as long as the unroll factor of each loop in the original loop nest does not exceed the tile size of that same loop. In practice tile sizes are typically much larger than unroll factors due to the difference between cache and register file capacities found in most machines.

Ordering of parameter selection. In general, optimization parameters may be interrelated, and the order in which they are evaluated may impact the search results. Compiler domain knowledge can be use to determine a search ordering for parameters that are considered independent. In memory hierarchy optimization the benefits from unroll-and-jam and scalar replacement are typically much higher than those of tiling and copying: reuse in registers reduces the *number* of memory operations, while reuse in cache reduces only the *latency* seen by the processor. Similarly, tiling reduces number of accesses to memory, while prefetching hides the memory latency. Therefore, the search for unroll factors precedes the search for tile sizes, which in turn precedes selecting prefetch distances. Since prefetching

[2] Without loss of generality, if the code has multiple loop nests, each nest will have such a set of parameters associated with them. For simplicity we treat them independently in this discussion.

[3] Parameters that are set to their default values at Phase 1 indicate that an optimization should not be performed. The default values for unroll factors, tile sizes and prefetch distances are 1, 1, and 0, respectively.

may displace data from the cache, tiling parameters may need to be adjusted after prefetch parameters are determined.

Starting points, stop criteria: Compiler models can suggest starting points for the search based on domain knowledge, and provide stopping criteria by estimating bounds for the performance of the optimized code variants.

3.1 A Systematic Search Space

Given the previous discussion, a systematic approach could search for parameter values using the specified ordering of parameters, and within the specified constrained range. Although much of the search space has been pruned away, there still remains a fairly large number of points to search.

In the following we discuss how to incorporate domain knowledge in a systematic search for parameter values, using the parameter space of the code variant shown in Figure 1(b) as an example.[4]

Figure 3 shows a tree representation of the parameter space of the code variant in Figure 1(b). The parameters of this code variant are the unroll factors U_I, U_J and U_K, the tile sizes T_I, T_J and T_K, and the prefetch distance of array P in loop K, $P_{P,K}$. The evaluation function is measured execution time. Each tree level, except the root, corresponds to a set of interdependent parameters. In this example the second level corresponds to unroll factors, the third level to tile sizes and the fourth level to the prefetch distance $P_{P,K}$. On a given level, each node corresponds to a set of integer values for the parameters associated with that level. For example, each node at the second level corresponds to a set $\{U_I, U_J, U_K\}$ where $1 \leq U_I, U_J, UK \leq R$, and R is the number of registers available. Hence each node is a partial set of parameters for the code variant. Selecting a set of parameters corresponds to finding a path, from root to a leaf node, such that the performance of the variant with the complete set of parameter values in this path is maximized.

This tree representation incorporates some of the compiler's domain knowledge discussed in the previous section.

Dependence: In Figure 3 each node in the second level represents a set of interdependent unroll factors and each node in the third level represents a set of tile sizes. Unroll factors and tile sizes are considered independent from each other and are represented as different levels of the tree.

Ordering: The search tree has three levels, with parameters that have greatest impact on performance at the highest levels. Thus the compiler's domain knowledge about the effect of optimizations on performance is captured by the order implied by the levels. Therefore the search for unroll factors is performed before selecting tile sizes, which is performed before selecting prefetch distances. If prefetching is found to be profitable the tree representation allows the search to backtrack to a previous solution. For example, the search can explore solutions with a larger tile size for the loop in which prefetches are inserted, to increase the amount of latency that can be covered so that prefetches are effective.

Pruning the parameter space: Constraints derived at Phase1 are used to prune the search. In Figure 3 all second-level nodes $< U_I = R, U_J \geq 2, \ldots >$ violate the constraint

[4] Additional domain knowledge for guiding the search is the subject of future work, and include: providing a direction for the search (upward, downward) based on estimated upper and lower bounds for a parameter; providing a step size for traversing a given range (such as tile sizes should be a multiple of the cache line size); exploiting characteristics of transformations (such as reuse increases monotonically with each unroll factor).

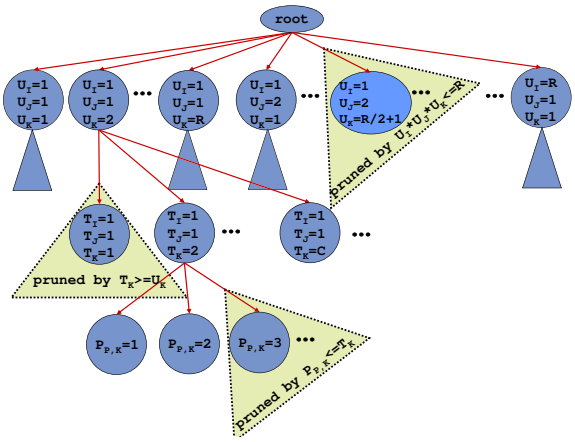

Fig. 3. Parameter space

$U_I * U_J * U_K \leq R$. Therefore all subtrees rooted at these nodes can be pruned. In addition, known properties of optimizations are used to guide and prune the search. For example, the amount of reuse exposed by unroll-and-jam increases with the unroll factors, until there are no more registers available and register spilling occurs. Hence when a set of unroll factors $U =< U_1, U_2, \ldots, U_n >$ results in a decrease in performance due to register spilling, all sets $V =< V_1, V_2, \ldots, V_n >$ such that $V_i \geq U_i$ can be pruned.

Starting points: At present, we use models to suggest a starting point for parameter values, based on the model's estimate of the optimal solution, and provide stopping criteria by estimating bounds for the performance of the optimized code variants.

3.2 AI Search Techniques

A multi-variable optimization problem, such as the one we are considering, can be cast as a search problem. The field of Artificial Intelligence (AI) has developed various search techniques for solving complex, multi-parameter optimization problems, which are characterized by very large and rough parameter landscapes. Search starts at some point in the parameter search space and progresses until a solution (a maximum in the objective function, such as performance) is found. Exhaustive algorithms, such as depth-first and breadth-first for searching trees, that evaluate every point in the parameter search space cannot be applied in practice due to the size of the search space. Methods such as hill climbing often fail due to roughness of landscape (that is, the existence of many local maxima). To address these issues, random and heuristic search algorithms have been developed [5].

Random search algorithms explore small neighborhoods of the search space at different points throughout the parameter space, keeping track of the quality of the solutions found. Typically, the search is terminated after some time when only a small portion of the search space has been explored. The resulting solution, while rarely the best, is often a good enough solution. Random algorithms have been shown to successfully solve hard optimization problems, such as GSAT.

Heuristic search introduces a function that evaluates the quality of the solution. The main differences between random and heuristic search techniques are how the parameter

space is explored and how the quality of a solution is evaluated. Heuristic hill climbing explores a local neighborhood of a solution, evaluating the new solution. If a new solution is better than the previous solution, the search resumes from this point. In effect, the search is guided to a local maximum. Simulated annealing and genetic algorithms typically choose a new solution at random, thus avoiding being stuck in local maxima. Simulated annealing in particular first samples many points in the parameter space randomly, then settles down for finer local search in the best neighborhood. Best-first search algorithms, on the other hand, choose a new point in the path to the best solution based on a heuristic, or an evaluation function. A* algorithm is a best first algorithm that includes the cost of getting to the current point in the parameter space in its evaluation function. Backtracking, or returning to a previous best solution, can be implemented to continue exploration of profitable paths while avoiding getting stuck in dead ends.

In future work, we plan to evaluate this set of AI search techniques to identify the contribution of domain knowledge to speeding up the search process, and compare the resulting code quality when search time is constrained.

4 Conclusion

This paper shows how the problem of optimizing for multiple levels of the memory hierarchy can be recast as a multi-variable optimization problem. We formalized our approach as an AI search problem and identified search algorithms suitable for our optimization problem. We feel this work is an important first step in a general strategy for developing a principled approach to solving complex multi-variable optimization problems in a compiler, such as managing locality and communication in parallel codes.

References

1. S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *International Journal of High Performance Computing Applications*, 14(3):189–204, Aug. 2000.
2. C. Chen, J. Chame, and M. W. Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *Proc. of the International Symposium on Code Generation and Optimization*, Mar. 2005.
3. K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. In *Proc. of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, May 1999.
4. M. Frigo. A fast Fourier transform compiler. In *Proc. of the Conference on Programming Language Design and Implementation*, May 1999.
5. N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufman, San Francisco, CA, 1998.
6. M. Stephenson, S. Amarasinghe, M. Rinard, and U. O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2003.
7. X. Vera, J. Abella, A. González, and J. Llosa. Optimizing program locality through CMEs and GAs. In *Proc. of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2003.
8. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, Jan. 2001.
9. K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2):358–386, Feb. 2005.