Automatic Creation of Tile Size Selection Models *

Tomofumi Yuki
Colorado State University
yuki@cs.colostate.edu

Lakshminarayanan Renganarayanan
IBM T.J. Watson Research Center
lrengan@us.ibm.com

Sanjay Rajopadhye
Colorado State University
Sanjay.Rajopadhye@cs.colostate.edu

Charles Anderson
Colorado State University
anderson@cs.colostate.edu

Alexandre E. Eichenberger
IBM T.J. Watson Research Center
alexe@us.ibm.com

Kevin O'Brien
IBM T.J. Watson Research Center
caomhin@us.ibm.com

Abstract

Tiling is a widely used loop transformation for exposing/exploiting parallelism and data locality. Effective use of tiling requires selection and tuning of the tile sizes. This is usually achieved by hand-crafting tile size selection (TSS) models that characterize the performance of the tiled program as a function of tile sizes. The best tile sizes are selected by either directly using the TSS model or by using the TSS model together with an empirical search. Hand-crafting accurate TSS models is hard, and adapting them to different architecture/compiler, or even keeping them up-to-date with respect to the evolution of a single compiler is often just as hard.

Instead of hand-crafting TSS models, can we automatically learn or create them? In this paper, we show that for a specific class of programs fairly accurate TSS models can be automatically created by using a combination of simple program features, synthetic kernels, and standard machine learning techniques. The automatic TSS model generation scheme can also be directly used for adapting the model and/or keeping it up-to-date. We evaluate our scheme on six different architecture-compiler combinations (chosen from three different architectures and four different compilers). The models learned by our method have consistently shown near-optimal performance (within 5% of the optimal on average) across all architecture-compiler combinations.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Optimization; Compilers

General Terms Performance, Algorithms, Languages

Keywords Tiling, Neural Network, Performance Modeling, Machine Learning

* This material is partly based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002

1. Introduction

The compute and data intensive kernels of several important applications are loops. Tiling [16, 35, 22, 41] restructures loop computations to exploit parallelism and/or data locality by matching the program characteristics (e.g., locality and parallelism) to those of the execution environment (e.g., memory hierarchy, registers, and number of processors). Effective use of tiling requires techniques for tile shape/size selection and tiled code generation. In this paper we focus on the key step of *tile size selection* (TSS).

TSS is an important step in the effective use of tiling. The importance is evident from the vast literature available on this topic, and is also highlighted in the performance chart shown in Figure 1. There is a 5x and 2.5x difference in performance between the best and worst tile sizes for Power5 and Opteron, respectively. TSS involves the development of a (cost) model, which is used to characterize and select the best tile sizes for a given combination of program, architecture, and compiler. For an overview of TSS models, see [31, 25].

TSS solutions can be broadly classified into two categories, viz., static model based [21, 35, 9, 15, 7, 34, 25, 42, 12, 33] and model-driven empirical search based [39, 10, 20, 8, 13, 30]. In the static model based approach, the compiler uses a pre-designed TSS model to pick the best tile sizes for a given program-architecture pair. In the model-driven empirical search approach, the TSS model is used to characterize and prune the space of good tile sizes. For each tile size in the pruned search space, a version of the program is generated and run on the target architecture, and the tile sizes with the least execution time is selected. Due to the large space of valid tile sizes an exhaustive search, without using any TSS model to prune the space, is often not feasible.

Both the static model and model-driven empirical search approaches require a well designed TSS model. Constructing a good TSS model is hard. The extensive literature on the TSS problem is evidence of the importance as well as the difficulty of the problem. The problem of creating accurate TSS models is further exacerbated by (i) the complexity of the memory hierarchy in multi-core processor architectures and (ii) the highly intertwined optimization phases of a compiler. For example, Yotov et al. [42, 43] show the level of detailed understanding of the architecture and compiler optimization required to construct effective TSS models.

In addition to the effort involved in creating a TSS model, adapting it to a different architecture and/or compiler requires significant effort. Further, keeping a TSS model up-to-date with respect to the evolution of optimizations in a single compiler is in itself a significant task. In fact, the recognition of this difficulty in constructing

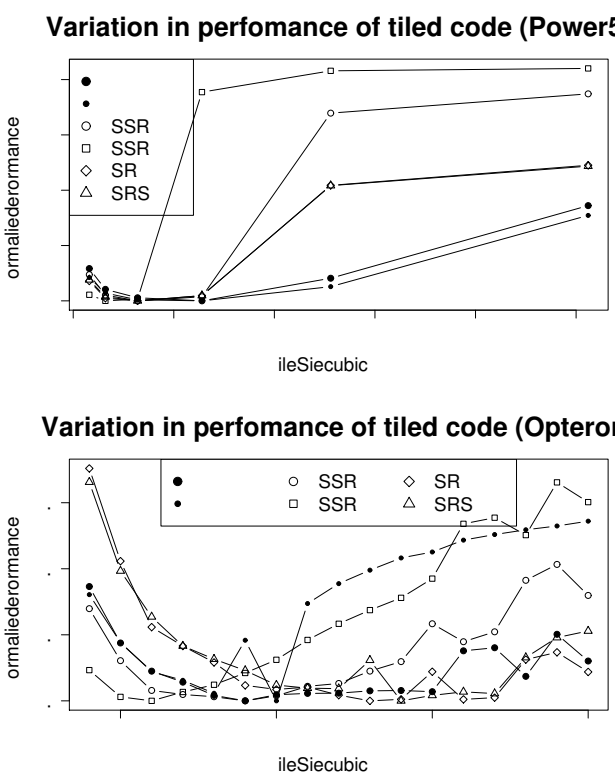

Figure 1. Variation in execution time of tiled code for six scientific kernels. The execution time is normalized to the best (lowest) running time of the points shown in the figure. Note that two processors show a very different behavior, and scale of x-axis is quite different in the two figures.

and maintaining accurate TSS models led to the wide spread use of empirical search techniques. Unfortunately, to be efficient and fast enough empirical search techniques also require TSS models to (at least) prune the search space, and these models themselves are also non-trivial to construct and adapt.

In summary, accurate TSS models are needed to select the best tile sizes and constructing and adapting them are becoming more and more difficult.

Previous approaches to TSS have used hand-crafted TSS models to either directly select the tile sizes [21, 35, 9, 15, 7, 34, 25, 42] or as a part of an empirical search to prune the search space [39, 10, 20, 8, 13, 30]. There are also TSS methods where hand-crafted TSS models are used to define a space of valid/good tile sizes and then machine learning techniques are used to efficiently search the space for the best tile sizes [38, 23, 11, 29]. As discussed earlier, the hand-crafted models used in these approaches are difficult to create, adapt, and maintain.

Instead of hand-crafting TSS models, can we automatically learn or create them? If so, we can use the same techniques to automatically adapt or keep them up-to-date with respect to changes in architectures and compilers. In this paper, we show, for a specific class of programs (3D loop nest with 2D data), that by using a combination of simple program features, synthetic kernels and standard machine learning techniques, highly effective and accurate TSS models can be learned with little or no human involvement. The two key ideas behind our approach are (i) the use of six simple program features that capture the effects of spatial and temporal locality of tiled programs and (ii) the use of synthetic and automatically generated programs to learn the TSS models.

We consider the problem of selecting tile sizes for a single level of tiling for caches. For validation, we use scientific computation kernels that are known to benefit from cache tiling. We report extensive validation of our scheme on three different architectures (Intel Core2Duo, AMD Opteron, Power5) and four different compilers (`gcc`, IBM `xlc`, PathScale `pathcc`, and Intel `icc`). We show that fairly accurate TSS models can be automatically created on all the six different architecture-compiler combinations. The tile sizes predicted by our machine-crafted models, trained separately for each architecture-compiler combination, consistently show near-optimal performance on a variety of scientific kernels. The training of the machine-crafted models involves a couple of days of data collection and very little effort to tune the neural network parameters. The resulting TSS model can be directly used by a compiler to compute the best tile sizes for a given program, or can be used by an autotuner to perform a model-driven empirical search.

The paper makes the following contributions:

- We identify a set of six simple program features that characterize the spatial and temporal locality benefits of a tiled program.
- We show that the simple structure of the program features can be exploited to generate synthetic tiled programs which can be used for learning the TSS models.
- We formulate a machine learning scheme which models the optimal tile sizes as a continuous function of the program features.
- We report extensive validation of our technique on six different compiler-architecture combinations. We show that very effective TSS models, which predict the near-optimal tile sizes across all the six platforms can be automatically learned for our target class of programs.

To the best of our knowledge, our work is the first one to use a combination of a simple set of features and synthetic programs to automatically create TSS models.

The rest of the sections are organized as follows. Section 2 introduces loop tiling for caches and the architectural features that affect the performance of tiled programs. Section 3 introduces the program features, predicted outputs, and the use of artificial neural networks to learn TSS models. In Section 4, we outline the different stages of our scheme and in Section 5 we present experimental evaluation. Section 6 compares the performance of the machine-crafted TSS models with two well known hand-crafted TSS models from the literature. Section 7 presents some conclusions and pointers to future work.

2. Loop Tiling For Cache Locality

Cache tiling, also called cache blocking, transforms a set of loops into another set of loops, which performs the same computation but in a different order so that the program has better cache locality. Tiling is used to maximize reuse and avoid costly accesses to lower levels in the memory hierarchy. For example, consider the loop nest in Figure 2, which is the symmetric rank k update (SSYRK) kernel. If you consider the access to the C matrix, the entire matrix is accessed in each iteration of the outer-most i loop. If the cache is large enough to store the entire C matrix, the program will read the matrix from the main memory when it was first used, and then the remaining accesses only needs to wait for the cache. With the original code, we need enough cache to hold N^2 elements of the C array, and additional $2N$ for the A array to maximize reuse. However, L1 data caches of modern processors are around 32KB to 128KB, and are often not enough to store the entire matrix for large problem instances.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=j; k<N; k++)
      C[j][k] += A[i][j] * A[i][k];
```

Figure 2. SSYRK

```
for (Tj = 0; Tj < N; Tj+=tSize)
  for (Tk = Tj; Tk < N; Tk+=tSize)
    for (i=0; i<N; i++)
      for (j=Tj; j<min(Tj+tsize,N); j++)
        for (k=j; k<min(Tk+tsize,N); k++)
          C[j][k] += A[i][j] * A[i][k];
```

Figure 3. Tiled SSYRK

```
for(i=0; i<N; i++)
  for(j=0; j<N; j++)
    for(k=0; k<=j; k++)
      if (k <= j-1)
        B[j][i]=B[j][i]-L[j][k]*B[k][i];
      if (k == j)
        B[j][i]=B[j][i]/L[j][j];
```

Figure 4. TRISOLV

Tiling partitions the iteration space and changes the order of execution so that data will be reused while still performing the same computation. A possible application of tiling to the SSYRK kernel is shown in Figure 3. New loops with indices Tj and Tk control the memory requirement by the tile size parameter $tSize$ so that $tSize^2$ elements fit in the available cache.

Tiling has been studied extensively and is an important optimization in highly tuned linear algebra libraries such as BLAS or ATLAS [40, 43]. The above example only shows the use of tiling to maximize L1 cache reuse, but tiling can be applied simultaneously to other memory hierarchies such as registers, and other levels of caches. In addition, multiple levels of tiling can be applied to optimize for multiple levels of the memory system.

2.1 Cache Tiling and Hardware Prefetching

Let us consider a hardware prefetcher that can prefetch subsequent cache lines to L1 cache when cache lines are accessed sequentially, such as the one in Power5 processor. On such a processor, all the references in Figure 2 can be prefetched, because all of them are along the cache line (assuming row-major layout). With hardware prefetching, the untiled code performs slightly better than tiled code with best tile size, since it does not suffer from loop control overhead associated with tiling.

However, not all programs have prefetcher-friendly structure. Consider Figure 4. Because the k loop is dependent on the outer j loop, simple loop permutation cannot make the reference `B[k][i]` prefetcher-friendly. Also, multiple computations may be fused in practice, which may result in a prefetcher-unfriendly code segment. Again by loop fusion, the total number of references in a loop nest may increase beyond the number of prefetcher streams available, which again limits the benefit of hardware prefetching.

It has been shown by Kamil et al. [17] that cache blocking may not be effective for stencil computations with 2D data. Stencil computations have uniform accesses that can be easily prefetched . Combined with large on-chip memories available on modern hardware, the level of reuse already achieved without transforming the loop is comparable with tiled code. We therefore exclude stencil computations from the target class of programs in this paper.

MMM	Matrix Matrix Multiplication
TMM	Triangular MM ($C = AB$)
SSYRK	Symmetric Rank K Update
SSYR2K	Symmetric Rank 2K Update
STRMM	In-place TMM ($B = AB$)
STRSM	Solve Triangular Matrix ($AX = \alpha B$)
TRISOLV	Solve Triangles ($Ax = b$)
LUD	LU Decomposition
SSYMM	Symmetric MMM

Table 1. Nine real kernels used for validation

Some of the recent architectures have even better hardware prefetchers that start prefetching when constant-stride accesses are observed in addition to the unit-stride accesses. With constant-stride prefetcher, references that access by columns (not along the cache line) can also be prefetched if the access pattern is regular. However, recent Intel processor series, which were the only processors with constant-stride hardware prefetchers, cannot prefetch if the stride crosses 4KB page boundaries [1]. Because of this constraint, constant-stride prefetcher in the Intel processors cannot do any better than unit-stride prefetchers when the problem instance is large enough so that each row is more than 4KB (512 doubles).

When optimizing for multi-core, tiling is used to expose parallelism and improve cache locality [3]. Some loop permutations that would make a loop nest prefetcher-friendly might not be the best transformation for parallelism. Consider SSYRK in Figure 2. Permuting the loops to make i loop the innermost loop can reduce the synchronization overhead when this code is parallelized for a shared memory environment. If this code were to be parallelized using OpenMP, one would prefer reordering the loops so that the j loop is outermost, and mark the outermost loop as the parallel loop. The number of synchronization is now reduced from N to 1, compared to inserting the parallelization pragma before the i loop in the original code (Figure 2). However, the new loop ordering (j,k,i) makes the two references to array a are prefetcher-unfriendly, and tiling could be used for better locality.

2.2 Target Class of Programs

In this paper, we focus on a class of scientific computations, such as linear algebra, which are known to benefit from tiling. Tiled code generators, such as TLoG [32], Pluto [3], PrimeTile [14], and D-tiling [18], are available for these class of programs, which make it easier to use our TSS learning approach. Although there are highly tuned libraries available for common kernels like matrix multiplication, computations that are not covered by the libraries may still come up by trying to use a specific loop ordering or as a result of other transformations, such as fusing multiple kernel computations. The nine kernels we use in this paper are summarized in Table 1.

Among this class of programs, we consider the programs that have three dimensional loops with two dimensional data. Many scientific kernels, like matrix multiplication, fit in to this subset of programs. Also programs with more than three dimensional loops can be still handled in our model by only tiling the inner three dimensions. We consider cubic tiles only to reduce data collection time. Allowing all three dimensions to have different tile sizes significantly increases the number of possible tile sizes. Our approach can be directly extended to predict rectangular tile sizes by increasing the data collection time, which will be described later in more detail. We do not consider data padding or copy optimization, since available code generators do not handle these optimizations.

```
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
    for (k=0; k<N; k++)
      C[i][j] += A[i][k] * B[k][j];
```

Figure 5. Matrix Matrix Multiplication

3. Learning TSS Models

In this section we describe the basic components of TSS model learning; the inputs to and outputs of the model, and the machine learning technique we use.

3.1 Input Program Features

In order to use a model to predict optimal tile sizes for different programs, the model needs to be provided with inputs that distinguish different programs. The inputs to our model are features of the programs. Previous methods that use machine learning techniques for compiler optimizations have often used syntactic features such as the number of operands or the number of loop nests [36, 27, 6]. After experimenting with a variety of features that capture the spatial and temporal locality effects of loop tiling, we arrived at a set of six features.

The features are based on the number of references in the innermost statements, classified into three different types of references. Then each type of reference is further classified into reads and writes for a total of six features. The three types of references are *non-prefetched* references, *prefetched* references, and references that are constant in the innermost loop (*invariant*). The *invariant* reference captures those references that are reused for all the iterations of the innermost loop. The *prefetched* reference captures references that enjoy spatial locality given by the prefetcher, and *non-prefetched* references are those that need temporal locality for good performance. Read and write references are distinguished because of the possible differences in how they are treated especially in multi-core processors where L2 cache is commonly shared among the cores.

For example, in matrix multiplication shown in Figure 5, the reference to array *c* is *write-invariant* (WI), because it is written to the same location by all iterations of the innermost loop. Reference to array *a* is *read-prefetched* (RP), because the innermost loop index *k* is used to index the columns of the array, and such accesses are prefetched by unit-stride prefetcher. Reference to array *b* is *read-non-prefetched* (RNP), since *k* is used to index the rows, and we are assuming row-major layout. These features can easily be extracted by looking at the loop orderings used to reference arrays. The compiler needs to be aware of the type of hardware prefetcher is used on each architecture to calculate these values, but we believe this is a simple requirement. Detailed information about the prefetcher is not required, the compiler only needs to know if unit-stride access is prefetched. With the current hardware prefetching technology, even a constant-stride prefetcher works effectively as an unit-stride prefetcher as discussed earlier. More information may become necessary with significant advance in hardware prefetching.

3.2 Predicted Outputs

There are multiple possible outputs for a TSS model. One possibility is to model the performance of tiled code itself and search the learned function for tile sizes that optimize performance given a specific program. However, this approach requires searching the function after modeling. In the case of analytical models, the function may be smooth and the optimal is easy to find using some kind of optimization method. With functions learned by neural networks, the function may not be smooth and optimization methods can get trapped in one of many local minima. Finding global minima in such functions is itself a separate and difficult problem, which we would like to avoid.

Another possible output is the optimal tile size itself. Classifying programs to optimal tile sizes allows skipping the step of searching the function and directly gives tile size as an output. Classifiers can be also learned by neural networks or other machine learning techniques such as support vector machines. However, simple classification is only capable of classifying programs into predefined set of optimal tile sizes. Thus, it does not suit our goal of predicting optimal tile sizes of unseen programs, unless we have enough training data to cover all possible tile sizes, which is unlikely.

We used a solution between the two ways described above. We use the optimal tile size as the output, but we do not learn classifiers. Instead, we formulate the TSS model to be learned as a continuous function from the six program features (described earlier in 3.1) to the optimal tile sizes. The following example gives an intuitive motivation for formulating the TSS model to be a continuous function. Consider three programs with identical program features except for number of RNP references. Program A has optimal tile size of 100 with RNP=1, program B has optimal tile size of 50 with RNP=3. It is reasonable to think that program C with RNP=2 has optimal tile size somewhere between 50 and 100, since it requires more data per iteration compared to program A, but less compared to B.

3.3 Using ANN to Learn TSS Model

We used artificial neural networks (ANN), a supervised learning method, to learn tile size prediction model. Supervised learning methods require pairs of input and desired output, and learn some function to minimize error between the output of the function and the desired output. The neural network we used is a back-propagation neural network and was trained with Scaled Conjugate Gradient method [26].

We use multi-layered neural networks [2]. The number of nodes in the output layer is equal to the number of outputs to be produced by the neural network. Each hidden layer can have any number of nodes. The inputs to the ANN is given to the first hidden layer, and outputs from each layer is given to the next layer. The outputs from hidden layers is a function of weighted sum of the inputs to that layer, where the function is the hyperbolic tangent. The output layer performs weighted sum of the outputs from the last hidden layer, but does not apply the hyperbolic tangent function. Given output layer weights w, output from the last hidden layer h, desired output b, and number of training data N, each output layer node tries to minimize the error calculated using the equation below.

$$\Sigma_{n=1}^{N}((h.w)_n - b_n)^2$$

The neural network starts with random values as initial weights in each node, and then iteratively updates the weights to minimize the error. The scaled conjugate gradient method is a type of gradient method that approximates the second order derivative to allow faster convergence to the local minimum, and hence accelerate training of ANN compared to standard back-propagation with gradient descent.

Models learned using neural networks return real valued numbers as optimal tile size. Since tile sizes are integers, we simply round the given value to a closest integer and use that as the predicted optimal tile size.

4. Our Approach

Our approach has the following four different stages

1. Synthetic program generation
2. Data Collection
3. Learning TSS models using ANN
4. Use of ANN based TSS model

Stages 1 through 3 are part of the TSS model creation phase and are done offline. Stage 4 represents the use of the learned TSS model and is done on-line during the compilation of a program to select the tile sizes.

4.1 Synthetic Program Generation

We need to collect training data to train neural networks. Data gathered from real applications or kernels are commonly used as the training data for machine learning based modeling. However, using real applications limits the training data to the real applications available at the time of training. The neural network cannot be expected to perform well on programs with features that are significantly different from any program in the training data. With real applications as training data, there is not much control over the range of programs that is covered by the neural network. In addition, some of the real applications need to be separated out from the training data for validation. Also, if multiple applications have the same program feature, the neural networks may become overtrained to better suit that program feature more than others.

We use synthetic programs to overcome these limitations. The synthetic programs we use are programs that fit in our class of interest (three dimensional loops and two dimensional data), with statements in the innermost loop that are generated to have the specified number of references for each type. We exhaustively search for optimal tile sizes of the generated programs to create the training data set. We used the open source tiled code generator, TLoG [32], to generate codes with all three dimensions tiled.

With synthetic programs, we have better control over training data, and the ability to train using a large number of training data points. We believe these benefits given by synthetic programs are one of the main reasons that lead to well performing models.

The use of synthetic programs was only possible because we have simple program features. If a large number of program features were used, then it becomes infeasible to try a large range of possible programs with synthetic programs. Even if real programs were used, the coverage of programs that can be represented by complex program features is going to be sparse.

We selected a range of values of program features, and hence programs, that cover all the kernels we used, but also includes many others so that the model is not specialized to just those kernels. Table 2 shows the range of values we used to bound the feature space. Column names represent the type of reference, prefetched (P), non-prefetched (NP), invariant(I) for read (R) and write (W). These bounds were constructed so that the space is not too large, but still captures a wide variety of programs. The number of reads are usually more than the writes, so we only have a small number of writes. RNP is always greater than 0, to ensure the program stays in the class of interest (at least one reference is not prefetched). There are a total of 2835 program instances in this space with at least one write.

The range of program features to be covered can be changed to better suit expected programs. If you know that the compiler is going to see a small subset of the possible feature space beforehand, the model can be specialized for that range of programs.

From the bounded feature space, we collected optimal tile sizes for a number of points in the feature space. Table 3 shows the points used for the training data. We also exclude from the training data, programs with features identical to features of real kernels, so that real kernels we used to test our model remain unseen during the training.

	RP	RNP	RI	WP	WNP	WI
Range	0-8	1-5	0-8	0-1	0-1	0-1

Table 2. Bounds on feature space for the model as number of references of each type

	RP	RNP	RI	WP	WNP	WI
Range	0,2,4,8	1-5	0,2,4,8	0-1	0-1	0-1

Table 3. Data points used for training

4.2 Data Collection

Collecting training data is time consuming, but can be done without much human effort. The use of a parameterized tiled code generator helped our data collection by avoiding compilation for each tile size explored for a program.

Data collection took between 20-40 hours for each compiler-architecture combination. We used problem sizes that take around 10 seconds of execution time for each synthetic program instance. Because the number of references in a program affects its running time, using the same problem sizes would cause the running time of some synthetic program instances to be much larger than 10 seconds. We used a very simple linear function of number of references to adjust the problem size. This adjustment was only for avoiding long execution times during data collection, and the accuracy of the adjustment was not an issue.

4.3 Learning TSS Models Using ANN

There are many parameters in the training process, including the range of programs to target, the range of training data, and parameters of the neural network. The former two can be made larger and larger if time permits, since we want the model to cover a larger range of programs, and having more training data always helps learning.

The parameters of the neural network are not as simple. There are many parameters for a neural network, the important ones being the number of hidden layers, the number of nodes in each layer, how initial weights are selected, and the termination condition. In addition, we can train multiple neural networks individually for more stable output. Averaging the output of multiple neural networks helps stabilize the output, because neural networks learned are heavily influenced by the initial weights. The number of neural networks to be trained for averaging in the end is also another parameter. We do not try to optimize the neural network parameters. Instead we manually tune the neural network parameters based on our intuition and testing on small data sets. We hope to develop an automated approach to optimize neural network parameters in the future.

We used three-layered (two hidden layers) neural networks with 30 hidden nodes per hidden layer, and weights randomly initialized to values between 1 and -1. The termination condition was slightly different for each architecture-compiler combination based on how easy it was to fit the training data for a particular combination. These parameters are picked by trying out multiple combinations of parameters and looking at the rate of convergence and root mean square errors, a measure of how far the predicted outputs are away from the desired outputs.

It took about two hours of initial tuning to get a good basic design of the neural network, and then the SAME basic neural network configuration was applied to all architecture-compiler com-

binations. Note that this design time is a one-time effort. After the basic design, for each architecture-compiler combination, a slight tuning of the termination condition was needed. We tuned the termination condition using the collected training data, by looking at how the error between predicted and desired output changes between iterations. Where the improvement starts to flatten could be different for each training data, and we selected a condition to terminate when it starts to flatten. This tuning is pretty standard and can be automated.

With the above configuration, training of each neural network completes within a minute for a total of at most five minutes for five different neural networks trained for each architecture-compiler combination.

4.4 Use of ANN Based TSS Model

Once we have the trained TSS model, it can be used as a part of the compiler to predict the optimal tile sizes, or used as a part of a model-driven search method to find the optimal tile sizes. The first step is to extract the program features, which should not take much time due to its simplicity. Then the six program features are used as an input to the learned model to produce the output, which can be directly used as the tile size selected for that program. When the model is used as a part of a model-driven search method, the tile sizes in the neighborhood of the predicted tile sizes can be empirically tested for the best performance. It is also possible to alter the neural network to output expected performance of a program with a given tile size, which may be a better model for mode-driven search.

The only on-line cost associated with the use of our model in a compiler is the use of the neural network and extraction of the six program features. The use of neural networks is computationally close to two matrix-vector products of size 31x6, which is trivial with modern compute power.

4.5 Extension to Rectangular Tiles

Our approach can be directly extended to predict rectangular tile sizes as well. The flow described above does not change at all. The only pieces that need to be changed is the outputs of the neural network, and the data collection. The outputs from the neural network needs to be increased from one to two or three, depending on whether two or three dimensions are tiled, because tile size in each dimension can be now different. The data collection phase needs to be adjusted accordingly to find the optimal tile sizes including rectangular tiles. Because neural networks naturally handle more than one outputs, no other change is required.

5. Experimental Validation of Machine-Crafted Models

We have used our approach to learn tile size selection models on the six architecture-compiler combinations summarized in Table 4. We used the nine kernels previously summarized in 1 with problem sizes adjusted for each architecture to run for about a minute. We chose a longer target execution time compared to the 10 seconds of the synthetic programs so that the problem sizes used were significantly different from the problem sizes used during validation. The problem sizes used for real kernels were around 3000, where as the problem sizes used during training was around 1000.

Feature extraction was done manually, but it can be easily done in a compiler. We used the same set of program features on all architectures, because even the constant-stride prefetcher on Core2Duo effectively worked only as a unit-stride prefetcher due to its constraint on access distance.

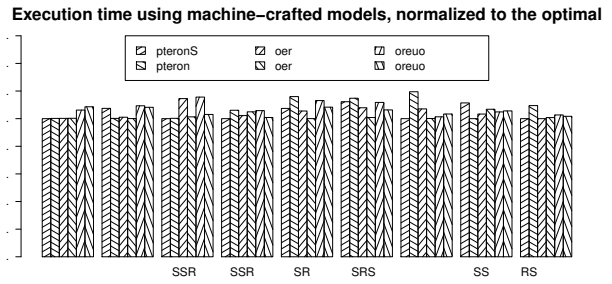

Figure 6. Execution time normalized to the true optimal of kernels with tile sizes selected by machine-crafted models for each combination of architecture-compilers.

5.1 Performance

For each architecture-compiler combination, we compare the execution time of kernels with the true optimal (found by exhaustive search) and the predicted optimal tile sizes. Figure 6 shows the normalized execution time of nine kernels with tile sizes selected by machine-crafted models learned for each architecture-compiler combination. The performance given by tile sizes predicted by our machine-crafted models is consistently near the optimal. The performance is only 20% off the optimal even in the worst case, which is significantly small compared to the slowdown one would get with a poor tile size (recall Figure 1). This supports our claim that well performing TSS models can be learned from simple features and training data collected through synthetic programs for different architecture and compilers. Although we do not have results for different versions of the same compiler, we have shown results for different compilers, which is just as hard if not harder. This indicates that the learned TSS models can be easily updated (re-trained) with respect to the evolution of a single compiler.

5.2 Performance with Local Search

The focus of this paper is on learning TSS models that can predict good tile sizes for different architecture and compilers without much human effort. In this section we quantify its potential when used as a part of model-driven empirical search approaches. We show how close the predicted tile sizes is to the optimal by simply looking at neighboring tile sizes within a certain distance. Here, distance is the difference between the predicted tile size and the tile sizes actually explored.

Table 5 shows the mean slowdown over all nine kernels when the best tile size within a certain distance of the predicted tile size were used. Only by looking at immediate neighbors of distance ten, the model can give the exact optimal performance for all kernels on Opteron, and for eight out of the nine kernels for Power5. The performance improvement on Core2Duo is relatively small compared to other architectures, but notable improvement can be observed.

We think the cause of relatively small improvement on Core2Duo is due to the very high set-associativity (8-way). There is a very wide range of good tile sizes, and the automated training data col-

Architecture	Compilers	L1 Cache	Options
Opteron	PSC, GCC	64KB 2-way	-O3, -O3
Power5	XLC, GCC	32KB 4-way	-O5, -O3
Core2Duo	ICC, GCC	32KB 8-way	-O3, -O3

Table 4. Architecture and compilers used

	Predicted	Distance 10	Distance 20
Opteron/PSC	4.3%	0%	0%
Opteron/GCC	6.3%	0%	0%
Power5/XLC	4.6%	0.4%	0%
Power5/GCC	1.7%	0.2%	0%
Core2Duo/ICC	7.8%	4.0%	1.9%
Core2Duo/GCC	5.1%	4.0%	1.9%

Table 5. Mean slowdown over all kernels when the best tile size within some distance from the predicted tile size is used.

	Machine-Crafted	LRW	EUC	Yotov et al.
Opteron/PSC	4%	178%	217%	169%
Opteron/gcc	6%	100%	147%	139%
Power5/XLC	5%	168%	268%	9%
Power5/gcc	2%	77%	133%	3%
Core2Duo/ICC	8%	6%	246%	5%
Core2Duo/gcc	5%	3%	128%	4%

Table 6. Mean of slowdowns with tile sizes predicted by each model over all nine kernels on each architecture-compiler combination

lection is likely to have more noise compared to others. The optimal on a flat surface can be easily affected by small noises from the operating system or other environment not necessarily connected to the program being executed. We believe this noise can be suppressed by increasing the problem size and/or running each instance of the synthetic program a number of times and taking the minimum (or mean) execution time

Another reason is that with GCC on Core2Duo, two kernels, MMM and TMM, are showing optimal performance with a tile size that is far from the tile size predicted by our model. We suspect that this is due to some optimization applied by the compiler only for the standard matrix multiplication, since by changing the compiler option from -O3 to -O2, the tile size predicted by our model shows identical performance with the true optimal.

How good are the models created by our approach, compared to randomly picking points in the search space, when local search is performed is a question that the readers may have. It has also been shown by Kisuki et al. [19] that random search takes large number of iterations (100+) to reach within 5% of the optimal for most of the benchmarks they used, which included dense linear algebra computations as well.

Even a naive local search around the tile sizes predicted by the machine-crafted models show significant improvements. This demonstrates its potential to perform better when combined with the more sophisticated search methods proposed in the literature.

6. Comparison with Hand-Crafted Models

Many static models have been previously developed to maximize performance of tiled code, and those models are analyzed in detail by Hsu and Kremer [15]. We have taken two models that were reported to perform well [15], and another model by Yotov et. al [42] used in ATLAS for comparison. First, we briefly describe the logic behind the three models, LRW [21], EUC [9] and the model-driven ATLAS model [42]. Then these models are evaluated in the same manner as in the previous section. The goal of this section is to show that our approach can generate models that are at least as good as the previous models for all architecture-compiler combinations.

LRW [21] is an analytical model developed by closely studying the performance of tiled matrix multiplication. It chooses square tiles for a given program using an estimate of cache misses based on memory footprint and self-interference. EUC [9] is also an analytical model with similar considerations used as in LRW, but it predicts rectangular tile sizes instead of square. EUC takes cross-interference into account as well as self-interference. Both of these models take problem sizes as an input, where as our model does not. Yotov et al. [42] uses multiple models to compute many parameters including tile sizes for cache tiling with cubic tiles. Their model for tile size selection assumes fully associative caches and does not consider conflict misses. We would like to clarify that when we refer to Yotov et al. in the following, we did not use the ATLAS auto-tuned kernels. We only used the analytical model for predicting square tile sizes for cache locality in [42].

6.1 Tailoring Hand-Crafted Models to Modern Architectures

We made a small but necessary modification to all the hand-crafted models to adapt to the current architecture. Since some of the hand-crafted models were developed when hardware prefetcher was not commonly available, they treat all references as non-prefetched. However, it is obvious that prefetched references do not have to stay in the cache, and they can be excluded when calculating the tile size so that the cache is utilized well. Because it is straight-forward and it would not take much effort to modify the model to take the prefetching into account, we modified all of the models so that prefetched references are excluded from calculation. Further, for programs that has more than one references that is not prefetched, we give smaller cache sizes to the model. Also, in [21], they briefly mention extension of their algorithm to set associative caches, we have used their modification so that larger tile sizes are selected when compared to assuming direct mapped cache.

6.2 Performance of Hand-Crafted Models

Figure 7 shows the normalized execution time of nine kernels using tile sizes given by hand-crafted static models. The same problem sizes used for measuring execution time using machine-crafted models were used. Although the hand-crafted models have predicted near optimal tile sizes for some of the kernels, the performance is not as consistent as what was observed with our machine-crafted models. LRW and Yotov et al. model performs relatively worse on Opteron compared to the other architectures. Opteron has smaller set associativity, which seems to make the performance more sensitive to small changes in tile size, leading to greater performance hit. LRW does perform more than 2x slower on Opteron with matrix multiplication that was used to develop the model.

EUC was unable to give good tile sizes for some kernels across all architecture-compiler combinations. It is interesting to note that EUC had predicted a tile size that performs better than the optimal cubic tile found for SSYR2K on Opteron with PSC. The optimal cubic tile size found for SSYR2K was 6x6x6, but it was not the optimal when rectangular tiles were considered. EUC was able to find a very thin tile that has better memory utilization for this case.

Table 6 summarizes the effectiveness of each model by showing the percentage of slowdown when compared to the optimal using cubic tile sizes. LRW and Yotov et al. model show comparable performance on some combinations, but overall the machine-crafted model provides consistently near-optimal performance across all architectures. We believe the reason for LRW and Yotov et al. model showing comparable performance on Core2Duo and Power5 is also due to the fact that these processors have very wide ranges of good tile sizes, as previously discussed in Section 5.2. In addition relatively high set associativity of these processors are closer to the assumption of Yotov et al. that caches are fully associative.

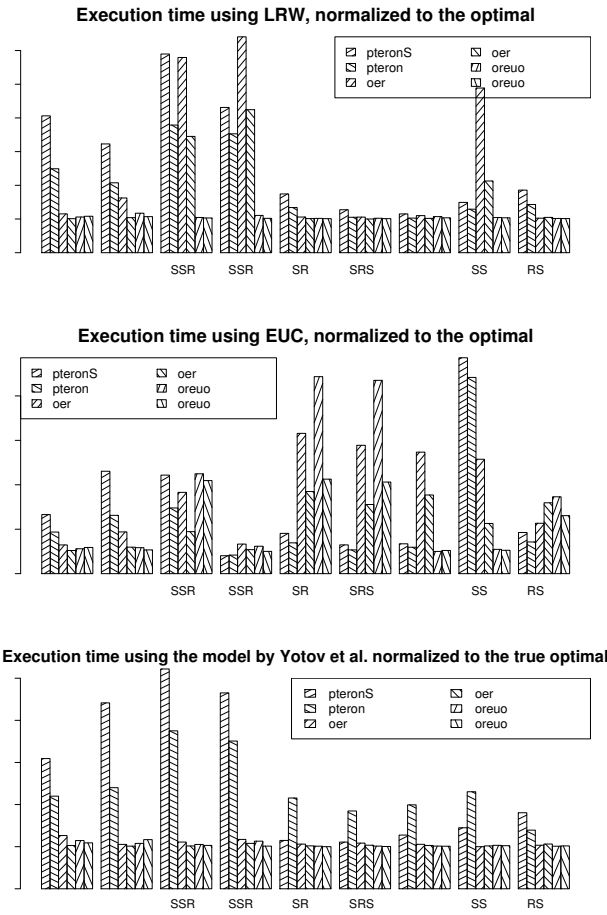

Figure 7. Execution time of kernels with tile sizes selected by hand-crafted models for each combination of architecture-compilers, normalized to the optimal.

7. Related Work

Many analytical models have been proposed in the past for TSS [21, 35, 9, 15, 7, 34, 25, 42, 12, 33]. These models are constructed by carefully observing the performance of a small set of kernels and modeling the performance using detailed hardware and software characteristics. Although developing analytical models can give greater insight into how the hardware and software interacts, the cost of development is quite high. Our work focuses on creating good tile size selection models with little human effort.

Another class of tile size selection techniques is the model-driven empirical search methods [39, 10, 20, 8, 13, 30, 19, 38, 29]. Empirical tuning with global search may be a feasible solution for optimizing libraries of commonly used kernels [39], but is not feasible for optimization during compilation. Model-driven approaches share the same motivation of achieving performance close to what can be obtained by global empirical search, but with less overhead. Replacing the hand-crafted models currently used in these approaches is a possible application of our work.

Recently, machine learning techniques have been successfully used in compiler optimization. Many of the applications were toward deriving models and heuristics to accurately predict the behavior of modern complex architectures. The wide range of applications include branch prediction [4], instruction scheduling within basic blocks [28, 24], and deciding if certain optimization should be applied [5, 6, 37, 27]. Some of the hand-crafted tile size selection approaches have also used some form of machine learning methods [38, 23, 11, 29, 36]. We use machine learning techniques to automatically learn TSS models.

8. Conclusions and Future Work

Tile size selection is an important step in the profitable use of loop tiling. Hand-crafting effective TSS models is hard and adapting or maintaining them is often harder. We have shown that highly effective TSS models can be automatically created using a small set of program features, synthetic programs and standard machine learning techniques. We have shown that the machine-crafted TSS models consistently predict near-optimal (on the average, within 5% of optimal) tile sizes across six different compiler-architecture combinations. We have also shown that, a naive search within a small neighborhood of the predicted tile sizes can find the true optimal tile sizes in some cases, and in other cases find tile sizes that are very close to the optimal. This clearly indicates the strong potential of machine-crafted TSS models in a model-driven empirical search scheme.

Several directions of future work are promising. The proposed approach can be directly extended to construct TSS models for multiple levels (cache and register) tiling. We have described how our approach can be extended to rectangular tile sizes with little change, which is also a possible direct extension. Another direct extension is to construct TSS models where tiling is used to expose coarse-grain parallelism [3]. Another promising direction is the use of machine-crafted TSS models together with sophisticated search techniques to develop an efficient model-driven empirical search technique for use in auto-tuners. Our approach in itself is quite flexible and with appropriate interpretation of program features, it can be extended to other class of programs. Extending to programs with irregular array references is another interesting direction.

There are some possible future work on the neural network side of our approach as well. As mentioned previously, automatic selection of neural network parameters is the only part that is not automated in our approach. There are well known approaches for selecting neural network structure, which may be applied to fully automate our approach. Our model cannot provide any insight about the underlying architecture, where the process of developing analytical models by hand tend to increase the understanding of the underlying architecture and programs as a sub-product. It would be useful if we could extract some insight from the trained neural network.

References

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

[2] C.M. Bishop et al. *Pattern recognition and machine learning*. Springer New York:, 2006.

[3] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral program optimization system. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2008.

[4] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zoren. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1):188–222, January 1997.

[5] J. Cavazos and J.E.B. Moss. Inducing heuristics to decide whether to schedule. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 183–194, 2004.

[6] J. Cavazos and M.F.P. O'Boyle. Method-specific dynamic compilation using logistic regression. In *Proceedings of the 21st annual ACM*

SIGPLAN conference on Object-oriented programming languages, systems, and applications, pages 229–240, 2006.

[7] Jacqueline Chame and Sungdo Moon. A tile selection algorithm for data locality and cache interference. In *1999 ACM International Conference on Supercomputing*, pages 492–499. ACM Press, 1999.

[8] Chun Chen, Jacqueline Chame, and Mary Hall. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 111–122, Washington, DC, USA, 2005. IEEE Computer Society.

[9] S. Coleman and K.S. McKinley. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 279–290. ACM New York, NY, USA, 1995.

[10] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley, and K. Yelick. Self-Adapting Linear Algebra Algorithms and Software. In *Proceedings of the IEEE*, 93(2):293, 2005.

[11] Arkady Epshteyn, María Jesús Garzarán, Gerald DeJong, David A. Padua, Gang Ren, Xiaoming Li, Kamen Yotov, and Keshav Pingali. Analytic models and empirical search: A hybrid approach to code optimization. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 259–273, 2005.

[12] K. Esseghir. Improving data locality for caches. Master's thesis, Rice University, 1993.

[13] Basilio B. Fraguela, M. G. Carmueja, and Diego Andrade. Optimal tile size selection guided by analytical models. In *PARCO*, pages 565–572, 2005.

[14] A. Hartono, M.M. Baskaran, C. Bastoul, A. Cohen, S. Krishnamoorthy, B. Norris, J. Ramanujam, and P. Sadayappan. Parametric multi-level tiling of imperfectly nested loops. In *Proceedings of the 23rd international conference on Conference on Supercomputing*, pages 147–157. ACM New York, NY, USA, 2009.

[15] Chung-Hsing Hsu and Ulrich Kremer. A quantitative analysis of tile size selection algorithms. *J. Supercomput.*, 27(3):279–294, 2004.

[16] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, pages 319–328. ACM, Jan 1988.

[17] Shoaib Kamil, Parry Husbands, Leonid Oliker, John Shalf, and Katherine Yelick. Impact of modern memory subsystems on cache optimizations for stencil computations. In *Proceedings of the Workshop on Memory System Performance*, pages 36–43, New York, NY, USA, 2005. ACM Press.

[18] DaeGon Kim and Sanjay Rajopadhye. Efficient tiled loop generation: D-tiling. In *The 22nd International Workshop on Languages and Compilers for Parallel Computing*, 2009.

[19] T. Kisuki, P.M.W. Knijnenburg, and MFP O' Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237. Citeseer, 2000.

[20] P. M. W. Knijnenburg, T. Kisuki, K. Gallivan, and M. F. P. O'Boyle. The effect of cache models on iterative compilation for combined tiling and unrolling. *Concurr. Comput. : Pract. Exper.*, 16(2-3):247–270, 2004.

[21] M.D. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. *Proceedings of the 4th international conference on architectural support for programming languages and operating systems*, 25:63–74, 1991.

[22] Monica S. Lam and Michael E. Wolf. A data locality optimizing algorithm (with retrospective). In *Best of PLDI*, pages 442–459, 1991.

[23] Xiaoming Li and María Jesús Garzarán. Optimizing matrix multiplication with a classifier learning system. *Workshop on Languages and Compilers for Parallel Computing*, pages 121–135, 2005.

[24] A. McGovern, E. Moss, and A. Barto. Scheduling straight-line code using reinforcement learning and rollouts. (UM-CS-1999-023), , 1999.

[25] N. Mitchell, N. Hogstedt, L. Carter, and J. Ferrante. Quantifying the multi-level nature of tiling interactions. *International Journal of Parallel Programming*, 26(6):641–670, 1998.

[26] Martin F. Møller. A scaled conjugate gradient algorithm for fast supervised learning. *Neural Networks*, 6:525–533, 1993.

[27] A. Monsifrot, F. Bodin, and R. Quiniou. A machine learning approach to automatic production of compiler heuristics. *Lecture notes in computer science*, pages 41–50, 2002.

[28] Eliot Moss, Paul Utgoff, John Cavazos, Doina Precup, Darko Stefanovic, Carla Brodley, and David Scheeff. Learning to schedule straight-line code. In *Proceedings of Neural Information Processing Symposium*, pages 929–935. MIT Press, 1997.

[29] Saeed Parsa and Shahriar Lotfi. A new genetic algorithm for loop tiling. *The Journal of Supercomputing*, 37(3):249–269, 2006.

[30] Apan Qasem and Ken Kennedy. Profitable loop fusion and tiling using model-driven empirical search. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*, pages 249–258, New York, NY, USA, 2006. ACM.

[31] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. Positivity, posynomials and tile size selection. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[32] Lakshminarayanan Renganarayanan, DaeGon Kim, Sanjay Rajopadhye, and Michelle Mills Strout. Parameterized tiled loops for free. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 405–414, New York, NY, USA, 2007. ACM.

[33] Gabriel Rivera and Chau wen Tseng. A comparison of compiler tiling algorithms. In *Proceedings of the 8th International Conference on Compiler Construction (CC'99*, pages 168–182, 1999.

[34] V. Sarkar, N. Megiddo, I.B.M.T.J.W.R. Center, and Y. Heights. An analytical model for loop tiling and its solution. *Performance Analysis of Systems and Software, 2000. ISPASS. 2000 IEEE International Symposium on*, pages 146–153, 2000.

[35] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, NASA Ames Research Center, Aug 1990.

[36] M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *Proceedings of International Symposium on Code Generation and Optimization (CGO)*, pages 123–134, 2005.

[37] Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta optimization: Improving compiler heuristics with machine learning. In *Proceedings of the ACM SIGPLAN '03 Conference on Programming Language Design and Implementation*, pages 77–90. ACM Press, 2002.

[38] Xavier Vera, Jaume Abella, Antonio González, and Josep Llosa. Optimizing program locality through cmes and gas. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, page 68, Washington, DC, USA, 2003. IEEE Computer Society.

[39] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27. IEEE Computer Society, 1998.

[40] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[41] Jingling Xue. *Loop Tiling For Parallelism*. Kluwer Academic Publishers, 2000.

[42] K. Yotov, Xiaoming Li, Gang Ren, M. J. S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance BLAS? In *Proceedings of the IEEE*, 93:358–386, 2005.

[43] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Think globally, search locally. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 141–150, New York, NY, USA, 2005. ACM.