THESIS

AUTOMATIC CREATION OF TILE SIZE SELECTION MODELS USING NEURAL NETWORKS

Submitted by

Tomofumi Yuki

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2010

COLORADO STATE UNIVERSITY

December 04, 2009

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY TOMOFUMI YUKI ENTITLED AUTOMATIC CREATION OF TILE SIZE SELECTION MODELS USING NEURAL NETWORKS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate Work

Committee Member : Charles Anderson

Committee Member : Gretchen Irwin

Committee Member : Michelle Strout

Advisor : Sanjay Rajopadhye

Acting Department Head : Adele Howe

ABSTRACT OF THESIS

AUTOMATIC CREATION OF TILE SIZE SELECTION MODELS USING NEURAL NETWORKS

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 the tested architecture-compiler combinations.

Tomofumi Yuki
Department of Computer Science
Colorado State University
Fort Collins, CO 80523
Spring 2010

ACKNOWLEDGEMENTS

I am very fortunate to have Dr. Sanjay Rajopadhye as my advisor. I was fascinated by parallel computing and optimizations through his class that I took as an undergraduate student at CSU. Since then, I have learned so many things from him. For a number of times, he resolved my confusion with one or two words, which represented exactly what was missing in my understanding.

I would like to thank Dr. Charles Anderson and Gautam Gupta for their advice I received while we worked on a project that eventually became the seed of my thesis.

It was a wonderful experience working at IBM Research as an intern, where the core of my thesis was developed. I enjoyed and learned a lot from discussions I had with my mentor Dr. Lakshminarayanan Renganarayanan, other members of the Advanced Compiler Technologies group, and other researchers/interns I met during my internship.

I would like to thank Dr. Michelle Mills Strout and Dr. Gretchen Irwin not only for inspiring me through their courses, but also for being on my committee.

I would like to thank all my friends at CSU for sharing and discussing ideas, and also for distracting away from work when I needed some break.

I would like to thank my parents for supporting my study in a very distant place. I was very fortunate to have their understanding for studying in the other side of the earth.

TABLE OF CONTENTS

Chapter 1

Introduction

The compute and data intensive kernels of several important applications are loops. Tiling [22, 42, 28, 51] 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 thesis 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 highlighted in Figure 1.1, which illustrates a huge difference in performance between the best and worst tile sizes (5x on Power5 and 2.5x on Opteron). The performance is relatively flat in Power5, the performance of most kernels are close to the best between tile sizes 16 and 64. On the other hand, the performance varies at a much smaller change in tile sizes with Opteron. A difference of only 5 to 10 can make a huge impact on performance, and it is also visible in the figure that the optimal can be different for different kernels. TSS involves the development of a (cost) model that is used to characterize an analytical optimization problem to select the best tile sizes for a given combination of program, architecture, and compiler.

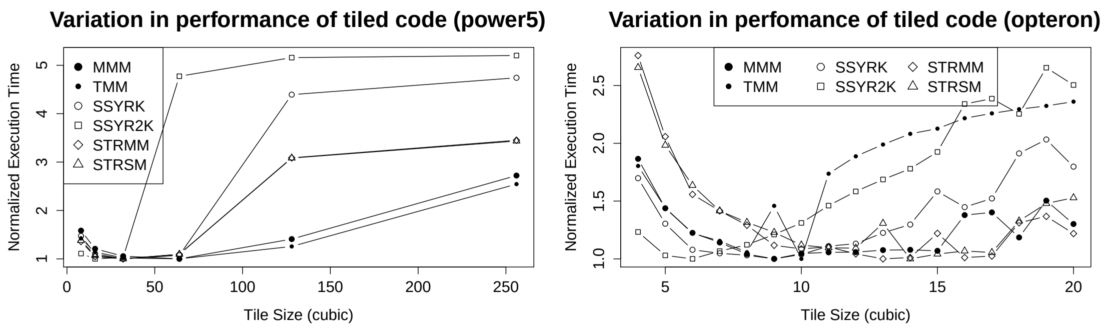


Figure 1.1: Variation in execution time of tiled code for six scientific kernels. The execution time is normalized to the best (lowest) running time of the points shown in the figure. Note that two processors show a very different behavior, and scale of x-axis is quite different in the two figures.

TSS solutions can be broadly classified into two categories, viz., purely models based, where models directly output estimated optimal [27, 42, 12, 21, 10, 41, 31, 52, 17, 39], and empirical search based [48, 15, 25, 11, 18, 36]. In the purely 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, (ii) the highly intertwined optimization phases of a compiler and (iii) rapidly changing architectures. For example, Yotov et al. [52, 53] 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 and maintaining accurate TSS models led to the wide-spread use of empirical search techniques. However, even empirical search techniques require TSS models to be efficient and fast enough 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 is becoming more and more difficult due to increasing complexity of modern hardware and software.

Previous approaches to TSS have used hand-crafted TSS models to either directly select the tile sizes [27, 42, 12, 21, 10, 41, 31, 52] or as a part of an empirical search to prune the search space [48, 15, 25, 11, 18, 36]. There are also approaches to TSS 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 [47, 29, 16, 35]. As discussed earlier, the hand-crafted models used in these approaches are difficult to create, adapt, and maintain.

The main question that we address in this thesis is the following. "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. We show for a specific class of programs, 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 a class of scientific computations that are known to benefit from cache tiling. We validate 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 requires 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 auto-tuner to guide a model-driven empirical search.

The key points in this thesis can be summarized as follows:

- 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 validation of our approach on six different compiler-architecture combinations. We show that very effective TSS models that predict the near-optimal tile sizes across all the six platforms can be automatically learned.

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

The remaining chapters are organized as follows. Chapter 2 introduces the necessary background including loop tiling for caches and the architectural features that affect the performance of tiled programs, and the neural network we used as our method of machine learning. Chapter 3 covers the related work, including previously presented TSS models and other instances of machine learning using in compiler optimizations. Chapter 4 introduces the program features, predicted outputs of our model and the different stages of our scheme. In Chapter 5 we present the experimental evaluation. Chapter 6 presents some conclusions and pointers to future work.

Chapter 2

Background

In this chapter, the necessary background for our work is introduced. Section 2.1 introduces loop tiling, and various aspects of programs and architectures that influence performance of tiled code. Section 2.2 briefly introduces artificial neural networks, a machine learning technique we use to learn TSS models.

2.1 Tiling

Tiling, also called blocking, transforms a set of loops into another set of loops, which perform the same computation but in a different order so that the program has better cache locality. Since it can be used to divide iteration space into smaller chunks, it is also used to expose coarser grained parallelism. In this section, we introduce tiling as well as the effects of some of the new hardware and compiler features on the behavior of tiled codes.

2.1.1 Class of Programs

In this thesis, we focus on a class of scientific computations, such as linear algebra, which are known to benefit from tiling. 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

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 2.1: Nine real kernels used for validation

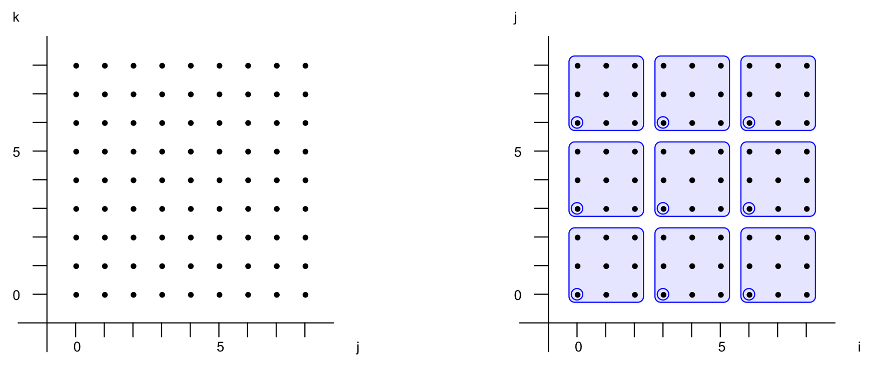
ordering or as a result of other transformations, such as fusing multiple kernel computations. This class is called Affine Control Loops (ACLs). ACLs have the following property:

- Loop bounds are defined as an affine function of surrounding loop indexes and parameters.
- Variables are accessed using affine function of loop indexes and parameters.

The nine kernels we use in this thesis, summarized in Table 2.1, are all ACLs. Several tools to generate tiled codes on this class of programs are available [37, 20, 5], which makes it easier to both explore and benefit from tiling.

Among this class of programs, we further restricted the programs to a subset that has 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 also limit our tiles to 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 size. We do not consider data padding or copy optimization.



```
//Original
for (i=0; i<9; i++)
  for (j=0; j<9; j++)
      ...
```

```
//Tiled
for (Ti=0; Ti<9; Ti+=3)
  for (Tj=0; Tj<9; Tj+=3)
    for (i=Ti; i < Ti+3; i++)
      for (j=Tj; j < Tj+3; j++)
        ...
```

Figure 2.1: Simple tiling example. 9x9 iteration space is tiled into 3x3 tiles. Figure generated by [40].

2.1.2 Simple Example

Figure 2.1 is an example of tiling, applied to a square iteration space. The original code with two nested loops that both go from 0 to 8 can be viewed as a collection of points in a 2D plane, where each point corresponds to an instance of some statement in the body being executed. In the original loop nest, all points in a column (points along the j-axis) are executed before the next column. In the tiled code, an additional set of loops are introduced to iterate over the set of points called tile origins (circled points in the figure). The inner loops now iterate over the points in a tile (3x3 squares in the figure), using the original order of execution. The legality of tiling itself and the legal execution order of tiles depends on the statement in the original code. Since all points in a tile are executed before later tiles, the order of execution is different. The size of the tile as well as the shape alters the execution order, and influence the performance.

```
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 2.2: SSYRK

2.1.3 Tiling for Locality

Tiling can be used to maximize reuse and avoid costly accesses to higher levels in the memory hierarchy. In the above simple example (Figure 2.1), if the statements writes to a 1D array, indexed by j, a tile only writes to 3 distinct memory locations, while the original computation writes to 9 locations when the same number of computation is performed. Thus, tiling can be used to control the memory footprint, and hence fit required memory into the target memory sub-system, such as cache or registers, to avoid high latency loads associated with storage.

Now we present an example from a dense linear algebra kernel, symmetric rank k update (SSYRK), shown in Figure 2.2. If you consider the access to the `C` matrix, the references do not change when i changes. If all elements of the `C` matrix remain in the cache between successive iterations of i, the main memory needs to be only accessed once for the entire duration of the program execution. 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.

Tiling partitions the iteration space and changes the order of execution while still performing the same computation, i.e. without changing the program semantics. A possible application of tiling is shown in Figure 2.3. New loops with indexes Tj and Tk control the memory requirement by the tile size parameter $tSize$ so that $tSize^2$ elements fit in the available cache.

```
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 2.3: Tiled SSYRK

Tiling has been studied extensively since it has been proposed over 20 years ago, and is used as one of the important optimizations in highly tuned linear algebra libraries such as BLAS or ATLAS [49, 53]. The above example only shows the use of tiling to maximize L1 cache reuse, but tiling can be applied to other memory hierarchies such as registers, and other levels of caches as well. In addition, it can be applied multiple times to optimize for multiple levels of the memory system.

2.1.4 Tiling for Parallelism

When the computation can be parallelized, tiling can be used to expose coarse grained parallelism. Once again, in the simple example (Figure 2.1), if the only dependence of a point in the iteration space was on the immediate south neighbor to compute, each column can be computed in parallel. It is also legal to compute each column of tiles in parallel, by letting a processor compute a tile instead of a point in the iteration space. By using tiles as the unit of computation performed on a processor, each processor has larger task, and hence coarser grained parallelism.

Coarser grained parallelism is often better when data transfer is involved. Communication start up cost is usually high in computational grids that use some implementation of the commonly used Message Passing Interface [44]. Thus, it makes sense to reduce the number of communications, in exchange for an increase in the volume. Tile size and shape influences the communication pattern, which in turn has a large effect on parallel performance.

Multiple levels of tiling can be used to exploit different levels of parallelism. Clusters of multi-core machines have parallelism both across machines and across cores within a machine, which can be utilized with additional levels of tiling. In addition, parallel programming for high performance would not make sense unless the sequential portion is also optimized, and tiling for locality should be done as another level of tiling.

In this thesis we restrict to sequential performance. Tile size selection models discussed here do not take parallel performance into account when selecting tile sizes. Extending this approach presented here to handle parallel performance is an important direction of future work.

2.1.5 Cache Size and Associativity

Over the past decade, caches in general propose processors have significantly changed. Caches have generally increased in size, and it is now common to have L2 caches that can store up to a few mega-bytes. Not only that, L2 or L3 caches are typically shared among multiple cores on a processor, which requires some type of cache coherency mechanism. Processors today do not have direct mapped caches anymore, but instead they use set-associative caches. In addition, other factors like the cache line size, cache evicting mechanism, or any other detailed design of the hardware can influence program behavior. It is difficult to reason how each one of these changes to the hardware affects programs, even more so when all of them are combined.

2.1.6 Tiling with Hardware Prefetching

In addition to caches discussed above, many modern processors now have some prefetching hardware to prefetch data from memory to caches. Prefetchers significantly change cache behavior. Hardware prefetchers keep track of memory access patterns, and fetches cache lines that are likely to be accessed in the future based on the pattern. For example, if cache line A, $A+1$ and $A+2$ are accessed in this order, a prefetcher can guess that $A+3$ is accessed next, and start fetching the data. The patterns that can be recognized by the prefetcher, the number of accesses (that follows some pattern) required to trigger prefetching, and the number of prefetching that can take place at a time depends on the hardware. The pattern used in the example given above is called unit-stride access, where successive cache lines are accessed with stride of one. Another pattern that are handled by some of the prefetchers are called constant-stride, where the stride can be some constant not necessary one. For example, given accesses to A, $A+10$ and $A+20$, constant-stride prefetcher can detect the pattern and start fetching $A+30$.

If hardware prefetchers can prefetch all required data before it is needed, tiling for locality is unnecessary. However, there are many cases where current hardware prefetchers cannot be used. In the following, the effect of hardware prefetching is discussed assuming a processor with hardware prefetchers that can only handle unit-stride accesses like Power5 or Opteron. On such a processor, all the references in Figure 2.2 can be prefetched, because all 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 matrix multiplication shown in Figure 2.4. In the innermost loop, references `C[i][j]` and `A[i][k]` are prefetcher friendly because successive references fit into the pattern of unit-stride access, and thus can be detected by the hardware prefetcher. However, reference `B[k][j]` is not prefetcher friendly because the first dimension of the reference changes before the second dimension. The access pattern would be B, $B+N$,

```
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 2.4: Matrix Multiplication

$B + 2N$, and so on, which cannot be detected by unit-stride prefetcher. Also, multiple computations may be fused in practice, which may result in a prefetcher-unfriendly code fragment. Again by loop fusion, the total number of references in a loop nest may increase beyond the number of prefetching that can take place at a time, which again limits the benefit of hardware prefetching.

We have assumed unit-stride prefetchers in the discussion above. There are processors that can detect constant-stride accesses, which would make all of the references in Figure 2.4 prefetcher friendly. 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).

Programs with the similar structure as shown above may still benefit of tiling for better locality for those references that cannot be prefetched. In some cases, simple loop permutation can be performed to make references prefetcher friendly. In the above example with matrix multiplication, j and k loop can be interchanged to make all references have unit-stride accesses. However, loop permutation cannot be applied if there are more complex dependencies that make loop permutation illegal. In addition, even for programs with loops that can be permuted, the loop may not get permuted due to other reason because tiling is not only used for locality.

One such example is illustrated in SSYRK kernel shown in Figure 2.2. Both j and k loops can be parallelized because the computation accumulates results using distinct memory location for each j, k pair (`C[j][k]`) from input array `A`. However, simply marking j and k loops to be parallel using would result in an inefficient parallel code. Because of the sequential outer loop i, synchronization is needed after each iteration of i. The current parallelization compute one step of the accumulation for all answers, before proceeding to the next step. Permuting the loops to make the i loop innermost would change the execution order so that a processor would complete the entire accumulation for an answer before moving on to the next. Since there are no sequential loops surrounding the parallel loops, the number of synchronization is now reduced from N to 1. However, the new loop ordering (j, k, i) makes the two references to array A prefetcher-unfriendly, and tiling could be used for better locality.

We have discussed how tiling for locality can still be beneficial even when hardware prefetching is present. However, there is a class of scientific computations where hardware prefetching is sufficient. It has been recently shown by Kamil et al. [23] that the traditional cache blocking may not be effective for stencil computations with 2D data. Stencil computations have uniform accesses that can be easily prefetched and 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, which is a very important scientific computation, from the target class of programs. However, stencil computations can still benefit from tiling through parallelism, 3D or higher dimensional data, and by tiling for other memory hierarchy such as memory.

2.2 Neural Networks

We have used artificial neural networks (ANN) as the machine learning technique for learning TSS models. ANN is a supervised learning method used to learn non-linear functions. Supervised learning methods require pairs of input and desired output, and learns some function that minimizes error between the output of the function and the desired output. The neural network we used was back-propagation neural network with Scaled Conjugate Gradient method [32]. We used neural networks with multiple layers, the first $n-1$ layers being the hidden layers, and the last one being the output layer. The number of nodes in the output layer is equal to the number of outputs to be given from the neural network. Each hidden layer can have any number of nodes. The inputs to the ANN are fed to the first hidden layer, and outputs from each layer are fed to the next layer. The outputs from hidden layers are nonlinear function (usually hyperbolic tangent) of the weighted sum of the inputs to that layer. The output layer performs the weighted sum of the outputs from the last hidden layer, but does not apply the hyperbolic tangent function.

Figure 2.5 is an example configuration of a neural network. Bias factor b (constant 1) is given as an input to all nodes to allow each node to learn constant offsets that are not related to any of the inputs. Each hidden and output node has its own weight vector w, which is updated during the training. The following is the equation for outputs of each hidden node, where w is the weight vector, i is the input vector from the previous layer, and N is the number of inputs.

$$\tanh\left(w_0 + \sum_{n=1}^{N} w_n i_n\right)$$

The output layer calculated using the same equation without the hyperbolic tangent function. Given a layer with N inputs and M nodes, the computation of the

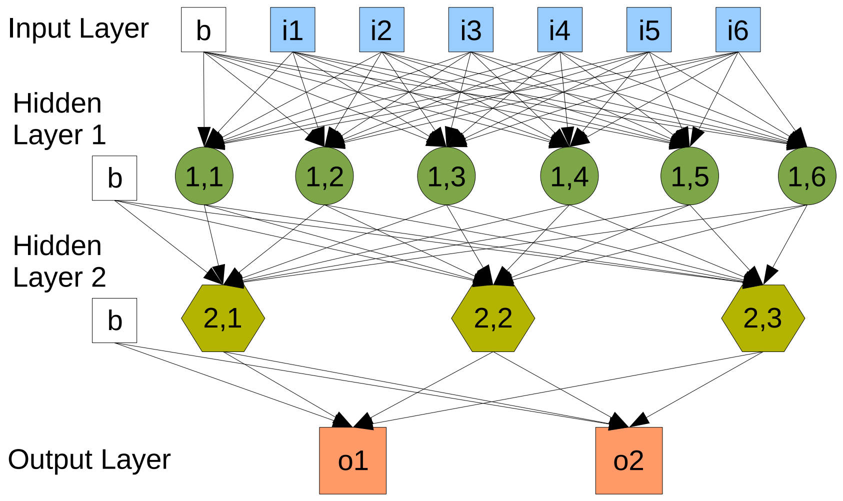


Figure 2.5: Diagram of a possible configuration of neural network with two hidden layers. Each node takes all outputs from the previous layer as input. In addition, bias factor b is given to all nodes. Outputs from a node is some form of weighted sum, and outputs of the neural network is the outputs of the output layer nodes.

outputs can be viewed as a matrix vector product of a vector of size N and $N \times M$ matrix. The error used for training E is computed using the following equation, where o_i and b_i are, respectively, the output of the output layer and desired output for training data i, and T is the number of training data used. Error E is computed for each output node separately.

$$E = \sum_{t=1}^{T}(o_t - b_t)$$

The neural network starts with random values as initial weights in each node, and then iteratively updates the weights to minimize the error. 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.

2.2.1 Neural Network for Classification

The error function can be changed to make neural network learn different functions. Neural networks can also be used for classification by adding another layer after the output and changing the error function. In classification, inputs are classified into one of the predefined classes. The additional layer is used to convert the output of the normal neural network described above to probability of an input being in each class. This is done through the following equation, where O is the outputs from the output layer (in the original NN) and Y is the new output.

$$Y_i = \frac{e^{O_i}}{\sum\limits_{n=1}^{N} e^{O_n}}$$

The new output Y_i is interpreted as the probability of input being in class i . The learning now maximizes the likelihood of a given input classified to the correct class.

2.2.2 Neural Network Parameters

There are many parameters used to configure neural networks. The number of hidden layers, and the number of hidden nodes in each layer defines the size of the neural network. Larger number of nodes tend to make the training faster, because wider range of weights are covered when the weights are initialized. Increasing the number of hidden layers can also make the training faster because more complex functions can be learned. The range of randomly initialized weights may significantly impact the learning as well. Because of the gradient method used in training, the weights will only be locally optimized, and if the initial values are far from the true optimal, it can never reach the true optimal.

The number of iterations to train and the condition to terminate learning are critical parameters to avoid over-fitting to the training data. Commonly used terminating conditions terminate the training when the error E during the training becomes lower than some threshold, and this threshold is the parameter.

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 a parameter.

Chapter 3

Related Work

The problem of finding good tile sizes to benefit from tiling, and models to find them have been studied extensively in the past. In this chapter, we cover such methods categorized into purely analytical models that directly predict the optimal, and mode-driven approaches where models are used as a part of iterative optimization. In addition, we cover another set of work that have used machine learning techniques in the context of compiler optimization.

3.1 Analytical Models

Many analytical models have been proposed in the past for tile size selection (TSS) [27, 42, 12, 21, 10, 41, 31, 17, 39]. 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 to 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.

3.1.1 Common Considerations

In the previous models, three types of cache misses are commonly considered. When a cache line is accessed, it will not be in the cache (unless it was prefetched), and results in an unavoidable cache miss called cold miss. The types of cache misses that can be avoided are those that happen when an element in the cache is evicted before the next access. One type of such cache miss, called capacity miss, is caused by programs that require more data than what would fit into the cache, before any reuse. Capacity misses are usually handled as an upper bound on the tile sizes so that the working set size is less than the cache capacity.

Another type is called conflict misses, where some element of the cache is evicted due to mapping conflicts. In a direct mapped cache, a cache line is mapped to a slot in the cache, and if another cache line with the same slot assigned comes in, the old cache line is evicted. Thus, in pathological cases, it is possible that a program that only access two cache lines continuously suffer from cache miss. Set associative caches assigns multiple slots to each cache line to remedy this problem, but it comes with the price of increased complexity of the hardware and higher latency to access caches.

In TSS models, conflict misses are further separated into self and cross interference. Self conflict misses are those conflicts that happen within an array. Cross conflict is when multiple variables (arrays) are involved in the conflict. Assuming each array is allocated contiguously in the memory, self conflict misses are easier to handle than cross conflicts. Most of the analytical models proposed in the literature take conflict misses into account, and try to minimize conflict misses while maximizing cache utilization.

3.1.2 Previous Models

Schreiber and Dongarra [42] presented algorithms to select tile shape as well as tile size. In their tile size selection algorithm, the performance was modeled using the ratio between computation and required memory. Using the amount of work per iteration and increase in memory requirement for each dimension of the loop nest, their model maximizes computation to memory requirement. Tile sizes were bounded by the amount of memory available. They considered capacity misses but conflict misses were not considered.

Lam et al. [27] have studied the performance of tiled matrix multiplication, and proposed an algorithm to find the optimal tile sizes for square tiles. They identified that problem sizes highly influence the optimal tile size, and used problem size as one of the inputs to the model. Their model considers utilization of cache as well as avoiding self conflict misses. In addition, they showed that copying arrays so that self conflict misses are avoided yields better and more stable performance. However, copy optimization requires more change in the code, and cost at run time to do the copying.

Esseghir [17] presented algorithms to compute rectangular tile sizes for a loop nest with one or more variables. In addition to capacity and self conflict misses their algorithm tries to minimize conflicts across variables. Conservative choice of relatively small self-conflict avoiding tiles are made while keeping all variables fit within the capacity to avoid cross conflicts.

Coleman and McKinley [12] presented an algorithm that computes rectangular tile sizes. They modeled cross interference rate (CIR) using memory footprints of array accesses and tried to minimize CIR as well as self interference. Their algorithm first tries if tile sizes with few columns of entire rows being a tile have a good utilization. If not, the row size is decreased, and different column sizes are explored. During the exploration, a new set of tile sizes is selected if the working set size is larger (better utilization) and CIR is lower (less cross conflicts).

Mitchell et al. [31] showed that when multi-level tiling is applied, applying TSS algorithm on each level independently may not be a good strategy. For an architecture with hierarchical memory with cache and TLB, they showed that optimizing for TLB and caches in concert yields better performance.

Chame and Moon [10] presented an algorithm that finds rectangular tiles that avoid self-interference as the first objective. From the set of tiles with no self conflicts, their algorithm minimizes the sum of capacity and cross interference misses. Cross conflict misses were modeled probabilistically as the ratio of memory footprint size and capacity.

Rivera and Tseng [38] showed padding and copy optimization can improve and stabilize performance given by previous tile size selection algorithms. Their approach manipulates data layout through padding and copying so that conflict misses are avoided. Since conflict misses are affected by the problem size, padding and copy optimization reduced the variation in performance with respect to the problem size.

Sarkar and Megiddo [41] presented a constant-time algorithm to find optimal tile sizes for doubly nested loops. They formulated a cost function as a function of problem size and effective cache size, and picked tile sizes with the lowest cost. Because they formulated the cost function as a quadratic equation, they were able to try all candidates for local minima in constant time.

The main weakness of these approaches is the static nature of analytical models. Analytical models are developed based on detailed analysis of the program and architecture. When a new factor like hardware prefetching comes in, models need to be updated through careful analysis of the new factor. This is very costly because it requires expert knowledge and detailed analysis. We address this problem in our work by automating the process of adapting to new factors.

3.2 Model-Driven Empirical Search

Another class of tile size selection techniques is model-driven empirical search [48, 15, 25, 11, 18, 36, 24, 47, 35, 52]. Purely empirical tuning with global search may be a feasible solution for optimizing libraries of commonly used kernels [48], 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.

ATLAS [48] is a auto-tuner for linear algebra kernels, which performs a global search over many different performance tuning parameters. The large cost associated with the search was accepted because the kernels were used in many programs. Yotov et al. [52] later added heuristics to first prune the search space so that costly global search is avoided. For tile size selection, their model assumes fully associative cache and tries to fit the working set size within the cache capacity. Since ATLAS use square tile sizes, their model also predicts the best square tile sizes according to their cost function. Epshteyn et al. [16] used curve regression to guide the empirical search from a starting tile size selected by the model by Yotov et al. [52]. The next tile size to try was decided based on how much information would an experiment give to help the regression.

Kisuki et al. [24] present an iterative compilation strategy for optimizing both tile size and unroll factors. They show that significant improvement can be achieved, but getting maximum speedup requires around an hour of compilation time. They have tried multiple searching methods including genetic algorithms (GA) and stimulated annealing. No model was used to prune the search space in their work.

Vera et al. [47] present a method to optimize tiled codes by selecting tile sizes and by padding. They formulate a cost function of loops using Cache Miss Equations [19]. The cost function is then searched by GA for the optimal. Their work is not an empirical search since they do not execute any program while searching the cost function. However, we consider their work to be a model-driven search for TSS, which is closely related to our work. Fraguela et al. [18] present a similar method that use Probabilistic Miss Equations combined with GA to search for the optimal without actually running any code. Both of these methods have significantly lower cost of searching, typically within several seconds. The primary reason for the low cost is that their search involves evaluation of a function, not compilation, execution, and timing of programs.

Parsa and Lotfi [35] presents a method to optimize tiled codes by selecting tile shapes along with tile sizes. They also use GA to search a complex cost function. The cost function consists of multiple sub functions, modeling I/O, loads to memory, communication costs, and memory requirements.

Knijnenburg et al. [25] have used the static model proposed by Coleman and McKinley [12] as part of their model-driven search. They show that iterative optimization can give significantly better performance compared to static approaches, but the number of iterations can be reduced by using static models. Static models are used to rank candidate tile sizes and unroll factors, and then a number of highly ranked candidates are actually executed to find the optimal.

Chen et al. [11] present a method for optimizing dense linear kernels to multiple levels of tiling using model-driven search. They optimize each level independently, carrying over decisions made in earlier levels. They prune the search space to tile sizes that occupy a certain percentage of the cache (based on set associativity), and empirically search the pruned search space.

Qasem and Kennedy [36] present an iterative compilation method for tiling combined with loop fusion. They use a cost model to first find the starting point for the search, and then the tolerance given to the cost model is increased until the empirical performance given degrades. Tolerance is a parameter that changes the behavior of the cost function, the estimated probability of conflict misses in the case of tiling.

Model-driven empirical search gives good performance in exchange with longer compilation time. These techniques cannot be used when long compilation times are not desired. The models created by our work only takes fractions of a second to use so that compilation time would not become an issue.

3.3 Machine Leaning Techniques in Compilers

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 performance of modern complex architectures. The wide range of applications include branch prediction [6], instruction scheduling within basic blocks [34, 30], and deciding if certain optimization should be applied [8, 9, 46, 33]. Some of the empirical search methods discussed above have used some form of machine learning to guide the empirical search [47, 16, 35]. In this thesis, we use machine learning techniques to automatically learn TSS models. There are a few cases where machine learning techniques have been applied to TSS of some form [47, 29, 16, 35, 45].

Stephenson and Amarasinghe [45] used classifiers to predict best loop unroll factors. The classifiers were trained with two machine learning techniques; near neighbor classification and support vector machines. Features extracted from the programs are classified into eight classes, corresponding to unroll factors one through eight. We share many things in common with their work, and unroll and jam may be considered as a level of tiling for the registers. However, one key difference between their work and ours is that they were able to use classifiers because possible unroll factors are much smaller than possible tile sizes. The amount of data that fits in registers are much fewer than that for the cache, resulting in much smaller search space. As a result, this approach cannot be used for TSS.

Li and Garzaràn [29] used learning classifier system (LCS) to construct models for selecting tile sizes and number of tiling levels for matrix-matrix multiplication (MMM). LCS is a machine learning technique that combines genetic algorithms and reinforcement learning for constructing rules. They trained the LCS by running MMM with different problem sizes and tile sizes, and the learned LCS was specific to MMM. Our approach learns a model that can be used for a range of programs, although we only consider one level of tiling.

Moss et al. [34], used supervised learning methods, including neural networks, to schedule instructions within basic blocks. Supervised learning methods were able to schedule basic blocks well, but was limited by the number of instructions that can be in a basic block. This limitation came from the use of supervised learning methods, because desired outputs to an input must be known for supervised learning methods to be applied. McGovern and Moss [30] later used reinforcement learning methods to overcome this limitation.

Calder et al. [6], used neural networks and decision trees for static branch prediction. Static features associated with programs were mapped to prediction of the branch. Their branch prediction performed better than previously known heuristics.

Stephenson et al. [46], used genetic programming to create heuristics that are used in compilers to decide what optimizations should be applied. They addressed hyper block formation, data prefetching, and register allocation. The heuristics they found were as good as or better than those designed by hand, but selecting features from the program and genetic programming parameters still required some human intervention.

Monsifrot et al. [33], used decision trees to determine whether a loop should be unrolled or not. Unrolling based on decision trees with five features performed better than compiler heuristics, but it is unclear how they chose the unroll factor because their model can only tell if it is beneficial to unroll the loop or not.

Cavazos and Moss [8] used a supervised learning method called rule set induction, to decide if instruction scheduling should be performed based on features of the basic block. They found that many blocks do not benefit from instruction scheduling, and avoiding unnecessary optimizations were important especially in just-in-time compilers. Using their induced heuristics, the cost of instruction scheduling was reduced to less than 25%, while maintaining most of the benefit from scheduling all the blocks.

Cavazos and O'Boyle [9] also used machine learning methods in dynamic compilation. They used logistic regression to train a heuristic that was then used to select the best set of optimizations for each method in a program, based on features of the method. With a trained heuristic, execution times for SPECjvm98 and DaCapo+ benchmarks were reduced by 25% and 51% respectively.

Another set of applications is in the field of embedded systems, used to efficiently search for optimal order of optimizations [7, 3, 14, 26]. The order of optimizations applied to a code fragment can largely affect code size and speed, and embedded software designers are willing to tolerate longer compilation time, since software for embedded systems is often used in large number of units with severe constraints on space, speed and energy consumption. Even though longer compilation times were acceptable, the search space of optimization sequences was too large to search exhaustively, and machine learning methods and genetic algorithms were used to reduce the search space.

3.4 Summary

Models for TSS model various causes for cache misses to predict optimal tile sizes. Purely analytical models are developed through detailed study of hardware and software that requires significant effort of experts. Empirical search methods executes the program multiple times to find a good tile size, which often gives better performance than simply using predicted tile sizes from a model. However, it takes time at compile time since the program is actually executed. Many of the empirical search methods use some kind of heuristic, analytical models in some cases, to reduce the time it takes to empirically search for good tile sizes. Our approach can be used to replace analytical models to adapt to changes to the hardware, software, or other changes in the environment, without much human effort.

Chapter 4

Creating TSS Models

In this Chapter we describe our TSS models and our approach to automatically generating such models for different environments. Section 4.1 describes our model, and Section 4.2 describes our approach to create instances of our model for an environment.

4.1 Target TSS Model

In this section we describe how we formulate the inputs and outputs for our TSS model, and the range of programs targeted by our model. The class of programs handled by our model was previously described in Section 2.1.1.

4.1.1 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 [45, 33, 9, 3]. After experimenting with a variety of features that capture the spatial and temporal locality effects of loop tiling, we arrived at a simple set of six features.

The features we use are the number of references in the innermost statements, classified into three different types of references, and each type of reference is further classified into reads and writes. 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 the L2 cache is commonly shared among the cores.

The following is an example using matrix multiplication show in in Figure 2.4, assuming row-major layout and unit-stride prefetcher. 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. These features can be easily extracted by looking at the loop orderings and indexes used to reference arrays. The compiler needs to be aware of what type of hardware prefetcher is used on each architecture to calculate these values, but we believe this is a simple requirement.

4.1.2 Importance of Simple Program Features

In previous work, it was common to use a large number of program features (sometimes up to 60). This was because it is relatively easy for a compiler to collect a number of syntactic information from the code, if not already available. Many of the features can be useless for prediction, and some effort has been made towards selecting the useful features and reducing the total number of features used [13, 3]. Large number of program features can cause the training to take longer than necessary, and using the resulting model will also take longer.

We initially started with a large number of program features, but noticed that some of the features were either not being useful or could be computed as some combination of other program features. Because of the multi-layered neural networks, inputs that are simple combination of other inputs can be easily learned in some form, and thus those inputs features turned out to be redundant.

4.1.3 Possible Predicted Outputs

There are multiple possible target outputs for a TSS model. The desired output given to the neural network during training becomes the output from the model. This makes it very easy to change the output if necessary.

The initial model we tried have used execution times of training program instances as the desired output, using problem size and tile sizes as additional inputs along with the program features described above. The trained model now predicts the expected execution time for a set of program features. However, this approach requires searching the function after modeling. We need to find the tile sizes that minimize the function for a given program feature and problem size. Finding the optimal of the function may be difficult depending on the type of function being learned. 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, one that we would like to avoid.

Another possible target is the optimal tile size itself. Directly predicting the best unroll factor through classification has previously shown to be successful [45]. 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 using a different formulation of error or other machine learning techniques such as support vector machines. However, classification can only partition the input into those classes that were observed during training. 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.

4.1.4 Output of Our Model

We used a solution between the two ways described above. We use the optimal tile size as the target, 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) to the optimal tile sizes. Learning as a function allows the model to predict tile sizes that was not seen during training, and by directly predicting the tile sizes, the potentially expensive search step is avoided.

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 non-prefetched read references. Program `A` has optimal tile size of 100 with one non-prefetched reference, program `B` has optimal tile size of 50 with three non-prefetched references. Since an increase in non-prefetched references implies an increase in memory footprint, it is intuitive that the optimal tile size is smaller. It is also reasonable to think that another program `C` with two non-prefetched references (other features identical) to have the optimal tile size between 100 and 50. With a classifier, the new program `C` would be classified to have the optimal tile size 50 or 100, when programs `A` and `B` were the only training data used. With a simple line fit, a function would say

75 is the predicted optimal for C, which is likely to be closer than both 50 or 100. Functions learned by neural networks would behave similarly, but the function would be much more complex than a simple linear line fit.

4.1.5 Using ANN to Learn TSS Model

We used artificial neural networks (ANN), a supervised learning method, to learn tile size prediction model previously described in Section 2.2. Supervised learning methods require pairs of input and desired output, and learn some function that minimizes the error between the output of the function and the desired output. 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 an integer and use that as the predicted optimal tile size.

4.2 Learning TSS Models

Given the target model described above, we use ANN to learn a function from inputs to outputs. 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.

4.2.1 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 applications available at the time of training. The neural network cannot be expected to perform well on programs with program features that are largely 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 over-trained 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, HiTLOG [37], 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 of synthetic programs are one of the main reasons that lead to good performance of our 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 difficult to try a large range of possible programs with synthetic programs. Even

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

Table 4.1: 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 4.2: Data points used for training

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 programs that covers all the kernels we used, but also includes many others so that the model is not specialized to just those kernels. Table 4.1 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 (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 2835 program instances in this space with at least one write.

From the bounded feature space, we collected optimal tile sizes for a number of points in the feature space. Table 4.2 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.

4.2.2 Data Collection

Collecting training data is time consuming, but we do not need much human effort for this. Analytical models require extensive case study to develop, and also require large amount of experiment to be run. The use of parametrized tiled code generator [37] helped our data collection by avoiding re-compilation for different tile sizes of the same program. The data collection was done through a set of script that generate synthetic programs, and measure their execution time with different tile sizes.

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. Since programs with more references also have more operations, it generally takes longer to execute. In order to avoid unnecessarily long execution times, a simple equation was used to adjust the problem size based on number of references. The problem sizes were generated using the following equation with $BASE$ configured differently for each architecture according to their computational power. The function `int` in the equations is a function to round down a given value to an integer.

$$BASE - 250(\texttt{int}((RP + WP + 2(RNP + WNP))/4))$$

Non-prefetched references are weighted more to account for increase in memory loads. This equation does not give problem sizes that are precisely around the target execution time, but all we wanted was a method to prevent data collection to take too long or too short.

4.2.3 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 larger range of programs, and having detailed training data only helps learning.

The parameters of the neural network are not as simple as previously described in 2.2.2. 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. Future work includes developing 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, initial weights 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 values are from the desired output.

It took about one to 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 combinations. 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. 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.

4.2.4 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 optimal tile sizes, or as a part of a model-driven search method to find the optimal tile size. The first step is to extract the program features discussed previously. This step should not take much time due to the simplicity of our program features. Then the program features are used as an input to the learned model to produce 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, neighboring 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 strategy for mode-driven search.

The use of neural networks is computationally close to two matrix-vector products of size 31×6, which is trivial with modern compute power. 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.

Chapter 5

Performance Evaluation

We have used our approach to learn tile size selection models on six architecture-compiler combinations summarized in Table 5.1.

Feature extraction was done manually, but it can be easily automated by a compiler. The compiler needs to know if the target architecture has a hardware prefetcher, and look at the order of surrounding loop indexes to figure out the type of reference for each array access. The list of features extracted from the kernels are shown in Table 5.2. We used the same set of program features on all architectures, because references that can be prefetched were the same across all processors (recall discussion in Section 2.1.6).

The problem sizes were selected to run for about 60 seconds to make sure the program runs long enough, and to ensure that the problem sizes are sufficiently different from training runs, which targeted around 10 seconds of execution.

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

Table 5.1: Architecture and compilers used

	RP	RNP	RI	WP	WNP0	WI
MMM	1	1	0	0	0	0
TMM	1	1	0	0	0	0
SSYRK	0	2	0	0	0	0
SSYR2K	0	4	0	0	0	0
STRMM	0	1	1	0	1	1
STRSM	0	1	1	0	1	1
LUD	0	1	1	0	1	1
SSYMM	0	1	2	0	2	2
TRISOLV	1	1	0	0	0	0

Table 5.2: Program features of kernels used for evaluation.

5.1 Performance

For each architecture-compiler combination, we compared the execution time of kernels with the true optimal, using cubic tile sizes, found by exhaustive search. Figure 5.1 shows the normalized execution time of nine kernels for tile sizes selected by machine-crafted models learned for each architecture-compiler combination. The performance for tile sizes predicted by our 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 1.1). This supports our claim that TSS models that perform well can be learned from simple features and training data collected through synthetic programs for different architecture and compilers. We have shown that our model can adapt to different compilers, which is just as hard if not harder compared to different versions of the same compiler. This indicates that the learned TSS models can be easily updated (re-trained) with respect to the evolution of a single compiler.

In the remainder of this section, we discuss the possible causes of inaccuracies and counter intuitive behaviors that lead to slowdowns shown in the figure.

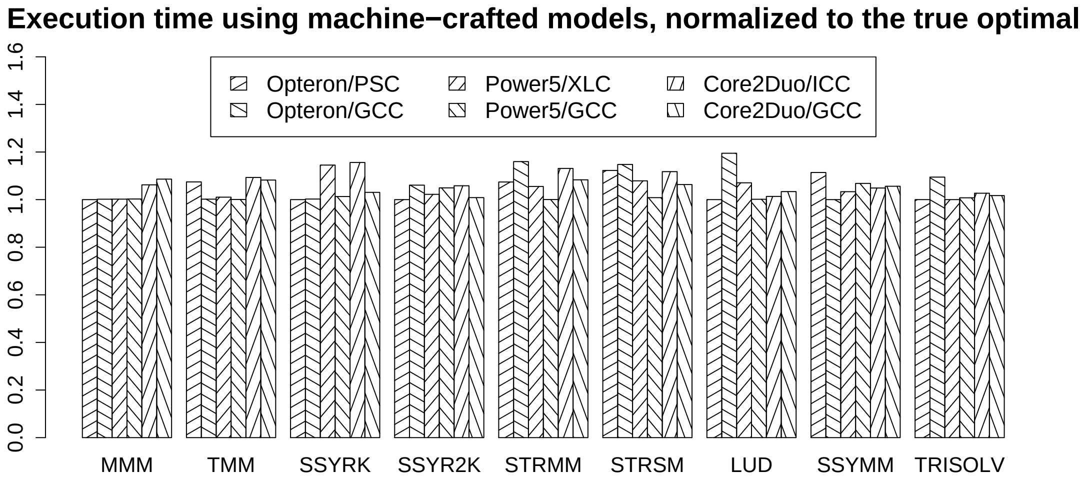


Figure 5.1: Execution times, normalized to the best tile size determined through exhaustive search, of kernels with tile sizes selected by machine-crafted models for each combination of architecture-compilers.

5.1.1 Program Features

Since MMM and TMM have the same statement with different bounds on the loops, the program features are identical, and thus the predicted tile sizes are the same. However, the optimal tile sizes found for MMM and TMM were different. STRMM, STRSM and LUD have identical program features, but their iteration space is also different. In addition, these programs have statements in loops that are not innermost, which is ignored when extracting program features. All of the above are possible sources of miss prediction, but trying to capture all of them would result in large number of program features. We made the decision to have a small number of program features for other advantages discussed previously in Section 4.1.

5.1.2 Sensitivity to Change in Tile Size

Through the exhaustive search process, we observed that Power5 and Core2Duo are significantly less sensitive to changes in tile size. As previously seen in Figure 1.1, performance on Opteron significantly varies in a relatively small changes in tile sizes compared to Power5. Core2Duo and Power5 shared the similar pattern of having a relatively wide range of good tile sizes, and the performance start degrading at a much larger tile compared to Opteron. Since Core2Duo had wider range of good tile sizes compared to Power5, we consider Core2Duo to be least sensitive, and Opteron to be the most sensitive to changes in tile sizes among the processors we used. We suspect that the set associativity is one of the reasons for Power5 and Core2Duo being less sensitive to changes in tile size. Higher set associativity reduces the probability of conflict misses that has been known to be one of many factors that complicates tile size selection.

It is easier to get good performance when it is not sensitive to changes in tile size. However, low sensitivity can lead to seemingly inaccurate predictions. During the automated data collection, the optimal tile size for each synthetic program instance is collected. When the performance is very flat, the optimal found through this process can significantly vary due to some noise during execution. We believe that this noise can be suppressed by increasing the problem size and/or running each instance of synthetic programs used for training a number of times and taking the minimal (or mean) execution time. Some way of detecting flatness in the performance may help avoiding this behavior. However, we have not used any of the above since the performance was still close to the optimal without such extension.

	PSC-Optimal	PSC-Predicted	GCC-Optimal	GCC-Predicted
MMM	8	8	8	8
TMM	10	8	8	8
SSYRK	9	9	8	7
SSYR2K	6	6	6	9
STRMM	13	11	8	11
STRSM	17	11	8	11
LUD	11	11	8	11
SSYMM	10	8	8	8
TRISOLV	8	8	8	8

Table 5.3: Predicted and optimal tile sizes for Opteron

5.1.3 Opteron

The AMD Opteron has the largest cache size, but smallest set associativity among the three processors we used. The optimal tile sizes found for this processor is relatively small, and small changes in tile sizes can have large effect on performance (recall Figure 1.1). Table 5.3 shows the optimal and predicted tile sizes for both Path Scale Compiler (PSC) and GCC for Opteron.

With PSC, we see that MMM and TMM that have identical program features have different optimal tile sizes. Similar behavior is observed for STRMM, STRSM, and LUD that have identical features. This is due to our program features being incomplete as discussed previously in Section 5.1.1. Small miss prediction that may have been caused due to incomplete program features have resulted in slowdowns of around 10%.

The optimal tile sizes given by GCC is surprisingly flat at 8, except for TMM. Miss predictions on LUD had shown the most slowdown among all architectures (20%). STRMM, STRSM and LUD have statements in the second loop, which could make the program behave differently from other programs with identical features, but with perfectly nested loops. We suspect this to be due to GCC behaving differently when imperfectly nested loops are present, but we are not

	XLC-Optimal	XLC-Predicted	GCC-Optimal	GCC-Predicted
MMM	113	105	108	105
TMM	109	105	106	105
SSYRK	49	56	58	51
SSYR2K	26	25	29	25
STRMM	49	52	48	54
STRSM	57	52	57	54
LUD	45	52	47	54
SSYMM	36	24	41	26
TRISOLV	105	105	108	105

Table 5.4: Predicted and optimal tile sizes for Power5

sure about the exact cause.

5.1.4 Power5

The IBM Power5 is the processor has smaller cache and medium set associativity among the three processors we used. The optimal and predicted tile sizes found for Power5 are shown in Table 5.4. The optimal tile sizes found are much larger compared to that of the Opteron. The predicted tile sizes are 5 to 10 away from the optimal. Despite this, the actual performance hit was not much, since the performance is less sensitive to smaller changes. For Power5, both XLC and GCC produced similar results.

5.1.5 Core2Duo

L1 cache of the Intel Core2Duo has the same size but higher set associativity compared to Power5. Core2Duo is also the only processor with constant stride prefetching among the three processors. The optimal and predicted tile sizes for Core2Duo are shown in Table 5.5. The predicted tile sizes are actually significantly off from what was found to be the optimal. Again, as seen in Figure 5.1, the performance given by predicted tiles are close to the optimal. This is because the Core2Duo is even less sensitive to changes in tile sizes as discussed above.

	ICC-Optimal	ICC-Predicted	GCC-Optimal	GCC-Predicted
MMM	137	95	16	61
TMM	145	95	16	61
SSYRK	48	76	20	44
SSYR2K	32	24	48	32
STRMM	40	63	48	78
STRSM	40	63	52	78
LUD	56	63	64	78
SSYMM	48	23	48	23
TRISOLV	44	95	42	61

Table 5.5: Predicted and optimal tile sizes for Core2Duo

GCC showed counter-intuitive behavior on Core2Duo. The optimal tile sizes for MMM and TMM is considerably smaller than other programs with larger memory footprint. We have tried multiple problem sizes, but the result was similar. We do not have a good explanation for this behavior. One conjecture we can make is that since matrix multiplication is a very well studied program, GCC may have applied more aggressive optimization and changed its program behavior. The conjecture was strengthened by experimenting with the optimization level. Changing the compiler option from -O3 to -O2 caused the tile size predicted by our model to show identical performance as the true optimal.

5.2 Performance with Local Search

The focus of this thesis is learning TSS models that can predict good tile sizes for different architectures 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.

Figure 5.2 shows the normalized execution time of each kernel using the best tile size within a certain distance of the predicted tile size. Table 5.6 shows the mean

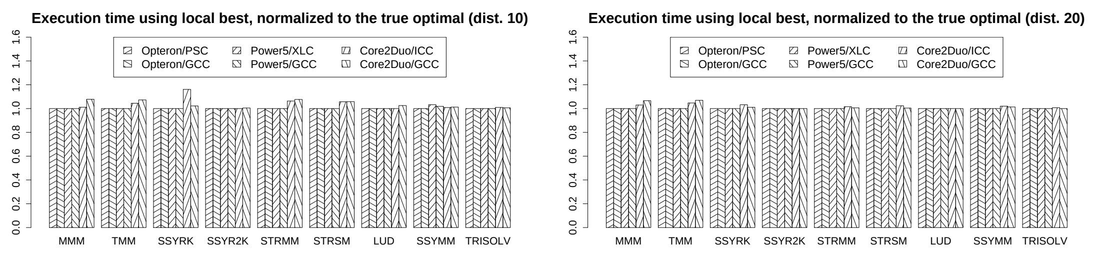


Figure 5.2: Normalized execution time using the best tile sizes found within distance 10 and 20.

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

slowdown over all nine kernels when the best tile size within a certain distance of the predicted tile size were used. By searching immediate neighborhood 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 that the cause of relatively small improvement on Core2Duo is due to the very high set-associativity (8-way). As discussed previously in Section 5.1.2, there is a very wide range of good tile sizes, and the automated training data collection 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.

Even a naive local search around the tile sizes predicted by the machine-crafted

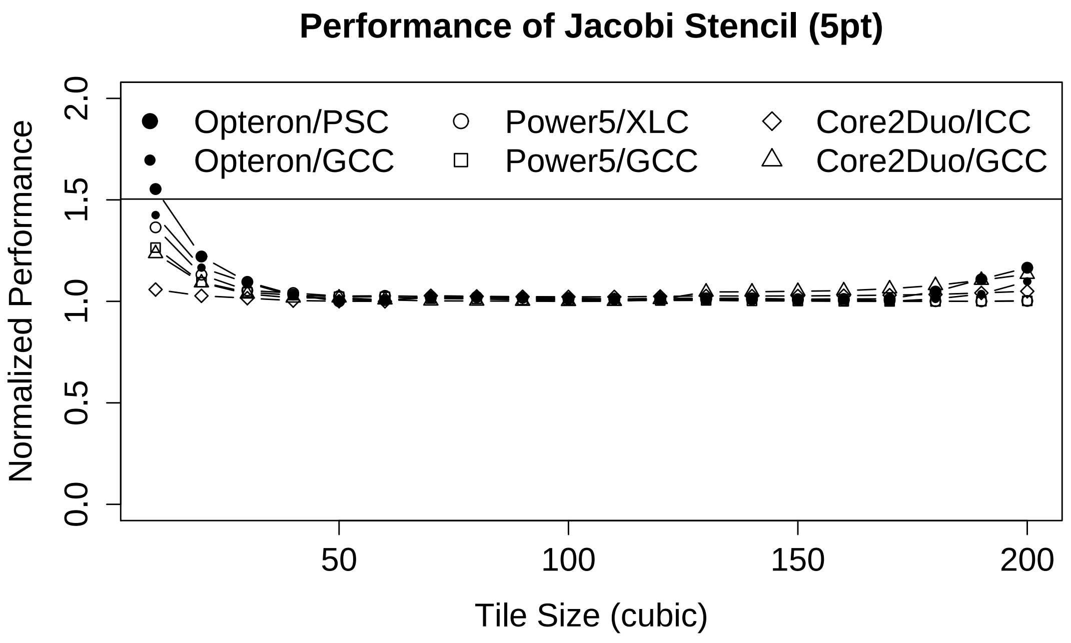


Figure 5.3: Performance of Jacobi Stencil on each architecture-compiler combination. The performance is flat except for very small tile sizes.

models show significant improvements. This demonstrates the potential of our method to perform better when combined with the more sophisticated search methods proposed in the literature.

5.3 Performance of Stencil

It has been previously discussed in Section 2.1.6 that stencil computations with 2D data may not benefit from tiling for L1 cache. We briefly show that it is true on all architecture-compiler combinations we use for evaluation. Figure 5.3 shows the performance of 5 point Jacobi stencil on the six architecture-compiler combinations. The performance is flat except for very small tile sizes and very large tile sizes.

The performance of other kernels on Opteron is a strong evidence that these performance drops are not due to L1 cache misses. Recall Figure 1.1 and the discussion in Section 5.1.3 about the performance of Opteron. The optimal tile sizes were around 5 to 15, and the performance significantly degraded if the tile sizes used were more than a few tile sizes away from the optimal. The performance of Jacobi on Opteron is flat like the other two architectures, illustrating the difference between Jacobi and other kernels.

The performance drop with small tile sizes can be explained by the loop overhead of tiling. When tile sizes are small, the number of tiles increase, and thus increasing the loop overhead of tile loops. Loop overhead has much less effect on tile size compared to cache misses, but when all references are prefetched, it becomes the main source of performance hit.

The performance drop with large tile sizes comes from other memory hierarchy such as L2 cache or main memory. Jacobi stencil requires 2D data array of size tile size squared, and 150^2 doubles or 176KB of data largely exceeds the cache capacity of any of the architecture. The memory requirement is actually twice because Jacobi uses two copies of the array since the result of the previous iteration is used to compute the current. Thus the performance drop cannot be due to L1 cache misses.

We have confirmed that stencil computations do not benefit from tiling for L1 cache. However, we would like to emphasize that tiling can still improve the performance of stencils by tiling for other memory hierarchy or for parallelism.

5.4 Comparison with Hand-Crafted Models

Many static models previously developed to maximize performance of tiled code are analyzed in detail by Hsu and Kremer [21]. We have taken two of those models that were reported to perform well and compared the performance given by these models and our machine-crafted models. Although these two models are from 1991 and 1995, these models are still commonly used for tiling without copy optimization or data padding. We first briefly describe the logic behind the two models, LRW [27] and EUC [12].

LRW 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. Their algorithm finds the largest square tile that avoids self conflicts. EUC is also an analytical model with similar considerations used as in LRW, but it predicts rectangular tile sizes instead of square. EUC uses Euclidean algorithm [2] to compute candidate heights of tiles. EUC takes cross-interference into account by estimating Cross Inference Rate (CIR) from the memory footprint. In each iteration of their algorithm, a new tile size is selected over the previous one when it has better cache utilization (larger working set) and smaller CIR. Both of these models take problem sizes as an input, whereas our model does not.

5.4.1 Tailoring Hand-Crafted Models to Modern Architectures

We made a small but necessary modification to the hand-crafted models to adapt to the current architecture. Since hand-crafted models were developed when hardware prefetchers were not commonly available, they treat all references as non-prefetched. However, it is obvious that prefetched references do not need 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 prefetch into account, we modified their models so that prefetched references are excluded from calculation. Further, for programs that has more than one non-prefetched references, we give smaller cache

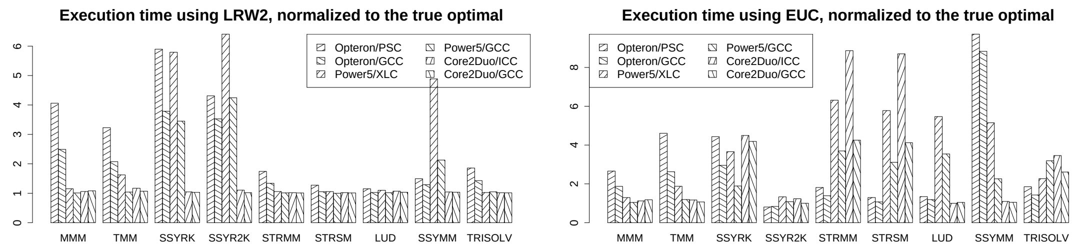


Figure 5.4: Execution time of kernels with tile sizes selected by hand-crafted models for each combination of architecture-compilers, normalized to the optimal.

sizes to the model. We also incorporated into LRW the extensions described by the authors to handle set-associativity [27]. This extension will select larger tile sizes for set-associative caches when compared to assuming direct mapped cache.

5.4.2 Performance of Hand-Crafted Models

Figure 5.4 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 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. Performance of tile sizes given by LRW is relatively worse on Opteron compared to the other architectures. Opteron is more sensitive to small changes in tile size, as discussed previously in Section 5.1.2, which makes predicting the optimal more difficult. LRW performs 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 because EUC is not restricted to square/cubic tile sizes, it 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 $6 \times 6 \times 6$, but it was not the optimal when rectangular

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

Table 5.7: Mean of slowdowns with tile sizes predicted by each model over all nine kernels on each architecture-compiler combination

tiles were considered. EUC was able to find a very thin tile with better cache utilization for this case.

Table 5.7 summarizes the effectiveness of each model by showing the percentage of slowdown when compared to the optimal using cubic tile sizes. LRW shows comparable performance on Core2Duo, but overall the machine-crafted model provides consistently near-optimal performance across all architectures. We believe the reason for LRW showing comparable performance on Core2Duo is also due to the fact that Core2Duo has a very wide range of good tile sizes, as previously discussed in section 5.2.

Chapter 6

Conclusions and Future Work

Tile size selection is an important step in the profitable use of loop tiling. Handcrafting 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. Another direct extension is to construct TSS models where tiling is used to expose coarse-grain parallelism [5]. 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 autotuners. 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 an interesting direction of future work.

The use of synthetic programs for data collection is a strength in our approach. However, it may become a limitation when our approach is extended to more general models. If it takes large number of features to capture the performance characteristics of programs with respect to optimization parameters, the space of possible program instances also increases. This may become a challenge when applying our approach to other optimizations or more complex tiling.

Tuning of the neural network parameters is the only part of our approach that is not automated. Even though the tuning did not take very long in our case, we would like to automate this process if possible. Since finding optimal neural network parameters is an important sub-problem when using neural networks, methods for finding optimal configurations has been extensively studied [50]. Integrating such method to our approach would make the model creation completely automated.

Although we have shown that we can create models that predict near-optimal tile sizes, one may criticize our work for not providing any feedback about the underlining architecture. Models learned by neural network cannot easily provide the insight about the architecture. Many statistical learning techniques suffer from the same problem, and are sometimes referred to as 'black boxes' [43]. Because artificial neural networks were often criticized for its black box nature, significant effort has been put towards understanding and extracting what ANN has learned during training [4]. Appropriate use of these techniques to extract more information about the underlining architecture is another important direction of future work.

REFERENCES

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

[2] *Neal Koblitz. Graduate Texts in Mathematics.* Springer-Verlag, New York, 1987.

[3] F. Agakov, E. Bonilla, J. Cavazos, B. Franke, Fursin G., M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using machine learning to focus iterative optimization. pages 295–305, 2006.

[4] R. Andrews, J. Diederich, and AB Tickle. Survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8(6):373–389, 1995.

[5] 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.

[6] 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.

[7] J. Cavazos, C. Dubach, F. Agakov, E. Bonilla, M.F.P. O'Boyle, G. Fursin, and O. Temam. Automatic performance model construction for the fast software exploration of new hardware designs. *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 24–34, 2006.

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

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

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

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

[11] 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.

[12] 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.

[13] Katherine E. Coons, Behnam Robatmili, Matthew E. Taylor, Bertrand A. Maher, Doug Burger, and Kathryn S. McKinley. Feature selection and policy optimization for distributed instruction placement using reinforcement learning. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 32–42, New York, NY, USA, 2008. ACM.

[14] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. pages 1–9, 1999.

[15] 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. *Proceedings of the IEEE*, 93(2):293, 2005.

[16] 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 *In Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*, pages 259–273, 2005.

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

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

[19] Somnath Ghosh, Margaret Martonosi, and Sharad Malik. Cache miss equations: A compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems*, 21:703–746, 1998.

[20] 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.

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

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

[23] 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.

[24] 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.

[25] 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.

[26] P. Kulkarni, W. Zhao, H. Moon, K. Cho, D. Whalley, J. Davidson, M. Bailey, Y. Paek, and K. Gallivan. Finding effective optimization phase sequences. *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 12–23, 2003.

[27] 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.

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

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

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

[31] 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.

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

[33] 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.

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

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

[36] 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.

[37] 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.

[38] Gabriel Rivera and Chau-Wen Tseng. Locality optimizations for multi-level caches. In *Supercomputing '99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 2, New York, NY, USA, 1999. ACM.

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

[40] Jonathan Roelofs. Tiling visualizer. http://www.cs.colostate.edu/ roelofs/, Nov. 2008.

[41] 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.

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

[43] Jonas Sjöberg, Qinghua Zhang, Lennart Ljung, Albert Benveniste, Bernard Deylon, Pierre yves Glorennec, Hakan Hjalmarsson, and Anatoli Juditsky. Nonlinear black-box modeling in system identification: a unified overview. *Automatica*, 31:1691–1724, 1995.

[44] M. Snir, S.W. Otto, D.W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference.* MIT Press Cambridge, MA, USA, 1995.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] D. Whitley. Genetic algorithms and neural networks. *Genetic Algorithms in Engineering and Computer Science*, pages 203–216, 1995.

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

[52] 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? *Proceedings of the IEEE*, 93:358–386, 2005.

[53] 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.