Model-Guided Performance Tuning of Parameter Values: A Case Study with Molecular Dynamics Visualization

Yoonju Lee Nelson*, Bhupesh Bansal†, Mary Hall*, Aiichiro Nakano†, and Kristina Lerman*
*USC/Information Sciences Institute; Marina del Rey, CA 90292
†University of Southern California; Los Angeles, CA 90089

Abstract

In this paper, we consider the interaction between application programmers and tools that automatically search a space of application-level parameters that are believed to impact the performance of an application significantly. We study performance tuning of a large scientific application, the visualization component of a molecular dynamics simulation. The key contribution of the approach is the use of high-level programmer-specified models of the expected performance behavior of individual parameters. We use these models to reduce the search space associated with the range of parameter values and achieve results that perform close to that of a more exhaustive search of the parameter space. With this case study, we show the importance of appropriate parameter selection, with the difference between best-case and worst-case performance with a particular input data set and processor configuration of up to a factor of 17. We show that through the use of models, we can drastically reduce search time, examining only 0.3% to 5% of the search space, and usually select an implementation that is close to the best performance, within 0.84% to 15%, even though the models are not completely accurate.

1 Introduction

The complexity of today’s architectures is growing, with many-core processors and petascale systems projected in the near future, increasing use of high-performance accelerators such as graphics processors and FPGAs, and incorporation of a diversity of architectural features including SIMD compute engines, deep memory hierarchies, and software-managed storage. Achieving high performance on a new architecture will demand that codes are carefully tuned to the unique features of that system.

Historically, the burden of achieving high performance on new platforms has largely fallen on the application programmers. For key portions of their computation, the programmer derives a sequence of different but equivalent implementations. Each implementation variant is first debugged, and then its performance characteristics are evaluated. This lengthy process continues until the programmer either arrives at a suitable implementation, or, as is often the case, decides to give up on further performance tuning. Different implementations are often needed for different input data sets, or as the application is ported to other platforms.

In this paper, we propose tool support for this performance tuning process, with the goal of dramatically increasing the productivity of application scientists in tuning and porting their codes. We focus on empirical techniques to find the best integer value of application-level parameters that the programmer has identified as being critical to performance. For example, in our case study, the computation partitioning across X, Y and Z dimensions represents a set of dependent application-level parameters. Our work is closely related to Active Harmony, an automated tuning system that performs on-line selection of application-level parameter values [5, 15]. As compared with Active Harmony, which assumes no a priori knowledge of the behavior of application parameters, we propose an interaction with the programmer whereby they provide high-level models of the impact of parameter values, which are then used by the system to guide and to prune the search for optimization parameters. We call this approach model-guided empirical optimization, in that models and empirical techniques are used in a hybrid approach.

To evaluate this concept, we perform extensive experiments with a case study application, the visualization component of a molecular dynamics simulation. The application programmer has identified a set of five application-level parameters believed to have significant impact on performance. Through the programmer’s insights into perceived performance impact of the parameters, we have developed a set of general parameter models that describe common behavior of an optimization parameter, when considered in isolation. The programmer has also provided guidance on appropriate default parameter values and ranges. We use the case study to demonstrate the effectiveness of this approach. We ran a series of experiments on 1, 2, 4 and 8 processors for three time steps of an MD simulation to evaluate how well the models predict application performance and limit the number of points to be tested.
The contributions of the paper are: (1) a set of high-level models that correspond to behavior of many optimization parameters; (2) how to use programmer-supplied models in empirical search of application-level parameters; (3) search algorithms that exploit the models to prune the search space; and, (4) a set of results for our case study that show that, through the use of models, we can find parameters that perform within 0.84% to 15% of the best performance in the sampled space while only searching between 0.3% and 5% points in the search space.

2 Related Work

The notion of using empirical search to select optimization parameter values was popularized by recent self-tuning domain-specific library-generators such as ATLAS [17], PhiPAC [1], FFTW [9] and SPIRAL [18], which incorporate models and empirical search to port these libraries to new architectures automatically.

In compiler optimization, historically models of architectures and impact of code transformations were used exclusively for guiding optimization. Recent research has shown success in hybrid techniques that use models to focus on the most promising candidate implementations and prune the search space, and empirical search to fine tune and select parameter values [3, 10, 19]. A variety of AI search techniques, such as simulated annealing [12], hill climbing [8] and genetic algorithms and machine learning [7, 13, 16] have shown promise in improving compiler optimization results; at the same time, the enormous cost of these searches can be prohibitive since they incorporate little if any domain knowledge to limit the search space.

The work most closely related to ours is Active Harmony, described in the previous section, an automated tuning system that makes applications tunable with minimal changes to application structures [5, 6, 14, 15]. Active Harmony is a sophisticated infrastructure that permits on-line rewriting of parameter values at run-time, and has been used with large scientific applications. A Simplex method is used to select parameter values, assuming little or no a priori knowledge of parameter behavior. As an exception to this, the system can exploit prior runs to select the starting point for search [5]. In contrast, our work has relied completely on application programmer models.

3 Key Concepts

Figure 1 presents an overview of the proposed system for facilitating performance tuning. The programmer provides the application code and the performance model for the parameters to be considered. Parameter Search Script describes how to search for optimization parameter values, provides ranges and default values for each parameter, and additional constraints on parameter values, similar to Chen et al. [4]. Since we are considering applications with multiple parameters, the search script describes an ordering in which the parameter search should be performed. While searching for the value of one parameter, a default or previously-selected best value for the other parameters is used. The centerpiece of the system is an experiments engine that systematically evaluates, through empirical measurement, a collection of different parameter values for a key computation. The remainder of the section describes some of these concepts in more detail.

3.1 Application Parameters

Many times application programmers can identify parameters associated with their application that impact performance. The best value for such a parameter is affected by input data, computation phases and system load, and multiple parameters may affect each other.

In our approach, we focus on integer-valued parameters, and our system automatically performs the search for the best parameter value. While in this paper we focus on off-line search, such a tool could also adapt behavior during application execution, as is done in Active Harmony [5, 6, 14, 15]. To guide the search for parameter values, we rely on models provided by the application programmer, which describe the relationship between parameter values and application performance.

To clarify the use of models in parameter search, we describe three models from the parameters in our case study. As the approach is refined and applied to additional programs, new models can be added, or the application programmer can develop their own functional specifications or programmable models.

3.2 Parameter Models

*Capacity Parameters:* Many application parameters are known to impact utilization of storage structures (e.g., cache or registers, or local memory), and through adjustment of this parameter value, the application will better exploit reuse of data in nearby storage. Holding everything else constant, the execution time associated with such a parameter will decrease up to the point where the storage is best utilized, and after this point, will increase due to exceeding the storage capacity, as in Figure 2(a) below. As examples of Capacity parameters from the compiler literature, when performing tiling of loop nests to access reused data out of cache, increasing tile sizes generally improves performance, but tile sizes that are so large that the data footprint...
of a tile exceeds cache capacity may degrade performance, sometimes significantly [3]. Similar examples arise in application code, and one such parameter will be described in our case study.

Balance Parameters: Some application parameters trade off the use of multiple resources. The lowest execution time is achieved when the selected parameter value achieves a balance of the two resources. In compiler optimization, the notion of achieving balance was introduced by Carr et al. to trade off reuse of data in registers and floating point unit performance [2]. In this case, unroll factors for multi-dimensional loop nests are selected to achieve reuse of data in registers for computations that are both memory-intensive and floating-point intensive. Larger unroll factors, which might increase reuse in registers, will not reduce execution time if the floating point unit is saturated. Further, too large an unroll factor can negatively impact execution time in other ways, such as, for example, register spills or reduced instruction cache utilization. Thus, the goal is to select an unroll factor that achieves a balance between register reuse and floating-point utilization, constrained to not exceed register capacity. As another example, selecting the number of processors to use for a parallel application may want to achieve a balance between speedup and processor utilization. As shown in Figure 2(b), Balance parameters have behavior similar to Capacity parameters, but may have a slightly different shape. Execution time does not necessarily increase after an inflection point, but it stops improving.

Functional Specification Parameters: The previous models are general, but other models may need to capture the complex relationship between input data properties and parameter selection. In such cases, it is useful to permit the programmer to provide a functional specification that maps data input properties to the selection of parameter values, as \( P = f(I) \).

4 Case Study: MD Visualization

A molecular dynamics (MD) simulation follows the trajectory of atoms to study material properties. High-end MD simulations involve over one billion atoms, which produce 100GB of data per frame to record atomic species, positions, velocities and stresses. Such large-scale MD simulations are typically performed at remote supercomputing centers using thousands of processors, whereas the resulting data are often stored and analyzed over an extended period on smaller computing resources available locally to the user.

Advanced MD data analysis algorithms are often based on an undirected graph, \( G = (V, E) \), where the vertex set \( V \) consists of atoms and the edge set \( E \) contains bonds between pairs of atoms. This paper focuses on the parallel molecular fragment analysis of large MD simulation data, which amounts to the connected component analysis of a multimillion-vertex graph. Fragment analysis may be performed on a wide range of datasets with sizes ranging from 100,000 atoms to 1 billion atoms and diverse characteristics, e.g., dense to sparse graphs, and a small number of large fragments to a large number of small fragments.

As described in the previous section, the application programmer has identified a set of parameters believed to impact the performance of the application significantly. Even within a single simulation, the best parameter values are sensitive to which time step is being analyzed. For example, as atoms disperse, the density of the simulated system changes dramatically, and consequently, the performance behavior changes. This change in behavior across time steps is a common feature of large-scale simulation since the purpose of the simulation is to analyze the impact over time. Thus, it is important not only to set the values of these parameters correctly for the initial time step, but also to adjust parameter values across time steps.

For this molecular fragment analysis implementation, the application programmer has identified five parameters, three of which are selected together. We now describe the parameters and their associated models.

Cell Decomposition Size parameter: We partition the 3-dimensional data space into cells, which consist of a collection of adjacent atoms. The atoms are assigned to cells according to their positions. The cell size directly affects performance; larger cells lead to greater data reuse in cache, but increase the amount of computation. The optimal cell size finds a balance between these two optimization goals, and is therefore a Balance parameter.

Cache Size parameter: The implementation uses temporary data structures to buffer currently used information, as well as that to be used in the near future. This parameter was introduced into the code to support analyses that exceed the capacity of various levels of the storage hierarchy, and particularly main memory. Analyses of very large out-of-core simulations or simulations that fit within main memory but exceed hardware cache capacity, are greatly affected by the Cache Size parameter. Given these properties, Cache Size is a Capacity parameter.

Computation partitioning parameter: The physical system to be simulated is partitioned into smaller subsystems. We use a standard 3D mesh decomposition, and each
subsystem is mapped to a processor in an array of \( x \times y \times z \) processors, where \( x, y, \) and \( z \) are positive integers. The decomposition is done to assign equal load to each compute node, in an attempt to minimize the execution time of the most heavily loaded processor. As compared to the previous two parameters that are treated independently, the parameter values of \( x, y, \) and \( z \) are dependent, and must be set simultaneously. Given particular values of \( x, y, \) and \( z, \) let \( \text{Atoms}_p(x, y, z) \) refer to the number of atoms assigned to processor \( p \) and \( \text{Cross}_p(x, y, z) \) refer to the number of atoms not on processor \( p \) that are linked to atoms assigned to processor \( p. \) We can use the following model for the functional mapping to predict what will be the best values for \( x, y, \) and \( z, \) once input \( I \) is known at the beginning of the time step.

\[
\min_{x,y,z} L(x, y, z) = w_1 \times \max_p \text{Atoms}_p(x, y, z) + w_2 \times \max_p \text{Cross}_p(x, y, z)
\]

That is, the best load balance is achieved when the maximum number of atoms and cross atoms assigned to any processor is minimized across all possible partitionings \([11].\) Architecture-specific weights \( w_1 \) and \( w_2 \) allow the model to vary the importance of atoms and cross atoms, depending on the cost of cross-processor communication.

5 Approach for Finding Parameter Values

In this section, we describe how the approach described in Section 3 was applied to the case study of Section 4 in an off-line performance tuning phase.

5.1 Search Strategy

To express the search process for our case study, the application programmer must describe the parameters, their models, and provide an order for the search. A script with this information for the case study of Section 4 is shown here.

```plaintext
PARAMETER (X,Y,Z), RANGE=(1:P), CONSTRAINT(X*Y*Z=P), MODEL=FUNCTIONAL(eq1)
PARAMETER (CELL=MAPPING(24)), RANGE=(0:48), MAPPING=(T*2+48), MODEL=BALANCE
PARAMETER (CACHE=MAPPING(2)), RANGE=(0:5), MAPPING=(T), MODEL=CAPACITY
```

The ordering of parameters in this script indicates that the most important parameter to performance, according to the programmer, are \( X, \) \( Y, \) and \( Z, \) which are called the computation partitioning parameters of Section 4. Because the values of these parameters are dependent and must be selected simultaneously, their description in the script appears in a single line. The script provides the name of the three parameters, which can then be connected to the code. A range of values for each parameter (1:P) is specified, but there is the additional constraint that \( X \times Y \times Z = P, \) which must be satisfied when new parameter values are tested. Because the parameter uses a functional mapping, there will not be a search and a fixed value will be selected for input \( I \) using \( \text{eq1} \) from the previous section. But the functional mapping could potentially be used to seed a value, with a search of nearby points.

According to the programmer, cell decomposition size is the next most significant parameter in terms of performance

```
CapacityParameterSearch(low, high, default)
  best = default;
  if (D_R > 0) \{ best = TripleSearch(default,high,defaul, default+2*D_R);
  if (best == default) then best = TripleSearch(default, default+1*D_R);
  \}
else return CapacityParameterSearch(low, high, default);

BalanceParameterSearch(low, high, default)
  best = default;
  if (D_L > 0) \{ best = TripleSearch(low, default, default+2*D_L);
  if (best == default) then best = TripleSearch(low, default+1*D_L);
  \}
else return BalanceParameterSearch(low, high, default);

TripleSearch(low, high, best, compare)
  if (high <= low) return best;
  if (low < compare < high) return best;
  if (Time(compare) < 1.02*Time(best))
    return BalanceParameterSearch(low, high, compare);
  else return best;
```

Figure 3. Search Algorithms for Parameter Models.
impact. For cell size, we will search in a small range, even-numbered values between 48 and 144. To aid in the process of selecting new points, the range of values is normalized to be from 0 to 48, and a MAPPING function takes these consecutive values and maps them into the range of even-numbered values from 48 to 144. The initial value is 96, which is in the middle of the range, and the model for this parameter is BALANCE.

The final parameter is cache size, with a range normalized to be from 0 to 5, with a mapping that expresses $3^2 = 9$, and the model for this parameter is CAPACITY.

5.2 Models in Parameter Search

Figure 3(a) and (b) presents representative algorithms that use the CAPACITY and BALANCE models, respectively, to guide parameter search, with an example in Section 6. Figure 3(a) given the expected functional form of CAPACITY parameters in Figure 2(a), is based on binary search. Our search favors smaller parameter values because smaller values are assumed to use less storage resources, and we would like to pick the smallest value that leads to good performance. Thus it starts on the left side of the default value. Two points are compared, and the range of values is adjusted. For cache size, which is a CAPACITY parameter, the range of values 0 to 6 is provided, and the associated mapping function above is used to set the parameter value. When the two results considered have less than 2% difference in performance, the search terminates.

For a BALANCE parameter model, we expect the best performance to usually occur at the high end of the parameter space, so the algorithm in Figure 3(b) focuses on points to the right of the default. The range of values to the right of default is divided into three rather than two portions, and we focus on the point that is two-thirds to the end first, then one-third. If the algorithm searches to the left of default, the point one-third to the left from default is searched first, followed by two-thirds.

6 Results

In this section, we describe a set of experiments that show the utility of using models to reduce the number of points to search.

6.1 Input data description

<table>
<thead>
<tr>
<th></th>
<th>Step I</th>
<th>Step II</th>
<th>Step III</th>
</tr>
</thead>
<tbody>
<tr>
<td># of Atoms</td>
<td>1,141,254</td>
<td>1,141,254</td>
<td>1,141,254</td>
</tr>
<tr>
<td># of Bonds</td>
<td>5,494,711</td>
<td>2,033,728</td>
<td>1,617,747</td>
</tr>
<tr>
<td>Volume</td>
<td>25.20M</td>
<td>9614.03M</td>
<td>1,1542.19M</td>
</tr>
</tbody>
</table>

Table 1. OAl₄ Input Data Set Information.

We have experimented with an input data set of 1.1M OAl₄ atoms, and three separate inputs which represent different time steps. Table 1 describes the structure of the data for all three time steps. Throughout this section, we will call them Step I, Step II, and Step III; they are not consecutive time steps, as we wanted to focus on the differences after many time steps, but this is the sequence in which they occur. Each input has the same number of atoms (1.1M) representing the nodes of a graph, but the number of Bonds, which represents the edges or connections between atoms, varies significantly across the three time steps. The Volume row is the result of multiplying the X, Y, and Z axis sizes. We found that Step I’s input data set has very high density, and after the detonation, the atoms disperse, so that Step II and Step III are much less dense. Figure 4 illustrates through visualization all three time steps of the OAl₄ input data set.

6.2 Experimental Methodology

Our experiments were conducted on a Linux cluster of 3.2GHz dual Intel Xeon nodes with 20GB memory. We show the wall-clock execution time for molecular fragment analysis. We have measured all three time steps on 1, 2, 4, and 8 processors. Due to the density of the Step I input data, we were unable to gather 1-processor data, so for this data, only results for 2, 4 and 8 processors are shown.

The number of possible values for computation partitioning parameters is 1 for 1 processor, 3 for 2 processors, 6 for 4 processors and 10 for 8 processors. There are 48 possible values for cell decomposition size, and 6 possible values for cache size. Thus, the overall search space for these five parameters is between 288 points (for 1 processor) and 2880 points (for 8 processors). To capture the landscape of performance results across different parameter values, we ran a very large number of experiments for each combination of input data set and number of processors. Because testing
the entire search space would take a prohibitive amount of time, we sampled the computation partitioning and cell size parameters. We tested only those computation partitioning parameters that were selected as either the lowest or highest cost according to (eq1). For cell size, we tested seven points between 48 and 144, using an interval of 16. With this sampling, we tested a search space of 42 points for 1 processor and 84 points for 2, 4 or 8 processors. A few additional points that were considered during the search algorithms were also tested.

To achieve stable performance results, we ran each point in the search space at least seven times. We dropped the best and worst performance and computed the average of remaining points. For each average, we computed the standard deviation, and if it was more than 10%, we repeated the measurement for that case. While this methodology produced stable results, in an on-line use of this tool, the programmer would not want to repeat experiments so many times. Active Harmony’s approach to on-line testing in unstable environments is to use the minimum execution time rather than the average, and rerun a point only so many times as is needed to obtain stable minimums [5]. But we found averages to be reliable measurements across so many runs of the code.

6.3 Overview of results
To examine the impact of the parameter choices on performance, Figure 5 shows the performance speedup between the best and worst parameter values among all points we tested, ranging from almost 1.5 to almost 17.5.

6.4 Validity of Models
Now we look at the performance impact of individual parameters, and the validity of the models provided by the application programmer.

6.4.1 Computation Partitioning
We first evaluate the computational parameters, X, Y and Z. The functional model in (eq1) selects the minimum cost for the weighted sum of maximum number of atoms and the number of cross atoms each processor. Generally, we found that with the three time steps, the weights for atoms and cross atoms was not important, since whenever the number of atoms is higher, so also is the number of cross atoms, but in one case, for 2 processors, Step I, the cross atoms must have a higher weight. In all but one case, the functional model selected the best performing solution. The one exception is Step III, with 2 processors, where performance of other X, Y, and Z is slightly better than selected parameters in the functional model (eq1).

Figure 6 presents the speedup of the computation partitioning parameter choices parameters which (eq1) predicts using the default cell decomposition size and cache size parameters. Overall, we achieved a performance improvement of up to 6.4x, with an average speedup of about 3.0x. For 2 processors, Step III, the speedup was 0.91, indicating the one case where the model was not an accurate predictor of the best performance.

6.4.2 Cell Decomposition Size
The cell decomposition size parameter is assumed to be a BALANCE parameter. Figure 7 shows a subset of curves of cell decomposition sizes in the range of 48 to 144, sampled every 16 points. For these figures, we held the computation partitioning parameters constant, and used the selections from (eq1). Each separate curve shows a different cache size value, as indicated in the legend. Each graph is a different combination of input data and number of nodes. The results, particularly for Step III and for larger numbers of processors, are not always following a stable performance trend. Nevertheless, overall it seems that BALANCE is an appropriate model for this parameter. Some graphs actually degrade, which makes sense given that the overhead associated with cell size may dominate for larger values. While not completely accurate, as will be discussed, the BALANCE model is useful in predicting cell decomposition size.

6.4.3 Cache Size
The cache size parameter is a CAPACITY parameter. In Figure 8, we show a subset of graphs for cache size, where each separate curve represents a different cell size. In general, the cache size parameter makes little difference to performance. There are a few very bad points, and some definite better points, but the curves are relatively flat. Some curves follow the capacity model, but for the most part, the application programmer’s assumption about the model was not valid. We suspect this is because the application fits in local memory, and caching is not having a significant performance benefit.
Figure 9. Prune Search Graphs in Step III, 8 Nodes.

(a) Example of Balance Parameter Search

(b) Example of Capacity Parameter Search

6.5 Pruning Search Using Models

Using the BALANCE and CAPACITY models, and the algorithms from Section 5.2, we now present results showing pruning of the search space using the models. We are using the fixed computation partitioning parameters selected by the FUNCTIONAL model of (eq1). First we will select the cell decomposition parameter using a default value for the cache size, and then we will search for the cache size. Figure 9(a) presents an example of pruning the search space for the cell decomposition parameter, using the algorithm of Figure 3(b). In this figure, the values inside circles represent the currently selected parameter for cell decomposition size, and the numbers next to the circles show the execution time for that parameter value. The dark-colored circle is the parameter value selected by the algorithm. The CUT annotation shows portions of the parameter space that are pruned by the algorithm. Similarly, Figure 9(b) shows an example of pruning the search space for the cache size parameter using the algorithm of Figure 3(a), after the cell decomposition parameter value is set to 140.

Table 2 shows the resulting cell decomposition and cache size parameters for each input data and number of processors. The fourth column is the speedup of these parameter choices over the parameter values exhibiting the

<table>
<thead>
<tr>
<th>Step</th>
<th>Node</th>
<th>Cell</th>
<th>Cache</th>
<th>Speedup</th>
<th>Points</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>2</td>
<td>80</td>
<td>81</td>
<td>2.00</td>
<td>15+4=19</td>
<td>1.02%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>88</td>
<td>9</td>
<td>1.25</td>
<td>13+3=16</td>
<td>8.59%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>140</td>
<td>9</td>
<td>1.30</td>
<td>7+3=10</td>
<td>5.49%</td>
</tr>
<tr>
<td>II</td>
<td>1</td>
<td>138</td>
<td>9</td>
<td>1.60</td>
<td>6+3=9</td>
<td>14.58%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>134</td>
<td>3</td>
<td>2.53</td>
<td>6+3=9</td>
<td>0.84%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>134</td>
<td>1</td>
<td>2.78</td>
<td>6+3=9</td>
<td>11.85%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>136</td>
<td>27</td>
<td>1.74</td>
<td>8+4=12</td>
<td>11.70%</td>
</tr>
<tr>
<td>III</td>
<td>1</td>
<td>96</td>
<td>9</td>
<td>1.76</td>
<td>10+3=13</td>
<td>2.62%</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>142</td>
<td>3</td>
<td>1.32</td>
<td>5+3=8</td>
<td>7.50%</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>112</td>
<td>9</td>
<td>1.73</td>
<td>9+3=12</td>
<td>11.17%</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>140</td>
<td>9</td>
<td>1.75</td>
<td>6+3=9</td>
<td>7.80%</td>
</tr>
</tbody>
</table>
worst-case performance in the sampled parameter space (but with computation partitioning parameter values held to the best-performing solution). The selected parameter values lead to a speedup of between 1.25 to 2.78 over the worst-performing solutions for fixed numbers of processors, with most of the performance gain coming from the cell decomposition parameter. The fifth column represents how many points were tested by the algorithms of Figure 3, first for cell decomposition, and then cache size, and the sum of the two. We selected the optimized parameters after testing between 8 and 19 points in the search space of between 288 and 2880 points. The last column, Distance, represents the difference in performance between the parameter value selected by the models and the best performance result in the sampled search space, ranging from 0.84% to almost 15%. In summary, through the use of models, we can achieve performance that is nearby the best performance in the sampled search space, while only testing between 0.3% and 5% of the search space.

7 Conclusion and Future Work

This paper has evaluated an approach to empirical tuning of application-level parameters that uses models of the parameters’ behavior as specified by the programmer. Even if not completely accurate, the models provide the programmer a way to express expected behavior, which can then be utilized by the system. In this paper, we used the models to guide optimization search and to prune the search space. We found that through the use of models, we examined a small fraction of the search space (less than 5%), and achieved performance within 15% and often much closer to the best performance in our sampled search space.

In general, we are interested in the question of how application programmers can interact with performance tuning tools in a productive way. We see several other uses for programmer-defined models in a performance tuning tool. First, the tool could validate the application programmer’s model through sampling of the search space. If the model is proven inaccurate, the programmer may develop a new understanding of their application, or the programmer or system may be able to learn a new model. Second, the system could learn a model for a parameter from a fixed set of possible models, using the same type of validation mechanism. In either case, feedback to the user on the model associated with a parameter might help refine the model (for example, to find new ranges of values) and could persist as part of the application as it is ported to new machines or adopted by new users. Third, the model could be used in conjunction with tools that employ more extensive empirical search to seed the search and to prune uninteresting portions of the search space.

These other uses of models, on-line performance tuning with models and considering a larger set of applications are the subject of future work.

Acknowledgements. This research has been supported by the National Science Foundation under awards CSR-0509517 and CSR-0615412. The authors wish to thank Ken-ichi Nomura, Cheng Zhang, and Weiqiang Wang from USC for assistance with the MD code and data sets.

References


