Machine Learning for Automatic Performance Tuning

Nicholas Chaimov <nchaimov@cs.uoregon.edu>
Advisor: Allen D. Malony
Computer & Information Science — University of Oregon

Introduction

Automatic Performance Tuning is the problem of producing, from input code, output code which performs better on than the input on some metric, such as time to execute or energy consumption. **Empirical Autotuning** is an approach to this problem which involves searching a space of transformed code variants (*see transformations, below*). By measuring the performance of code variants, autotuning produces variants optimized for particular execution environments or input parameters. An exact solution to this search problem involves mixed integer nonlinear programming, which is NP-hard, so approximate search is used. Here, we use decision tree learning to select starting configurations for search in order to generate runtime-adaptive code and to speed autotuning by selecting good starting search configurations.

Transformations

Permutation / Loop Interchange

```
for(int i = 0; i < I; ++i) {
   for(int j = 0; j < J; ++j) {
      for(int k = 0; k < K; ++k) {
         computation(i,j,k);
      }
   }
}
```

→

```
for(int j = 0; j < J; ++j) {
   for(int i = 0; i < I; ++i) {
      for(int k = 0; k < K; ++k) {
         computation(i,j,k);
      }
   }
}
```

Loop Unrolling

```
for(int i = 0; i < I; ++i) {
   for(int j = 0; j < J; ++j) {
      for(int k = 0; k < K; ++k) {
         computation(i,j,k);
      }
   }
}
```

→

```
for(int i = 0; i < I; ++i) {
   for(int j = 0; j < J; ++j) {
      for(int k = 0; k < K; k+=2) {
         computation(i,j,k);
         computation(i,j,k+1);
      }
   }
}
```

Loop Tiling / Blocking

```
for(int i = 0; i < I; ++i) {
 for(int j = 0; j < J; ++j) {
  for(int k = 0; k < K; ++k) {
   result[(i*I)+j] +=
    a[i*I + k] * b[k*K + j];
```

→

```
for(int ii = 0; ii < I; ii += B)
 for(int jj = 0; jj < J; jj += B)
  for(int kk = 0; kk < K; kk += B)
   for(int i = ii; i < MIN(ii+B,I); ++i)
    for(int j = jj; j < MIN(jj+B,J); ++j)
     for(int k = kk; k < MIN(kk+B,K); ++k)
      result[(i*I)+j] +=
       a[i*I + k] * b[k*K + j];
```

Loop Fusion

```
for(int i = 0; i < I; ++i) {
   computation_1(i);
}
for(int i = 0; i < I; ++i) {
   computation_2(i);
}
```

→

```
for(int i = 0; i < I; ++i) {
   computation_1(i);
   computation_2(i);
}
```

Loop Splitting

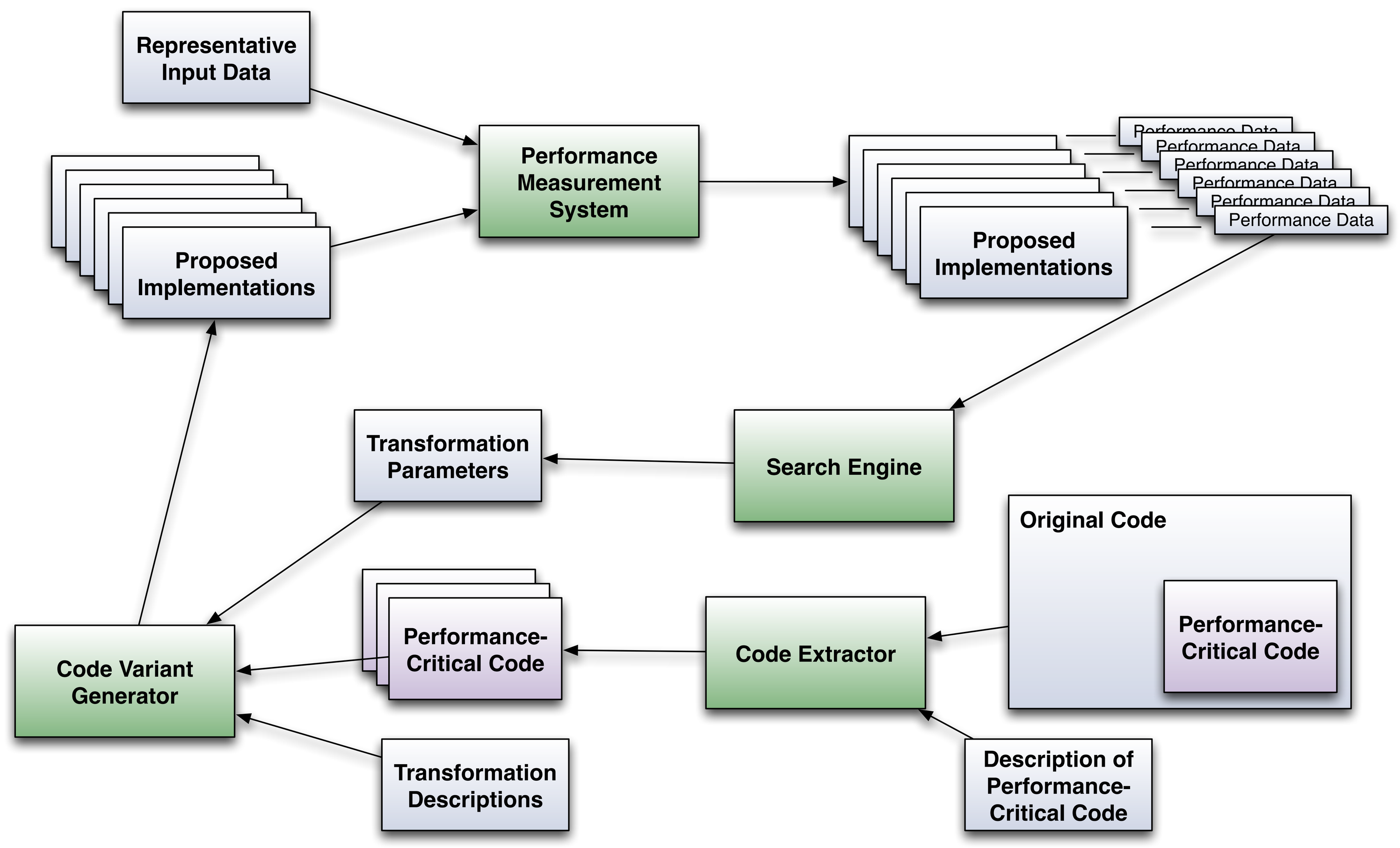
```
for(int i = 0; i < I; ++i) {
   computation_1(i);
   computation_2(i);
}
```

→

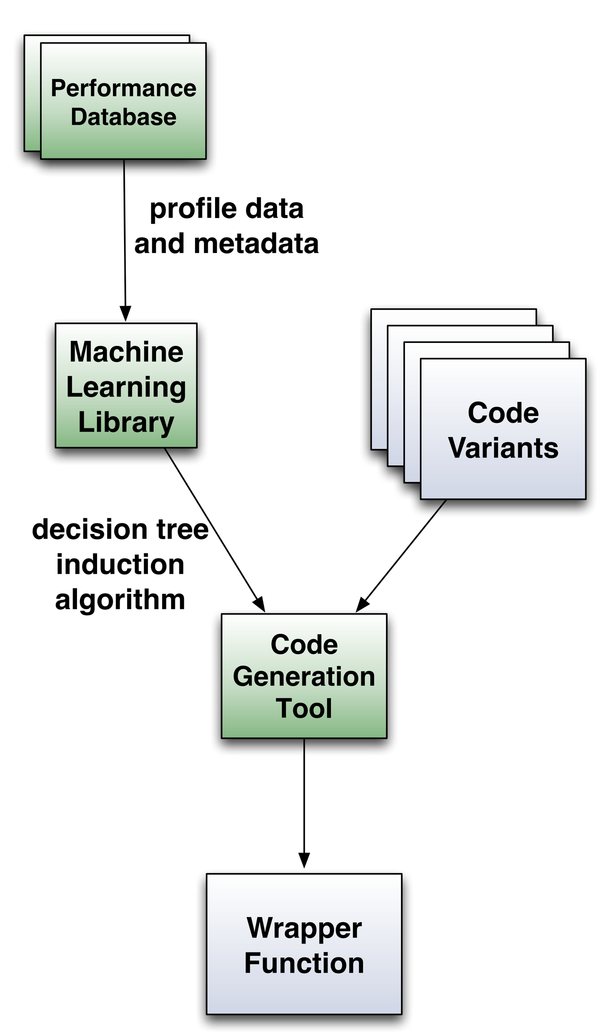
```
for(int i = 0; i < I; ++i) {
   computation_1(i);
}
for(int i = 0; i < I; ++i) {
   computation_2(i);
}
```

Design

We have designed a system for empirical autotuning with an architecture shown in the figure below:

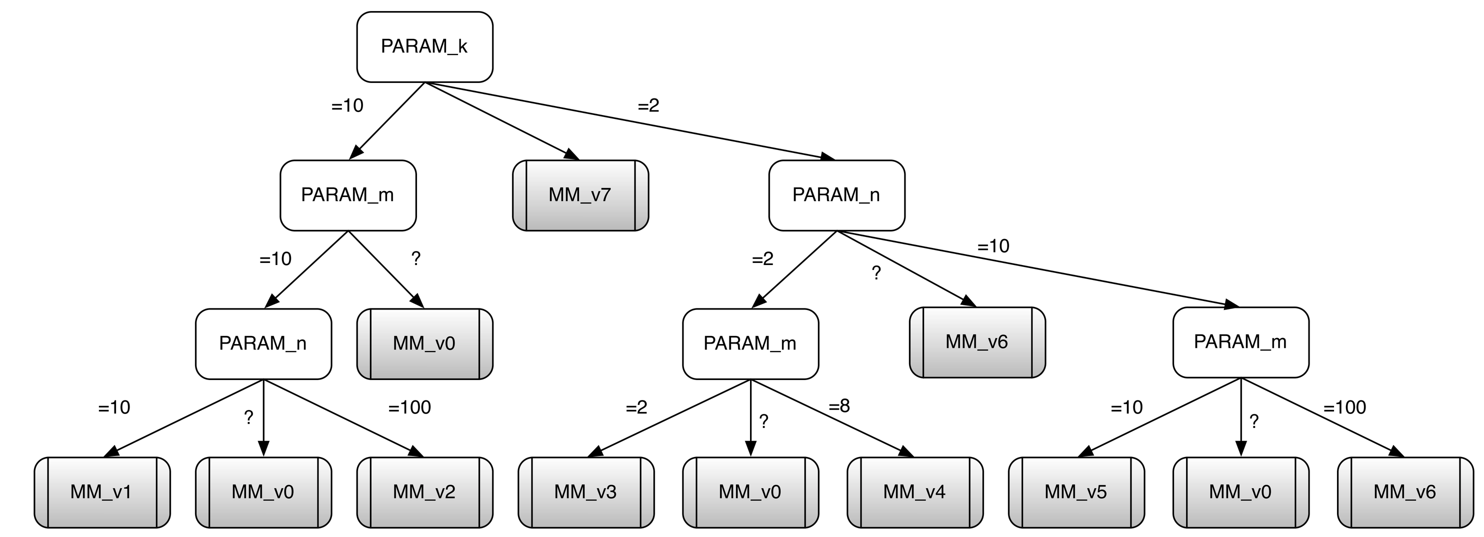


In our system, a baseline performance measurement of the unoptimized code is made using the TAU Performance System. A search engine (such as Active Harmony [1] or Orio [2]) proposes variants based on a parameterized transformation script. A code variant generator (such as Orio [2] or CHiLL [3]) generates the variants, which are substituted into the original program. The transformed programs are then run and their performance measured, and the process repeats until the search termination conditions are met. All of the performance profiles are tagged with metadata describing the transformations applied, the execution environment, and the input data, and are stored in a centralized performance database, TAUdb.

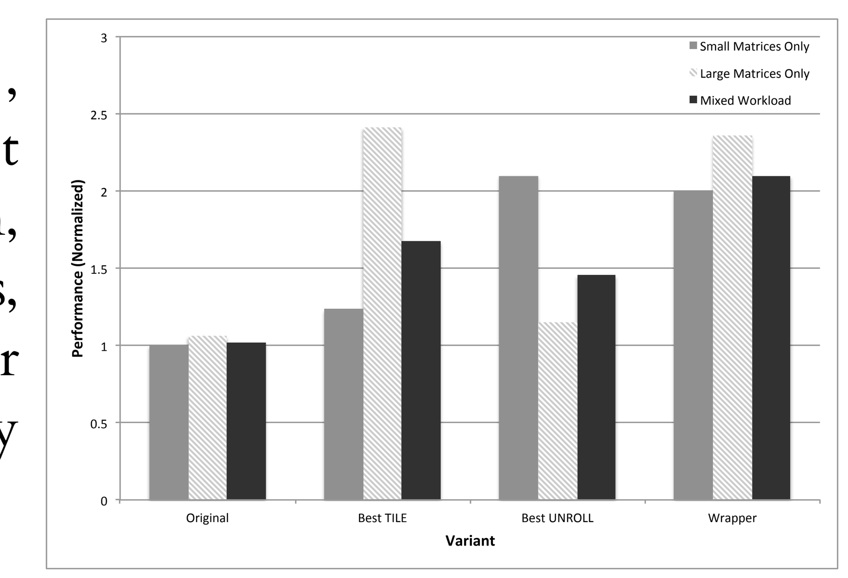


Autotuning is repeated across multiple computers and multiple input datasets, yielding **annotated performance data** which we can use to generate **runtime-adaptive code** or to **improve the search process**. In both cases, this is accomplished by performing **decision tree learning** over the performance data. To generate runtime-adaptive code, the decision tree is converted into executable code in the form of a wrapper function.

Evaluation



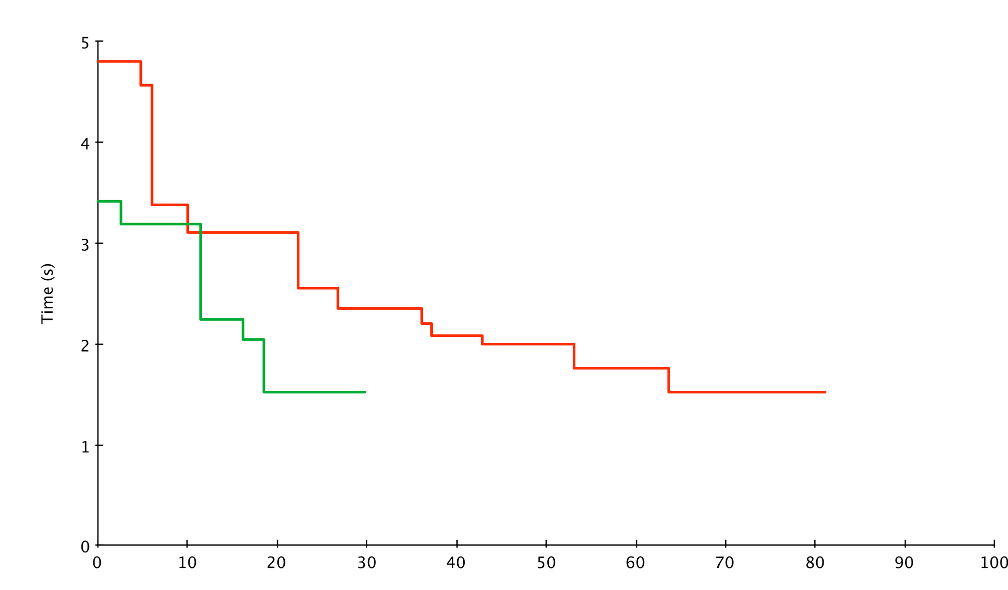
The decision tree can be generated using **execution environment** (such as CPU architecture, number of cores, cache size, etc.) and **input parameters** (such as input size) as features. For example, the above tree selects a variant of a 3D matrix multiplication routine based upon the size of the input matrix.



For small matrices, unrolling is the best optimization to perform, while for large matrices, loop tiling yields better performance by improving data reuse.

When the sizes of the input matrices are not known until runtime, using a generated wrapper function to select a variant produces better performance than either the unoptimized variant or variants optimized for small or large matrices.

The variant selected by a decision tree can also be used as the initial configuration for search, resulting in the search process converging earlier, as in the graph below, showing time to converge for default (red) and predicted (green) configurations.



References

[1] Hollingsworth, J. and A. Tiwari (2010, June). End-to-End Auto-Tuning with Active Harmony, Chapter 10, pp. 217–238. CRC Press.
[2]Hartono, A., B. Norris, and P. Sadayappan (2009). Annotation-based empirical per- formance tuning using Orio. In Proceedings of the 23rd IEEE International Par-
[3]Tiwari, A., J. K. Hollingsworth, C. Chen, M. Hall, C. Liao, D. J. Quinlan, and J. Chame (2011, August). Auto-tuning full applications: A case study. Int. J. High Perform. Comput. Appl. 25(3), 286–294.