PTUNE: A TOOL FOR ONLINE AUTOTUNING WITH OPTIMIZING COMPILERS

THESIS

Presented to the Graduate Council of Texas State University-San Marcos in Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

by

SANTOSH R. SARANGKAR

San Marcos, Texas December 2011
PTUNE: A TOOL FOR ONLINE AUTOTUNING WITH OPTIMIZING COMPILERS

Committee Members Approved:

________________________
Apan Qasem, Chair

________________________
Carol Hazlewood

________________________
Yijuan Lu

Approved:

________________________
J. Michael Willoughby
Dean of the Graduate College
FAIR USE AND AUTHOR’S PERMISSION STATEMENT

Fair Use

This work is protected by the Copyright Laws of the United States (Public Law 94-553, section 107). Consistent with fair use as defined in the Copyright Laws, brief quotations from this material are allowed with proper acknowledgment. Use of this material for financial gain without the author’s express written permission is not allowed.

Duplication Permission

(Choose one of the two below and type only it on the page)

As the copyright holder of this work I, Santosh Sarangkar, authorize duplication of this work, in whole or in part, for educational or scholarly purposes only.

OR

As the copyright holder of this work I, Santosh Sarangkar, refuse permission to copy in excess of the “Fair Use” exemption without my written permission.
ACKNOWLEDGEMENTS

It is with enormous gratitude that I acknowledge the support and help from Dr. Apan Qasem, who had been my advisor in my thesis, and academics. Especially the support and help I experienced during my master’s degree was extraordinary. I am so fortunate to have him as my thesis advisor. Also, I would like to thank him for always being available to help. It would have been next to impossible to write this thesis without his timely help and guidance.

Also, I want to thank my committee members Dr. Yijuan Lu and Dr. Carol Hazlewood, for their valuable suggestions. This thesis is dedicated to my parents Raosaheb and Chanchal, brother Devidas, sister Mangal, and sister-in-law Namrata who have given me the opportunity of education from the best institutions and support throughout my life. Lastly, I offer my honors and blessings to all of those who supported me in any respect during the completion of my master’s degree.

This manuscript was submitted on August 1, 2011.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>RELATED WORK</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Autotuned Libraries</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Search-based Tuning of Applications</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Tunable Transformations</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Component Tools</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>TUNING FRAMEWORK</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Search Space Specification</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Exposing Optimization Flags in the GNU Compiler</td>
<td></td>
</tr>
</tbody>
</table>
Collection (GCC) .................................................. 17
3.4 Search Algorithms ........................................... 20
  3.4.1. Genetic Search ......................................... 21
  3.4.2 Random Search .......................................... 24
  3.4.3 Direct Search ............................................ 25
  3.4.4 Simulated Annealing ................................... 27
4: EXPERIMENTAL RESULTS ....................................... 28
  4.1 Experimental Setup ...................................... 28
  4.2 Overall Performance Improvement ....................... 29
  4.3 Search Space Exploration ............................... 31
5: CONCLUSION .................................................. 35
REFERENCES ........................................................ 36
**LIST OF TABLES**

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: gcc Compiler Optimizations</td>
<td>18</td>
</tr>
<tr>
<td>2: Benchmark Applications and their Descriptions</td>
<td>29</td>
</tr>
<tr>
<td>3: Platform Specifications</td>
<td>29</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figures</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: PTUNE Framework</td>
<td>14</td>
</tr>
<tr>
<td>2: Example configuration file for PTUNE</td>
<td>16</td>
</tr>
<tr>
<td>3: Specifying Flags</td>
<td>20</td>
</tr>
<tr>
<td>4: Randomly Generated Population</td>
<td>23</td>
</tr>
<tr>
<td>5: Generating New Population</td>
<td>24</td>
</tr>
<tr>
<td>6: Performance Improvement over the baseline</td>
<td>30</td>
</tr>
<tr>
<td>7: automotive_bitcount: Random Vs. Genetic</td>
<td>32</td>
</tr>
<tr>
<td>8: Search Space Exploration bzip2e: Random Vs. Genetic</td>
<td>32</td>
</tr>
<tr>
<td>9: Network_dijkstra: Random Vs. Genetic</td>
<td>33</td>
</tr>
<tr>
<td>10: Security_blowfish_d: Random Vs. Genetic</td>
<td>33</td>
</tr>
<tr>
<td>11: telecom_CRC: Random Vs. Genetic</td>
<td>34</td>
</tr>
<tr>
<td>12: telecom_gsm: Random Vs. Genetic</td>
<td>34</td>
</tr>
</tbody>
</table>
CHAPTER 1: INTRODUCTION

Over the last several decades we have witnessed tremendous change in the landscape of computer architecture. New architectures have emerged at a rapid pace with greater computing capabilities that have often exceeded our expectations. However, the rapid rate of architectural innovations has also been a source of major concern for the high-performance computing community. Each new architecture or even a new model of a given architecture has brought with it new features that have added to the complexity of the target platform. As a result, it has become increasingly difficult to exploit the full potential of modern architectures for complex scientific applications. It often requires many person-months to tailor large applications to microprocessor-based parallel systems to achieve a high fraction of peak performance. Manual tuning of applications not only creates problems with code maintainability and verification but also decreases programmer productivity.

The emergence of multicore processors has greatly amplified this problem. Although chip multiprocessors have dramatically increased the performance potential of computer systems, much of the responsibility of harnessing this potential falls on software. In particular, software needs to play a bigger role in discovering hidden parallelism, automatically transforming code for parallel execution and perhaps most importantly,
exploit parallelism and data locality at multiple levels. Hence, it is likely that in the future even more time and effort will need to be devoted into manual performance tuning. To address this problem, several research groups have proposed methods and techniques for automatic performance tuning [3,12,16,17,19,30,31,38,41,44,49,51]. In an autotuning framework, a code is analyzed, alternate code variants are generated with tuning parameters and then a heuristic search is performed based on execution time feedback to obtain an implementation that yields optimal or near-optimal performance for the target platform. All of these tasks are performed with minimal to no human intervention. Autotuned libraries for specific domains, such ATLAS [51] for linear algebra and SPIRAL [41] and FFTW [19] for signal processing have been quite successful. Several other research efforts are under way that aims to apply the ATLAS-model for general scientific codes [12,16]. Although many of these research endeavors hold great promise, for autotuning to gain widespread acceptance in the scientific computing community, several key challenges need to be overcome.

A main impediment to practical and efficient automatic tuning is the enormous search space of transformation sequences and parameters containing billions of feasible points. Over the years compiler writers have developed a rich array of program transformations, which restructure programs in complex ways. There is strong interaction between many of these transformations and most of them are sensitive to underlying hardware parameters. Some transformations are also sensitive to the input data set of an application. All of these factors combine to give rise to a large and complex multi-dimensional search space. Hence, for autotuning to be more practical, we need a strategy
that can both cut down the search space to manageable proportions and apply advanced search heuristics to navigate that space. Another limitation of most autotuning frameworks is that the search module is tightly coupled with the transformation engine within the framework. Thus, exploiting these tools for automatic search space generation and exploration becomes problematic.

This thesis addresses these two challenges and describes the design and implementation of PTUNE, a heuristic search-driven autotuning framework. PTUNE facilitates navigation of the optimization search space with greater flexibility and efficiency through a number of novel features. These features are highlighted below:

(1) Use of Genetic Algorithms: PTUNE implements a Genetic Algorithm to search the space of optimization phase-orderings. Prior to this work, genetic algorithms had only been used to explore a subspace of the optimization search space. We provide an intuitive way to represent optimization sequences as chromosomes in Genetic Algorithms and propose new strategies for mutation and crossover.

(2) Enhanced Feedback: The quality of feedback plays a key role in the success of any automatic tuning system. Focusing on one particular performance metric provides a myopic view of application performance and generally does not lead to effective code optimization. To be able to tailor codes to complex systems, the application developer needs to identify the causes behind the performance problems. Looking at a variety of performance metrics to identify performance bottlenecks is common practice in the world
of manual tuning. However, in the context of automatic tuning, its application is almost non-existent. Consequently, applying even the most intelligent search methods generally does not lead to the best program variant in any reasonable amount of time. Another issue with feedback in the context of autotuning is granularity. Most search-based systems use whole program execution time as the only feedback metric. Collecting performance measurements at the program level is usually sufficient when tackling small kernels, where one loop nest dominates the entire execution. However, for larger applications in which execution time is distributed over several loop nests, whole program granularity is no longer sufficient. This is particularly true when dealing with loop transformations. Loop transformations such as tiling, if applied to multiple loop nests within a program, can have widely varying effects on each of those nests. Thus, to accurately determine the effects of changing loop transformation parameters, the search algorithms require feedback at a finer granularity. Relying on whole program feedback generally leads to longer search times.

The search engine in PTUNE collects fine-grain feedback, through HPCToolkit [62] and PAPI [63], at procedure-, loop- and statement-levels. These feedback metrics are exploited in parallel search algorithms that explore multiple code regions concurrently, thus reducing overall tuning time. Additionally, a variety of performance metrics is used as a guiding heuristic to speed up steepest descent search methods.

(3) Tuning of multi-threaded kernels: By integrating OpenMP and MPI flags in the search space, PTUNE provides the ability to tune multi-threaded kernels. Given the
growing trend of placing an increasing number of cores per chip, the ability to tune multi-threaded kernels is of paramount importance. To our knowledge, no other framework explicitly provides this level of control with GCC.

(4) Tool Sharing: PTUNE is designed to work as an independent search engine and provides a search API that can be used by other autotuning frameworks. This modular design of the search engine facilitates greater tool integration within the autotuning community. In particular, PTUNE can be easily integrated into PERI and MILEPOST GCC, two of the most prominent autotuning frameworks. On the other hand, PTUNE leverages the complementary strengths of several open-source software including LoopTool and HPCToolkit, thus increasing their applicability.
CHAPTER 2: RELATED WORK

We divide the discussion of related work into four parts. The first section reviews related work in autotuned libraries, the second section discusses more general autotuning frameworks, the third section provides a review of code transformations commonly used in autotuning and finally, the fourth section briefly discusses some of the tuning tool that can be integrated with PTUNE.

2.1. Autotuned Libraries

A number of successful empirical tuning systems provide efficient library implementations for important scientific domains, such as those for dense and sparse linear algebra \([7,17,51]\), signal processing \([19,41]\) and tensor contraction \([6]\). Among these, ATLAS \([51]\), is the most widely used within the scientific community and has become the \textit{de facto} standard for evaluating other autotuning systems. The ATLAS-model has even found its way into commercial compilers in the form the Math Kernel Library (MKL) distributed with the Intel compiler suite \([25]\). ATLAS produces highly optimized linear algebra routines by probing the underlying hardware for platform specific information and using a global search to find the best transformation parameters, searching for these parameters one transformation at a time. The transformations considered by ATLAS include multi-level tiling, unroll-and-jam and pipeline scheduling.
Unlike ATLAS, the SPIRAL [41] and FLAME [7] projects have looked at the problem at a higher-level and concentrated more on the issue of algorithmic choice rather than exploring options of alternate implementations of the same algorithm. In the SPIRAL framework, signal-transform routines are expressed generates by mathematical formulas using a special purpose language [55] and a suitable implementation is chosen based on matrix factorization calculations and a simple sequential search. More recently, the Peta Bricks project has adopted the SPIRAL and FLAME approach for general algorithmic tuning [4].

2.2. Search-based Tuning of Applications

The success of automatically tuned domain specific libraries has sparked considerable interest in applying search-based methods for tuning general applications. Research efforts in whole application tuning can be broadly classified into two categories based on the parameter search space on which they operate. Several ongoing research projects tackle the phase-ordering problem using empirical methods [3,31,37,38,49]. That is, they aim to find the best sequence of transformations that minimizes some objective function such as execution time or power. On the other hand, some of the work in autotuning concentrates on finding the best parameter values for transformations that use numerical parameters [12,16,30,44]. More recent efforts strive to combine the two methods to provide a more unified solution that involve compile-time tuning with source-to-source transformations and runtime tuning and optimization [5]. Our proposed methods are aligned with this integrated approach of autotuning, since the enhancements to the feedback mechanism can be utilized for both offline and online search. Earlier work in
autotuning mainly focused on finding good search strategies or modifying existing ones to reduce tuning times. Genetic algorithms [3,31], stochastic hill climbers [3] and greedy constructive algorithms [3] have been used to explore the search space of optimizations sequences.

In terms of the search space of numerical parameters, there has been work in applying direct search methods [44,45], simulated annealing [20,28,42,45], pyramid search [29,42], window search [28,42], binary search [12] and random search [28,42]. However, none of the search strategies proved particularly effective in the context of autotuning; in most cases, yielding at most a 5% improvement over random search [42,59]. The limited success of search algorithms lead to research in model-based tuning where some form of analytical modeling or guidance is used to prune the search space, guide search heuristic or reduce time spent in program evaluation during tuning. The issue of model-guided tuning has been approached from several different angles. Most notable among these is the use of compiler-based analytical models in limiting the search space [13,31,44,48,58]. Chen et al. showed the analytical models can significantly cut down the search space for a set of transformations including tiling, loop interchange and unroll-and-jam [12]. Qasem and Kennedy have used models for pruning the combined search space of loop fusion and tiling [44]. The Active Harmony project focuses on runtime optimizations and use analytical models to establish an ordering of transformations rather than reducing the size of the search space [23]. The OSE compiler uses static heuristics for generating a pruned search space for optimization sequences [48].
Kulkarni et al. use techniques such as detecting redundant sequences and identifying equivalent code to cut down the number of program evaluations [31]. Apart from compiler models, machine learning techniques have been applied to tune unroll factors [47] and also for selecting the best optimization set (without re-ordering) [1]. There has been some work in using statistical models to explore the search space of optimization parameters. Vuduc et al. establish early stopping criteria to eliminate less promising search space regions on-the-fly [50]. Pinkers et al. use a statistical method based on orthogonal arrays to choose the optimal sequence of transformations [40]. To avoid running the target program in every search step and save tuning time, some autotuning systems employ static performance estimators to predict the best program variant. This strategy has been proven effective in cutting down tuning time for the OSE compiler [48] and the PEAK tuning system [39]. However, as one would expect, using static estimators inevitably leads to some sacrifice in performance. More recent work, advocates using check pointing to terminate execution of long running loops to save tuning time [5].

Although none of the above mentioned model-guided strategies proved to be the holy grail for autotuning, it is generally agreed that some form of modeling or guidance is required to realize the long term vision of autotuning. Our framework does not aim to subsume any of the approaches mentioned above, rather it complements most of these strategies. The use of enhanced feedback in the search algorithm can be used in conjunction with existing pruning strategies to speed up the overall tuning process. Moreover, detailed feedback methods can also be used to validate or verify static estimators, and statistical or analytical models.
2.3. Tunable Transformations

In a world of massively-parallel applications and high availability of computational resources, data locality optimizations still play a crucial role in improving overall application performance. Not surprisingly, the literature on memory hierarchy transformations is vast and spans more than three decades. Here, we briefly discuss the work that is most relevant to the set of transformations supported by our framework. Most data locality optimizations aim to improve the memory hierarchy performance by concentrating on the dominant loop nests in an application. Improved memory performance is achieved either by restructuring loop nests to improve temporal and spatial locality or by modifying the data layout in memory to avoid conflicts at different levels of the hierarchy.

LoopTool supports a wide array of loop nest and data layout transformations. Among the transformations currently supported are loop fusion [26,27,32], tiling [10,11,35,54], unroll-and-jam [2], loop unrolling, array contraction, iteration space splicing [43] and scalar replacement [8]. In addition, LoopTool also supports several enabling transformations such as loop alignment and targeted code motion. For fusion, tiling and unroll-and-jam, LoopTool also provides architecture-aware heuristics for selecting the best tile sizes, unroll factors and fusion configurations in an integrated fashion [42]. Although the literature is replete with heuristics for selecting tile sizes and choosing unroll factors [9,10,11,14,18,35,54], attempts at integrating all these transformations have been less common [43,46,53]. Song et al. [46] present a model that combines loop fusion, loop alignment and array contraction. In their model, the primary goal is reducing
bandwidth requirements by reducing the size of arrays. Although they apply conditions to check for excessive register pressure and cache capacity they do not address the issue of conflict misses.

Wolf et al. [53] describe a strategy that combines loop distribution, loop fusion, tiling and unrolling. Although they look at a larger class of transformations their model does not capture all of the interactions between loop fusion and tiling. In their model, the tiling decisions are made after the optimal loop structure has been determined through fusion and distribution. Thus, LoopTool’s ability of applying transformations in integrated fashion is one of its most useful features.

2.4. Component Tools

PTUNE utilizes the complementary strengths of several open-source tools. Each tool can play an important role in the overall effectiveness of our tuning strategy. As background to our proposed work, we provide a description for each tool summarizing their key features.

**HPCToolkit:** This tool suite consists of components for measuring the performance of fully-optimized executables generated by vendor compilers, analyzing application binaries to correlate measurements with program structure, and novel analysis techniques for pinpointing performance bottlenecks in parallel programs [34].
HPCToolkit uses hardware performance counters to measure a variety of performance metrics including number of cache misses at different levels, TLB misses and number of stalled cycles. This tool suite is also capable of providing synthetic metrics such as bandwidth consumption. Another key feature of HPCToolkit is its ability to measure performance measurements at a fine granularity. HPCToolkit provides file-, subroutine-, loop and statement-level performance metrics, which make it a very useful tool for both manual and automatic tuning of applications. HPCToolkit has been used for performance analysis and manual tuning of several large-scale scientific applications [33].

**LoopTool:** LoopTool is a source-to-source transformation tool that assists expert human programmers by transforming Fortran loop nests for high performance [43]. It enables application developers to apply a complex set of well-known loop transformations to improve data reuse at various levels of the memory hierarchy. The transformations supported by LoopTool include tiling, unroll-and-jam, multi-level loop fusion, array contraction, and iteration space splicing.

The key feature in LoopTool that makes it a suitable for use in an empirical tuning system, is its ability to provide fine-grain control over transformation parameters, through the use of source-level directives. This feature is extremely important for our research, since in order to exploit fine-grain feedback we need a transformation tool that exposes the control at a fine level. This level of fine-grain control over transformations is usually not available in commercial compilers and many of the source code restructuring tools used in autotuning systems. For example, MIPSPro allows a user-specified tile size, but
applies it to every loop nest in the compilation unit. Loop-level optimization parameters cannot be specified at the command-line in any useful way. To specify an unroll factor for a particular loop at the command-line, the user would need to specify the index of the loop in lexical order and also its nesting depth. Specifying unique parameters for multiple optimizations and multiple loops would require the user to input a long complicated string that the compiler would then need to parse. Thus, the use of source directives in LoopTool provides a novel and useful way of specifying optimization parameters at loop-level granularity.
CHAPTER 3: TUNING FRAMEWORK

3.1 Overview

Fig. 1 provides an overview of the PTUNE framework and shows it’s interconnections with other tuning tools. The major components of the framework include a source-to-source transformer (LoopTool), a set of performance measurement tools (HPCToolkit),
and the search module that uses the measurements to guide selection of program transformations. At each step in the tuning process, the search module generates a set of optimization parameters that are applied to the input program by LoopTool. The program is then compiled using the native compiler and run on the target machine. During program execution, performance measurement tools collect variety performance measurements to feed to the search module. The search module uses these metrics in combination with results from previous passes to generate the next set of tuning parameters. This process continues until some pre-specified optimization time limit is reached or the search algorithm converges to a local minimum. Although the structure of our autotuning framework is not dramatically different from that of other systems, there are several key ideas that make our framework unique. Unlike most other automatic tuning systems, our framework uses a full-scale dependence-based transformation tool, which enables us to verify the legality of complex loop transformations. Another unique feature of our system is the use of loop-level performance measurements and the application of transformations at loop-level granularity. In addition, the search module can operate on both the search space of parameterized transformations and the search space of transformation parameters. The rest of this section discusses the core components of our framework in some detail.

3.2 Search Space Specification

Input to PTUNE is a configuration file that describes the search space of optimization parameters. Fig. 2 shows an example configuration file. The syntax for describing a search space is fairly simple. Each line in the configuration file describes one search
dimension. A dimension can be one of three types: range (R), permutation (P) or enumerated (E). Range is used to specify numeric transformation parameters such as tile sizes and unroll factors. Permutation specifies a transformation sequence and is useful when searching for the best phase sequence. An enumerated type is a special case of the range type. It can be used to describe a dimension where only a subset of points is feasible within a given range. An example of an enumerated type is the prefetch distance in software prefetching. In addition, PTUNE supports inter-dimensional constraints for all three dimension types. For example, if the unroll factor of an inner loop needs to be smaller than the tile size of an outer loop then this constraint is specified using a simple inequality within the configuration file.

```
100    # maximum number of program evaluations
1      # number of dimensions in the search space
R 1 10 # range : 1 .. 10
P 4    # permutation : sequence length 4
E 2 8 10 # enumerated : two possible value 8 and 10
```

**Figure 2: Example configuration file for PTUNE**

For some optimizations all integer values within a specified range are not always legal. For example, the prefetch distance for software prefetching might be specified in multiples of the cache line size within a certain range. An enumerated type may be used
to describe this search dimension. The user may wish to provide additional information within the configuration file that may be useful to the search engine. One piece of information that can be supplied in the configuration file is an inter-dimensional constraint. For example, if the unroll factor of an inner loop needs to be always smaller than the tile size of an outer loop then this constraint is specified using a simple inequality within the configuration file. Information specific to a search algorithm is specified elsewhere. For example, for simulated annealing the alpha and beta factors for each dimension is specified in a separate file. The parameters for the search algorithm have been deliberately kept separate to make the search space representation more general. Both the configuration file and the search parameter file can be written by hand or automatically generated by a transformation engine. This feature facilitates the use of PTUNE with model-based search strategies.

3.3 Exposing Optimization Flags in the GNU Compiler Collection (GCC)

A main contribution of this thesis is the development of a software interface that allows us to utilize the GNU compilers within an autotuning framework. GCC supports over a hundred different code optimizations and this number is growing with each new release. Because the optimization set is large and certain optimization flags subsumes by others (e.g., enabling optimization flag i, might enable optimizations i, j and k), leveraging the code transformation framework within GCC for autotuning has been challenging. For instance, in GCC 4.1.2, both -O and -O1, although listed as separate flags perform exactly the same set of optimizations. Thus, inclusion of both –O and -O1 flags in the search space leads to unnecessary increase in search space dimensionality and can
potentially skew the search results. On the other hand, applying the –O3 flag enables a wide range of loop transformation in concert, making it difficult to evaluate the effects of individual transformations. To address this problem, we conducted a systematic study of the compiler optimizations in GCC, that enabled us to isolate the command-line flag for each optimization and also discard flags that are ineffective or subsumed by others. Table 1 lists the set of specific optimization flags that were selected from this study.

To allow tuning of these optimizations, we developed an interface that takes as input a bit-stream representing an optimization sequence and maps each bit in the stream to the corresponding optimization flag in GCC. All optimization flags are then concatenated and output as a command-line string that can be directly embedded into a Makefile. Fig. 3 shows an example of how a bit-stream is translated into a set of optimization flags in GCC.

Since our approach only uses the command-line interface and is independent of the underlying GCC implementation, it provides a flexible and extensible method of interfacing GCC not just with our tuning framework but other autotuning systems as well. In particular, it can be easily integrated into the PERI autotuning system [64].

**Table 1: gcc compiler optimizations**

<table>
<thead>
<tr>
<th>No.</th>
<th>Optimization Flags</th>
<th>No.</th>
<th>Optimization Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-fdefer-pop</td>
<td>27</td>
<td>-fgcse-lm</td>
</tr>
<tr>
<td>Table 1 Continued</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-fguess-branch-probability</td>
<td>28</td>
<td>-fexpensive-optimizations</td>
</tr>
<tr>
<td>3</td>
<td>-fcprop-registers</td>
<td>29</td>
<td>-fstrength-reduce</td>
</tr>
<tr>
<td>4</td>
<td>-floop-optimize</td>
<td>30</td>
<td>-frun-cse-after-loop</td>
</tr>
<tr>
<td>5</td>
<td>-fif-conversion</td>
<td>31</td>
<td>-frun-loop-opt</td>
</tr>
<tr>
<td>6</td>
<td>-fif-conversion2</td>
<td>32</td>
<td>-fcaller-saves</td>
</tr>
<tr>
<td>7</td>
<td>-ftree-ccep</td>
<td>33</td>
<td>-fpeephole2</td>
</tr>
<tr>
<td>8</td>
<td>-ftree-dce</td>
<td>34</td>
<td>-fschedule-insns</td>
</tr>
<tr>
<td>9</td>
<td>-ftree-dominator-opts</td>
<td>35</td>
<td>-fschedule-insns2</td>
</tr>
<tr>
<td>10</td>
<td>-ftree-dse</td>
<td>36</td>
<td>-fsched-interblock</td>
</tr>
<tr>
<td>11</td>
<td>-ftree-ter</td>
<td>37</td>
<td>-fsched-spec</td>
</tr>
<tr>
<td>12</td>
<td>-ftree-lrs</td>
<td>38</td>
<td>-fregmove</td>
</tr>
<tr>
<td>13</td>
<td>-ftree-sra</td>
<td>39</td>
<td>-fstrict-aliasing</td>
</tr>
<tr>
<td>14</td>
<td>-ftree-copyrename</td>
<td>40</td>
<td>-fdelete-null-pointer-checks</td>
</tr>
<tr>
<td>15</td>
<td>-ftree-fre</td>
<td>41</td>
<td>-freorder-blocks</td>
</tr>
<tr>
<td>16</td>
<td>-ftree-ch</td>
<td>42</td>
<td>-freorder-functions</td>
</tr>
<tr>
<td>17</td>
<td>-funit-at-a-time</td>
<td>43</td>
<td>-falign-functions</td>
</tr>
<tr>
<td>18</td>
<td>-fmerge-constants</td>
<td>44</td>
<td>-falign-jumps</td>
</tr>
<tr>
<td>19</td>
<td>-fomit-frame-pointer</td>
<td>45</td>
<td>-falign-loops</td>
</tr>
<tr>
<td>20</td>
<td>-ftree-sra</td>
<td>46</td>
<td>-falign-labels</td>
</tr>
<tr>
<td>21</td>
<td>-fthread-jumps</td>
<td>47</td>
<td>-ftree-vrp</td>
</tr>
<tr>
<td>22</td>
<td>-fcrossjumping</td>
<td>48</td>
<td>-ftree-pre</td>
</tr>
<tr>
<td>23</td>
<td>-foptimize-sibling-calls</td>
<td>49</td>
<td>-finline-functions</td>
</tr>
<tr>
<td>24</td>
<td>-fcse-follow-jumps</td>
<td>50</td>
<td>-funswitch-loops</td>
</tr>
<tr>
<td>25</td>
<td>-fcse-skip-blocks</td>
<td>51</td>
<td>-fgcse-after-reload</td>
</tr>
<tr>
<td>26</td>
<td>-fgcse</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Specifying flags in PTUNE:

**Figure 3: Specifying Flags**

E.g. gcc -O0 -c -ftree-dse -ftree-ter -ftree-lrs -ftree-sra …… -ftree-copyrename foo.c

### 3.4 Search Algorithms

PTUNE implements a number of search strategies including genetic algorithm, direct search, window search, taboo search, simulated annealing and random search. This section provides brief descriptions of each implemented strategy.
3.4.1 Genetic Search

Our genetic search algorithm (GA) finds better sequence of optimizations for execution time. It uses the previous set of transformations to find out next set of transformations to apply. It finds the best possible optimization by merging part of the previous optimization sequences and changing few flags randomly. This is a good fit algorithm in iterative self-tuning and finds a better sequence in fair amount of time [60].

We compare the final optimization sequence we get from Genetic algorithm with three searches: Random Search, Direct Search and Simulated Anneal search and we found that Genetic search is doing better than all the others. See the experimental section for the results.

In our framework, we have 51 optimizations and an optimization is analogous to chromosome. So, the length of chromosome is 51. Each chromosome is either 0 or 1. Chromosome 1 means the corresponding optimization is ON and if it is 0 then it is OFF.

Population is number of sets of chromosomes. Initial solutions are randomly selected to form an initial population. The default population size in PTUNE is 5. The user can specify the population size through command line. Based on the population, compile the program is compiled and run with the transformations and execution time is measured. Fitness value is execution time of the program based on the population. The population with the highest fitness value, which is the maximum execution time, has the worst performance and the population with the lowest fitness value has the best performance.
The population is sorted in ascending order (best one in 0\textsuperscript{th} position and worst at the 4\textsuperscript{th} position) based on the fitness values.

Merging previous population and randomly changing a bit from each population generate reproduction of population for the next step. Ignore last population from the previous sequence and divide other each population in two half. For first two populations, we replace second half of first population with the first half of second population. We do same thing for the third and fourth population. Then we change one bit from each one randomly. Randomly generated new population replaces the fifth population, which has the highest fitness value.

Each generation in the experiments consists of the following steps

**Step 1: Initial Generation**

User can select the number of populations. The default population size is five. We use random generator to generate five random numbers. We pass those five random numbers to a function, which converts those random numbers into binary sequence. The length of each generated binary sequence is 51, which is equal to length of chromosomes.
Figure 4: Randomly Generated Population

**Step 2:** Rearranging the population

Each population is converted into sequence of optimizations based on whether it is 0 or 1. The code being compiled is passed through the compiler and the code is optimized based on the sequence defined by population. The corresponding fitness value is stored in an array. The population with the highest fitness value is considered as worst performance and vice versa. The population is stored in the ascending order based on the fitness value stored in the array. The lowest fitness value is stored at the 0th position and highest execution time in 4th location of the array. We have comparatively best population in population #1 and worst population in population #5.

**Step 3:** New Generation

We generate the new population by exchanging the chromosomes from the previous populations. Also, we change a bit from each new population randomly. We divide each
population in two halves. We exchange the second half of first population with the first half of second population. Do the same thing for third and fourth population.

One population selected randomly and replaced with the fifth population. The exchange of chromosomes is shown by arrow in the following figures.

**Population 1:**

```
0 1 0 1 1 1 0 0 1
```

**Population 2:**

```
1 1 0 1 0 0 0 0 0
```

**Population 3:**

```
1 1 0 0 1 1 1 1 1
```

**Population 4:**

```
1 0 1 0 0 1 0 1 1
```

**Population 5:**

```
1 0 0 0 0 0 1 1 1
```

**Figure 5: Generating New Population**

**Step 4:** Repeat the same procedure for number of iterations.

**Step 5:** Done.

### 3.4.2. Random Search

We include random in our framework as a benchmark search strategy. A search algorithm is considered effective only if it does better than random on a given search space. The chief reason for including a random search in our framework is for us to be able to compare the performance of difference search strategies. A random search picks random points within the search space and keeps track of the best value found at every step.
Unlike the other search strategies described above, random search does not use any heuristics and it does not have any convergence criteria. The search is terminated after a pre-specified number of evaluations.

3.4.3 Direct Search

There are two main flavors of direct search that have been used for exploring the optimization search space. The simplex method is usually applied for a continuous search space, whereas the pattern-based method is used for discrete search spaces. Since the search space of transformation parameters is discrete we implemented the pattern-based direct search method in our framework.

The variant of direct search implemented in our framework is the pattern-based method, originally proposed by Hooke and Jeeves [24]. This algorithm works on a discrete space and aims to find the optimal point in the search space using a method of steepest descent. The algorithm proceeds by making a set of exploratory moves and pattern moves. By visiting neighboring locations, the exploratory moves identify a promising direction of movement from the current position. Once this direction has been identified, the search takes a leap in that direction (pattern move) and then explores neighbors of that new location. This process continues until the exploratory moves fail to find a new promising direction for some point. This point is returned as the optimal location.

The major steps of the algorithm are sketched below:
• N denotes an n-dimensional search space, where each dimension represents a transformation parameter that is being tuned

• p = (p1, p2, ...pn) denotes a point in the search space where pi is the value of the its parameter

• f (p1, p2, ...pn) denotes the execution time for the program compiled with transformation parameters p1, p2, ...pn

• s denotes the step size, this value determines the size of the subspace that is explored during the exploratory moves

Step 1: Pick an initial base point p. This is done by choosing the midpoint within the range for each parameter.

Step 2: Make exploratory moves. For each parameter pi we first increment its value by step size s and evaluate the program at p'(p1, ...pi +s, ..., pn). If the execution time at p' is less than the current minimum then we set the value of parameter pi to (pi+s) and move on to the next parameter. Otherwise we decrement the value of the parameter by s and evaluate the program at p'(p1, ...pi − s, ..., pn). If f(p') is less than the current minimum then we set the value of parameter pi to (pi − s). Otherwise the value of the parameter remains unchanged. Once all the parameters have been explored, we move to Step 3.

Step 3: Make pattern move. The series of exploratory moves gives us a new point p’ in N where we are likely to find a value that is less than the current minimum. The pattern move moves the base point in the direction of p', that is p  p' − p. The execution time at
this new point is evaluated. If this execution time is less than the current base point execution time then we go to Step 2. Otherwise we move to Step 4.

**Step 4:** Reduce step size. If we have reached the minimum step size then we move to Step 5. Otherwise, we reduce the step size by the step size reduction factor and go back to Step 2.

**Step 5:** Done.

### 3.4.4 Simulated Annealing

Initially a random point is selected in the search space and its neighboring points are explored. At each step, the search moves to a point with the lowest value or depending on the current temperature, to a point with a higher value. The temperature is decreased over time and the search converges when no further moves are possible.
CHAPTER 4: EXPERIMENTAL RESULTS

4.1 Experimental Setup:

For the experiments, we have used five cBench benchmark applications and one SPEC benchmark application. cBench Benchmark: It is a collection of open-source sequential programs with many datasets assembled by the cTuning community [61]. CBench is partially based on modified MiBench benchmark. cBench has been used for benchmarking simulators including GCC, LLVM, Intel, Open64, and Pathscale. We have selected five cBench benchmark applications for our experiment. Table 2 provides short description of each of the benchmarks.

SPEC CPU2006 is industry-standardized, CPU-intensive benchmark suite, emphasizing a system's processor, memory subsystem and compiler [62]. SPEC designed CPU2006 to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications. These benchmarks are provided as source code. We have used bzip2e application for our experiments.
### Table 2: Benchmark Applications and their Descriptions

<table>
<thead>
<tr>
<th>No.</th>
<th>Benchmark Applications</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Automotive_bitcount</td>
<td>print bit pattern of bytes formatted to string</td>
</tr>
<tr>
<td>2.</td>
<td>network_dijkstras</td>
<td>Shortest Path Algorithm</td>
</tr>
<tr>
<td>3.</td>
<td>network_blowfish</td>
<td>Symmetric block cipher with a variable length key</td>
</tr>
<tr>
<td>4.</td>
<td>telecom_CRC32</td>
<td>Telecommunication Application</td>
</tr>
<tr>
<td>5.</td>
<td>telecom_gsm</td>
<td>Telecommunication Application</td>
</tr>
<tr>
<td>6.</td>
<td>bzip2e</td>
<td>Compression Algorithm</td>
</tr>
</tbody>
</table>

We use core 2 duo as our main platform and the specifications are given in Table 3.

### Table 3: Platform Specifications

<table>
<thead>
<tr>
<th>No.</th>
<th>Platform</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Compiler</td>
<td>Gcc version 4.1.2</td>
</tr>
</tbody>
</table>
| 2.  | Architecture      | - Model: Intel i386, CPU E6550 @ 2.33 GHz  
                      | - Cache size: 4096 KB  
                      | - Address sizes: 36 bits physical, 48 bits virtual                      |
| 3.  | Operating System  | - Red Hat 4.1.2-33                                                       |
4.2 Overall Performance Improvement

We first consider the overall performance achieved on the selected benchmarks using our search-based strategy. Fig. 3 shows the speedup obtained using four different search strategies: Genetic Algorithm (GA), random search (random), Direct Search (direct), and Simulated Annealing (anneal). The speedup reported is the one achieved over the fully optimized variant generated by GCC and is computed using the following formula:

\[
\text{Speedup} = \frac{\text{Execution time for fully optimized code}}{\text{Execution time of tuned code}}
\]

To keep the comparisons fair, each search algorithm is initiated from a random point in the search space and are allowed to converge independently, with a maximum ceiling of 100 evaluations.
Overall, GA yields the best performance across all benchmarks, achieving as much as a 1.16 speedup for bzip2e and telecom and 1.09 speedup on average. Moreover, unlike other search methods, GA obtains at least some performance improvement for all applications. direct performs well on telecom and bticount but causes performance degradation on nDiskstra. Similarly anneal does well on sblowfish and telecom but fails on bitcount and bzip2e. Somewhat surprisingly, random performs quite well and delivers performance close to that obtained through GA. One reason for random producing better results than direct and anneal is that both these searches tended to converge to a local minima much sooner then the maximum allotment of 100 evaluations. Thus, these results reiterate the difficulties of using steepest descent methods for exploring the optimization search space.

4.3 Search Space Exploration

In this section, we examine the manner in which the search space is explored by GA and random. Fig. 4-9 show the best values found by GA and random after each program evaluation. As before both searches are terminated after 100 evaluations. The Y-axis in these figures indicates program execution times in seconds. Thus, a lower y-value indicates a better performance point.

For automotive_bitcount, GA finds the best value after 50 iterations and the best improvements in performance comes between iterations 30 and 50. The trend line for random reveals that only 4 probes among 100 leads to better execution time. These results indicate a relatively smoother search space for automotive_bitcount with fewer
peaks and troughs. Also, the results indicate that a forced termination of the search space at an earlier phase might be beneficial.

Figure 7: automative_bitcount: Random Vs. Genetic

For bzip2e, GA finds the best value after 60 iterations and the best improvements in performance comes between iterations 50 and 60. The graph indicates a relatively smoother search space for bzip2e with fewer peaks and troughs.

Figure 8: bzip2e: Random Vs. Genetic
For Network_dijkstras, GA finds the best value after 80 iterations and the best improvements in performance comes between iterations 70 and 80. The trend line for genetic reveals that it is finding the better performance gradually. These results indicate a relatively smoother search space for Network_dijkstras with fewer peaks and troughs. Also, the results indicate that a continuing exploration of the search space would have helped more to get the better performance.

![Network_dijkstra: Random Vs. Genetic](image)

**Figure 9: Network_dijkstra: Random Vs. Genetic**

For security_blowsfish, genetic search found the minimum value at 40\textsuperscript{th} iteration but Random search could not find the minimum value.

![security_blowsfish_d: Random Vs. Genetic](image)

**Figure 10: Security_blowsfish_d: Random Vs. Genetic**
For telecom_CRC, the graph shows how genetic search is finding the minimum value step by step.

Figure 11: telecom_CRC: Random Vs. Genetic

For telecom_GSM, Genetic search found the minimum value after 40 iteration but it took around 80 iterations for Random search to find the minima.

Figure 12: telecom_gsm: Random Vs. Genetic
CHAPTER 5: CONCLUSION

This thesis described the implementation and experimental results of genetic search in PTUNE, an automatic performance tuning tool. We provided a way to represent optimization sequences as chromosomes in Genetic Algorithms and proposed new strategies for mutation and crossover. We conducted experiments to demonstrate the effectiveness of genetic search algorithm and the tuning tool. We compared the results of genetic search with other search techniques: Random Search, Simulated Annealing Search, and Direct Search. The results show that the genetic search is doing better in improving application’s performance and tuning time than all the other searches.
REFERENCES


63. L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and
N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel
programs. Concurrency and Computation: Practice and Experience, 22(6):685–701,

64. D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting Performance Data with

Hovland, S. Moore, K. Seymour, J. Shin, A. Tiwari, S. Williams, and H. You. PERI
6596/125/1/012089.
Santosh Raosaheb Sarangkar was born in a farmer’s family in Rajuri, Maharashtra, India on December 25, 1982, the son of Raosaheb Babaji Sarangkar and Chanchal Raosaheb Sarangkar. He received the degree bachelor of engineering (B.E) in Electronics and Tele-Communication from Vidya Pratishthan College of Engineering Baramati, Maharashtra, India in 2006. He entered in Masters of Science (M.S) degree program at Texas State University- San Marcos in August 2008. During his masters he was employed as HPC programmer, research assistant, and lab instructor with Texas State University- San Marcos. Also, he received the awards graduate research excellence, graduate academic excellence, outstanding graduate student nomination, and excellence in service at Texas State University. During the summer of 2010 and the summer of 2011, he was employed as a summer research intern at Freescale Semiconductor Inc., Austin, Texas and National Center for Atmospheric Research (NCAR), Boulder, Colorado respectively.

Permanent Email Address: santosh.sarangkar@gmail.com

This thesis was typed by Santosh R. Sarangkar.