MATS : A Model-Driven Adaptive Tuning System for Parallel Workloads

Santosh Sarangkar¹, Apan Qasem²

¹Intel Corporation, Phoenix, AZ, USA
²Department of Computer Science, Texas State University, San Marcos, TX, USA

¹santoshx.sarangkar@intel.com; ²apan@txstate.edu

Abstract- Building software that can effectively utilize underlying hardware resources has been a perennial challenge for the high-performance computing community. In recent years, the HPC community has responded to this challenge by creating adaptive compilation systems that allow domain experts to automatically tune their code to different architectures; thus relieving some of the burden of manual re-targeting of code. Although autotuning of applications holds great promise, the enormity and complexity of the optimization search space and the associated long tuning times, coupled with the emergence of multicore and manycore architectures pose significant challenges in its acceptance as a viable alternative to manual performance tuning. This paper describes the design and implementation of a model-driven, extensible autotuning framework that addresses current challenges in automatic performance tuning. The novel contributions of this framework include a mechanism for exposing tunable parameters for parallelism; a model-based search space pruning strategy based on the notion of tolerance values and the integration of multi-level code transformations within a single unified framework. With experimental evaluation on four different platforms, we demonstrate how these elements come together and facilitate efficient exploration of the multi-dimensional search space to obtain integer factor performance improvements on a range of scientific and engineering parallel applications.

Keywords- Performance Tuning; Intelligent Search; Compiler Optimization; Parallelization

I. INTRODUCTION

As computer architectures and scientific applications both grow in scale and complexity, achieving a high fraction of peak performance becomes more and more challenging. The complexity of the underlying hardware often renders traditional compilation techniques ineffective, leaving scientists and application programmers with the task of manually tailoring their code to achieve the desired performance. Manual tuning is not only a tremendous resource sink for scientific computing, a single application often requiring several person-months but the process itself is tedious and error-prone, making code maintainability and verification a nightmare. The emergence of multicore processors and accelerators has greatly exacerbated this problem. Although chip multiprocessors have dramatically increased the performance potential of computer systems, it is widely agreed that much of the responsibility of harnessing this potential will fall 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.

In response to this daunting challenge, recent research has focused on techniques for automatic performance tuning or adaptive compilation [1–13]. 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, resulting in efficient portable code. Although many of the 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 managing 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 these factors combine to give rise to a large and complex multi-dimensional search space. Furthermore, to achieve portable performance on current chip multiprocessor systems, parameters related to parallelism must be included in the search space, which further increases its size and complexity.

This paper describes the design and implementation of MATS, a model-driven extensible autotuning framework that aims to alleviate the challenges mentioned above. We address these challenges in several ways. First, we take a systematic approach to identifying parameters related to parallelism, analyzing the source to extract those parameters and exposing them to the search engine for tuning. Specifically, MATS explores parameters related to thread affinity and granularity that have not been explored by other adaptive tuning systems [1–3, 6]. Second, to tackle the exponential growth of the optimization search, our framework employs a novel search space pruning strategy based on a set of architecture-sensitive cost models. The key idea behind this pruning strategy is to identify critical architectural resources that affect the profitability of multiple code transformations, build models to estimate
their effective capacity and then apply them as constraints during search to reduce the dimensionality of the space and the range of values along each dimension. This approach not only causes a significant reduction in the search space but also accounts for complex interaction between transformations. The third unique feature of our framework is the integration of both high-level and low-level optimizations into a single integrated search space. To date, most autotuning systems have either focused on solving the phase-ordering problem of compiler optimizations \cite{1, 6, 9} or addressed the issue of selecting suitable parameters for a set of high-level transformations \cite{2, 3, 7, 12}. In MATS, we combine these two approaches to obtain better performance improvements for programs that benefit from both classes of optimizations. In addition to the three features mentioned above, MATS includes several strategies that make it a suitable tool for autotuning. Among these are the use of hardware performance counters for collecting fine-grained feedback during search; implementation of an extensive set of heuristic search algorithms and support for offline tuning through the use of machine learning algorithms. To summarize, the key contributions of this work are as follows:

- a methodology for identifying and exposing tunable parameters that affect a program’s parallel performance;
- a pruning technique that uses architecture-sensitive cost models and the notion of tolerance terms to reduce search space dimensionality;
- a search space representation framework that allows tuning of both high-level and low-level code transformations in an integrated fashion.

The rest of the paper is organized as follows. Section II describes related work. Section III gives an overview of the tuning framework. Sections IV-VI describe the three major contributions of MATS. Experimental results are presented in Section VII. Finally, we conclude and discuss future work in Section VIII.

II. RELATED WORK

A number of successful adaptive tuning systems provide efficient library implementations for important scientific domains, such as those for dense and sparse linear algebra \cite{4, 13, 14}, signal processing \cite{5, 11} and tensor contraction \cite{15}. Among these, ATLAS \cite{13}, is the most widely used within the scientific community and has become the de facto standard for evaluating other autotuning systems. The ATLAS-model has even found its way into commercial compilers in the form of Intel’s Math Kernel Library (MKL) \cite{16}. 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, SPIRAL \cite{11} and FLAME \cite{14} 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 SPIRAL, signal-transform routines are expressed by mathematical formulas using a special purpose language \cite{17} and a suitable implementation is chosen based on matrix factorization calculations and a simple sequential search. More recently, the PetaBricks project has adopted the SPIRAL and FLAME approach for general algorithmic tuning \cite{18}.

The success of automatically tuned domain specific libraries 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 \cite{6, 9, 10}. 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 \cite{2, 3, 7, 12}. More recent efforts strive to combine the two methods to provide a more unified solution that involves compile-time tuning with source-to-source transformations and runtime tuning and optimization \cite{20}.

Earlier work in autotuning mainly focused on finding good search strategies or modifying existing ones to reduce tuning times. Genetic algorithms \cite{1, 6, 16}, stochastic hill climbers \cite{1} and greedy constructive algorithms \cite{1} have been used to explore the search space of optimizations sequences. For the search space of numerical parameters, there has been work in applying direct search methods \cite{12, 21}, simulated annealing \cite{21, 24, 25, 26}, pyramid search \cite{22, 28}, window search \cite{22, 23}, binary search \cite{22} and random search \cite{22, 26}. None of the search strategies, however, proved particularly effective, in most cases, yielding at most a 5% improvement over random search. \cite{22, 26}. The limited success of search algorithms led 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 \cite{6, 12, 27-29}. Chen et al. show that analytical models can significantly cut down the search space for a set of transformations including tiling, loop interchange and unroll-and-jam \cite{21}. Qasem and Kennedy have used models for pruning the combined search space of loop fusion and tiling \cite{12}. 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 \cite{20}. The OSE compiler uses static heuristics for generating a pruned search space for optimization sequences \cite{28}. Kulkarni et al. use techniques such as detecting redundant sequences and identifying equivalent code to cut down the number of program evaluations \cite{6}. Apart from compiler models, machine learning techniques have been applied to tune unroll factors \cite{30} and also for selecting the best optimization set (without reordering) \cite{32}. More recently, reinforcement learning techniques have been used for dynamic scheduling of parallel loops \cite{33}. The C-tuning project has applied machine learning strategies for
selecting the best optimization sequence in the GCC compiler [34]. There has also been some work in using statistical models to explore the search space of optimization parameters. Vuduc et al. establish an early stopping criterion to eliminate less promising search space regions on-the-fly [35]. Pinkers et al. use a statistical method based on orthogonal arrays to choose the optimal sequence of transformations [36].

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 autotuning framework, in which we employ analytic modeling to reduce the search space, is aligned with this goal. Unlike previous approaches, however, we utilize the notion of tolerance in conjunction with architecture-aware cost models, which allows our framework to not only reduce the search space but also collapse multiple search dimensions and perform a non-orthogonal search. Furthermore, our framework exposes thread granularity and affinity as tunable parameters, an aspect of the search space not explored by earlier work. Lastly, MATS integrates both high-level and low-level optimizations in a unified search space. In this regard, our work can be thought of as combining two divergent threads in autotuning research.

III. OVERVIEW OF FRAMEWORK

Fig. 1 provides an overview of the MATS framework. The system comprises of four major components: a source-to-source code restructurer (CREST), a program analyzer and feature extractor, a set of performance measurement tools, and a search engine for exploring the optimization search space. MATS can operate in two modes: offline and online. In the online mode, the source code is first fed into the program analyzer, which utilizes program characteristics and architectural profiles gathered at install time to generate a representative search space. The search engine is then invoked, which selects the next search point to be evaluated. Depending on the configuration, the search point gets translated to a set of parameters for high-level code optimizations or compiler flags for low-level optimizations or a combination of both. The program is then transformed and compiled with the specified optimizations and executed on the target platform. During program execution, performance measurement tools collect a variety of measurements to feed back 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 local minima. In the offline mode, the source program is fed into the feature extractor, which extracts the relevant features and then maps it to one of the pre-trained predictive models using a nearest neighbor algorithm. The predictive model emits the best optimization for the program based on its training. The gathering of training data and the training of the predictive models occurs during install time.

Next, we describe the four major components of the our adaptive tuning system and highlight their key features.

A. Program Analyzer and Feature Extractor

When a source program arrives in MATS, it is initially dispatched to a tool for program analysis and feature extraction. The program analyzer uses a dependence-based framework to determine the safety of all high-level code transformations that may be applied at a later phase by the code restructurer. The analyzer incorporates models that aim to determine the profitability of each optimization and based on these models, it creates the initial search space. The analyzer also contains the analysis needed to generate the pruned search space of architectural parameters, as explained in Section 7. Furthermore, at this stage, hot code segments are identified to focus the search to code regions that dominate the total execution time.

The feature extraction module builds and analyzes several program representations including data and control flow graphs, SSA and def-use chains. A variety of information that characterizes a program’s behavior is extracted from these structures. Table I lists some example features collected by our tool. A total of 74 different features are collected for each program. The extracted features are encoded in an integer vector and stored in a database for use by the machine learning algorithms.

![Overview of framework](image)
TABLE I SAMPLE OF EXTRACTED PROGRAM FEATURES

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of basic blocks</td>
<td></td>
</tr>
<tr>
<td>Number of conditionals in [function, loop-nest]</td>
<td></td>
</tr>
<tr>
<td>Number of conditionals in each loop nest</td>
<td></td>
</tr>
<tr>
<td>Number of edges in the control-flow graph</td>
<td></td>
</tr>
<tr>
<td>Number of critical edges in the control-flow graph</td>
<td></td>
</tr>
<tr>
<td>Number of array references at a given level in the loop-nest</td>
<td></td>
</tr>
<tr>
<td>Number of scalar references at a given level in the loop-nest</td>
<td></td>
</tr>
<tr>
<td>Number of floating-point operations in [function, loop-nest]</td>
<td></td>
</tr>
</tbody>
</table>

B. CREST: Source-to-Source Code Restructurer

To provide greater flexibility and power in autotuning HPC applications, we developed a source-to-source code-restructuring tool, CREST that is capable of performing a range of complex loop transformations to improve data reuse at various levels of the memory hierarchy. One attribute that distinguishes CREST from other restructuring tools is that it allows retargeting of memory hierarchy transformations to control thread granularity and affinity for programs parallelized with OpenMP directives. For example, tile size and shape can be used to determine the number of required synchronizations and the amount of work done per thread. Some of the transformations currently supported by CREST include tiling, unroll-and-jam, multi-level loop fusion and distribution, array contraction, array padding, and iteration space splicing.

```
cdir$ fuse 1
cdir$ uj 4
do j = 1, N
cdir$ block 16
do i = 1, M
cdir$ block 16
do k = 1, L
  a(k,i,j) = b(j,i) + 17

cdir$ fuse 1
do j = 1, N
do i = 1, M
do k = 1, L
c(k,i,j) = a(k,i,j) + 1
```

Fig. 2 Source-level directives in CREST

Another feature in CREST that makes it suitable for use in an autotuning system is its ability to provide fine-grain control over transformation parameters, through the use of source-level directives. Fig. 2 shows an example of directives embedded in Fortran source code. A directive is simply a comment line that specifies a particular transformation and one or more optional parameter values, which can be associated with any function, loop, or data structure. This implies that CREST can apply transformations at program, procedure or loop-level; a level of control that is usually not available in commercial compilers or other source code restructuring tools used in autotuning systems. For example, GCC does not allow the user to specify a tile size as a command-line option whereas Intel’s icc allows a user-specified tile size, but applies it to every loop nest in the compilation unit. Thus, the use of source directives in CREST provides a novel and useful way of specifying optimization parameters at loop-level granularity.

C. Feedback Parser and Synthesizer

MATS utilizes HPCToolkit [37] and PAPI [38] to probe hardware performance counters and collect a wide range of performance metrics during each run of the input program. Measurements collected in addition to the program execution time include number of cache misses at different levels, TLB misses and number of stalled cycles. HPCToolkit supports measuring performance of fully optimized executables generated by vendor compilers and then correlating these measurements with program structures such as procedures, loops and statements. We leverage HPCToolkit’s ability to provide fine-grain feedback with CREST’s fine-grain control over transformations to create search instances that can explore independent code regions of an application, concurrently. To use HPCToolkit in MATS, we developed FeedSynth, a feedback parser and synthesizer that parses the performance output produced by HPCToolkit and delivers it to the search engine.

D. Search Engine

As mentioned, our framework supports both offline and online tuning. The search engine implements a number of online search algorithms including genetic algorithm, direct search [39], window search, taboo search, simulated annealing [40] and random search. Apart from random search, all other algorithms are multidimensional in nature, which facilitates non-orthogonal exploration of the search space. We include random search in our framework as a benchmark for other search algorithms. A search is considered effective only if it performs better than random on a given search space.

For offline tuning, we implement several supervised and unsupervised learning algorithms. The implemented algorithms include support vector machines (SVM), k-nearest neighbor, k-means clustering and principal component analysis (PCA). Additionally, to support these learning algorithms, we implement three statistical predictive models including independent and identically distributed model (IID), Markov chain and logistic regression.

IV. EXPOSING TUNABLE PARAMETERS FOR PARALLELISM

A unique feature of MATS is its ability to expose tuning parameters that influence an application’s parallel performance. Although many of the factors that impact parallel performance are known to experts, restructuring a parallel application to generate variants that conform to a specific set of parameters, is a significant challenge. Our framework addresses this particular issue. Using a combination of source code instrumentation and detailed analysis of dependence patterns in the program, MATS is able to automatically restructure source code to generate alternate parallel variants. The main novelty of the approach
is the application of a set of carefully selected high-level code transformations in controlling the amount of work done per thread. MATS supports several tunable parameters for parallelism including thread count, affinity and granularity. In this section, we describe the methodology for extracting and exposing these parameters for tuning. Our strategy is mainly centered on tuning OpenMP applications and therefore, the following discussion focuses on OpenMP. However, the strategy can be directly applied to expose parameters such as affinity and granularity for Pthreads and MPI programs as well.

A. Thread Count and Scheduling

The number of threads used to execute a parallel region can have a significant impact on performance. Generally, setting the number of threads to be equal to the number of available cores works reasonably well. Nevertheless, factors such as system load, OS scheduling heuristic and the presence of hyperthreading can make this simple strategy ineffective. Thus, for portable parallel performance on current architectures, thread count is a parameter that deserves tuning.

Our framework supports tuning of thread count for programs parallelized using OpenMP. For each parallel region, we insert the num_threads(n) clause with a dummy parameter (if the clause is not present). When a new optimization configuration is received from the search engine, the parameter values corresponding to thread count are extracted and used to replace each of the dummy parameters inserted earlier. MATS also supports a mode where the number of threads for an entire program is controlled using a call to the OpenMP, omp_set_num_threads(n) function.

A similar strategy is used to select the scheduling policy. All supported scheduling policies in OpenMP (e.g., CYCLIC, BLOCK etc.) are enumerated and initially, for each parallel region, the scheduling directive is set to the default value. Once a sequence is received, the default value is replaced with the parameter specified in the sequence.

B. Thread Affinity

On modern CMP architectures, hardware resources are shared by co-running threads at different levels. Thus, it is important for scheduling policies to be topology-aware and consider not only when a thread should be scheduled but also where it should be scheduled. For example, if two threads that exhibit good inter-core locality are mapped to two cores that share a cache, the performance is likely to be better than if they are scheduled on two threads that do not share a cache. Our framework facilitates tuning for thread affinity by utilizing the OpenMP environment variable GOMP_CPU_AFFINITY (and KMP_AFFINITY for Intel’s icc compiler) to explicitly set the affinity of each thread in a parallel application. For a given program-platform pair, we support tuning of all possible affinity configurations. That is, the affinity of any thread, \( t \), may be set to any available core, \( c \). Thus, for a program with \( T \) threads and \( C \) cores, the number of supported configurations is \( C^T \). When tuning an application, all feasible affinity configurations are enumerated and represented using a single integer value in the search space. When a new search configuration is generated, the affinity value is extracted and mapped to the appropriate configuration. This configuration is then translated to conform to the syntax of the proc-list parameter associated with the GOMP_CPU_AFFINITY variable. Finally, a system call is emitted to set the appropriate value before program invocation.

C. Thread Granularity

Determining the right granularity of threads is one of the key factors in achieving good parallel performance. Finer granularity can improve performance by allowing more flexibility in terms of scheduling of threads. On the other hand, a coarse-grained decomposition might benefit from improved locality in shared caches. We introduce a novel strategy for parameterizing a parallel application and controlling its thread granularity. In our approach, applications with parallel loop-nests are automatically transformed to produce variants with different granularity of parallelism. This transformation is achieved through a set of high-level code optimizations that impact the granularity of parallel code. Table II lists the main transformations employed and their effect on thread granularity. Below we discuss the transformation framework and describe the analysis used to attain different amounts of granularity using these code transformations.

<table>
<thead>
<tr>
<th>Code Transformation</th>
<th>Coarse?</th>
<th>Finer?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Selection</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Strip-mine</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Loop Interchange</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Loop Fusion</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Loop Distribution</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

In our framework, we assume, we have a fully parallelizable \( m \)-dimensional loop nest \( L \). The iteration space for \( L \) is denoted as follows

\[
I = \{(i_1, \ldots, i_m) \mid 1 \leq l \leq m: lb_1 \leq i_l \leq ub_l\}
\]

where \( i_l \) denotes a specific iteration, and \( lb_l \) and \( ub_l \) refer to the lower and upper bounds of the \( l \)-th loop in the nest. To reason about relative granularity of program variants and to facilitate search space construction, we assign a unit to the granularity of parallelism. We say that the granularity, \( \gamma \) of a parallel loop-nest \( L \) is one when each thread executes a single point in the iteration space. When \( \gamma = 1 \), total number of threads = \( |I| \) and the task performed by the \( k \)-th thread is denoted as

\[
t_k = \{(i_1, \ldots, i_m) \mid 1 \leq l \leq m: i_l = k\}
\]

In the other extreme, when \( L \) is parallelized at coarsest granularity, we have \( \gamma = |I| \), and there is only one thread \( t \) which executes the entire iteration space. Thus, the amount of work done by that thread is \( I \).

The goal of our transformation framework is to identify all feasible points of granularity between \( I \) and \( |I| \) and
expose them as tunable parameters. Next, we show how the source-level transformations in our framework attain the various amounts of thread granularity.

1) Loop Selection:
In a fully parallelizable loop-nest, selecting a subset of loops to parallelize can impact granularity. To distinguish loop selection from loop interchange, we assume that when loop \( l_j \) in \( L \) is selected, all loops above \( j \) are also selected for parallelization. Thus, for an \( m \)-dimensional loop-nest we have \( m \) distinct configurations for loop selection. Now, when \( l_j \) is selected for parallelization, it implies that each iteration of \( l_j \) executes as a single thread (with no other specification). Thus, in this case the task performed by the \( k \)th thread is expressed as follows

\[
t_k = \{(i_1, \ldots, i_j, \ldots, i_m) \mid 1 \leq l \leq j-1 : lb_l \leq i_l \leq ub_l, \text{ and } j \leq p \leq m : i_p = k\}
\]

and we obtain the following formula for the granularity of \( L \),

\[
\gamma = \left( \frac{ub_l - lb_l}{N_t} \right) \times B
\]

Therefore, loop selection, guided by Eq. 1 can be utilized to obtain a granularity for \( L \), as specified by the search space configuration. Depending on the number of loops parallelized in the original loop-nest, loop selection can be used to both increase and decrease granularity.

2) Loop Interchange:
Loop interchange can affect thread granularity if at least one loop in the nest is sequential. Moving a parallel loop outwards, beyond a sequential loop increases granularity whereas moving it inwards decreases granularity. If \( l_i \) is sequential and \( l_j \) is parallel in \( L \), where \( i \leq j \) then interchanging \( l_j \) and \( l_i \) will increase the granularity for the threads that execute the body of \( l_i \). Let \( p \) be the number of threads used to execute the iterations of \( l_j \) in the interchanged loop-nest. Then for these \( p \) threads the granularity can be obtained from Eq. 1. The rest of the threads will still execute with granularity one. Thus, in this case, average thread granularity of the loop-nest is increased. If we consider the reverse transformation, where a sequential loop is pushed inwards, the granularity of the resulting loop-nest will decrease.

3) Strip-Mine:
Strip-mining can be used to group iterations of a loop into strips or blocks, where each block is executed by a single thread. Thus, this transformation allows us to control the granularity of an individual loop within a nest. If in the original version of the code, each iteration of a loop executes as a separate thread then strip-mining will always increase granularity. If we strip-mine a loop at Level \( p \), by a factor of \( B \), then the number of threads needed to execute that particular loop is given by

\[
N_{t,b} = \frac{(ub_{p,j} - lb_{p,j})}{B}
\]

and the task performed by each thread can be expressed as follows

\[
t_k = \{(i_1, \ldots, i_j, \ldots, i_m) \mid 1 \leq l \leq j-1 : lb_l \leq i_l \leq ub_l, \text{and } q = j, \text{ and } i_j = lb_j + (k \times B) : jj \leq i_q \leq (jj + B - 1) \text{ and } 2 \leq p \leq m : i_p = k/(ub_p - lb_p)/N_t,b \} + 1
\]

Therefore, granularity of a loop-nest strip-mined by a factor of \( B \) can be expressed as follows

\[
\gamma = \left( \frac{ub_l - lb_l}{N_t,B} \right) \times B
\]

4) Loop Fusion and Distribution:
Fusing two loop-nests with iteration spaces, \( L_1 \) and \( L_2 \) results in a combined iteration space \( L \) which is expressed as follows

\[
L = \{(i_1, \ldots, i_m) \mid 1 \leq l \leq m : i_1 = <i_1 \text{, } i_2> \}
\]

Thus, when fusing \( k \) loop-nests at all levels, the granularity is reduced by a factor of \( k \). Therefore, when two loops are fused at Level \( l \), the granularity of the fused loop-nest is given by

\[
\gamma = \left( \frac{ub_l - lb_l}{N_t} \right) \times B
\]

The expression of loop distribution using an iteration space framework is fairly complex since individual statements within the loop body need to be accounted for. We observe, however, that the granularity of a distributed loop-nest can be derived without a formal analysis of the iteration space. Let \( d \) be the distribution factor and \( g \) denote the granularity of the \( i \)th loop in \( L \). Then, distributing a loop at level \( l \) by a factor of \( d \) gives us

\[
L = \{(i_1, \ldots, i_m) \mid d \}
\]

Eqs. 1-4 form the basis for parameterization of thread granularity in a parallel application. Each parallel loop-nest in the input program is analyzed and Eqs. 1-4 are used to determine a feasible search space of varying thread granularity. Overlapping points are eliminated and the granularity configurations are included in the main search space as a single dimension. During search, the value in the granularity dimension is interpreted and translated into some of subset of code transformations described above. The corresponding transformations are applied to the source code to obtain a variant with the desired granularity.

The goal of our transformation framework is to identify all feasible points of granularity between 1 and \( ||l|| \) and expose them as tunable parameters.

V. PRUNING THE SEARCH SPACE OF CODE OPTIMIZATIONS

MATS implements a novel technique for pruning the optimization search space. In our approach, we move away from the search space of parameterized transformations and instead focus on the search space of architecture-dependent
parameters embedded within compiler cost models. The strategy is based on the key insight that the best parameter values for many optimizations, particularly memory hierarchy transformations depend on estimates of certain architectural resources \[\text{[ref2, ref3]}\]. For example, tile sizes are constrained by the estimated capacity of the target cache, whereas profitable unroll factors may depend on the number of available registers. Generally, the quality of the parameter (i.e., how profitable the optimization is) depends on the accuracy of the resource estimates. Driven by this notion, in our framework, we construct a search space based on architectural resources and use search to find better estimates for those resources, which in turn deliver better optimization parameters. The search space generation and pruning strategy for two memory hierarchy transformations, loop fusion and tiling is depicted in Fig. 3. Our pruning strategy reduces the size of the search space in two ways. First, we can use a single parameter to capture the effects of multiple transformations, which reduces search space dimensionality and allows us to convert a non-orthogonal search space into an orthogonal one. For example, we can use the estimate of the cache size parameter to tune both loop fusion and tiling transformations. Second, for transformations that have different parameters for different loops (e.g., loop unrolling), we can again use just a single parameter to tune each of the loops in the program. Thus, the search space we explore does not grow with program size. For large applications with many loop-nests, this property can be very effective in limiting the size of the search space.

![Fig. 3 Search space pruning](image)

Our search space pruning strategy involves applying reuse analysis to determine cache footprint and locality in the source code, building models to estimate capacity of architectural resources, and parameterizing these models to make them suitable for search. Below we outline these steps.

A. Modelling Resource Constraints

The first step in our pruning strategy involves identifying key architectural resources that affect the profitability of the transformations in question. Once the architectural resources have been identified, we employ analytical models to estimate the amount of that resource that is available to the program. The amount of resource that can be exploited by a program is determined by a host of factors. For example, the fraction of cache we can exploit depends on the size and associativity of the cache, the number of different arrays accessed in the program and also the size of each of those arrays. For each resource \(R\), we construct a function that computes the effective size of \(R\). This function is denoted with \(E\) and expressed as follows

\[
R' = E(r_1, r_2, ..., r_n) \text{ s.t. } R' \leq R
\]

where \(r_1, r_2, ..., r_n\) are parameters that determine the effective size of \(R\).

The main challenge in incorporating these models in the framework is finding suitable parameterizations so that the relevant parameters can be exposed for search-based tuning. We accomplish this by introducing the notion of a tolerance term. We derive tolerance terms for each of the machine parameter estimates such that a relationship between the tolerance term and the architectural resource can be established. For example, using the Hill and Smith associativity model \[\text{[ref41]}\], we can derive the effective capacity of caches (and TLBs).

Assume, \(m_1\) and \(m_2\) are references to the same cache line. \(d\) is reuse distance between \(m_1\) and \(m_2\), \(s\) is the number of sets in cache, \(a\) is the associativity. Now, if each line from \(d\) is equally likely to be mapped to any of the sets and we introduce a tolerance term, \(T\) that expresses how high a probability of a conflict miss we are willing to accept, we have

\[
T \geq P[\text{conflict miss on } m_1] = 1 - \prod_{i=0}^{a-1} \left(1 - \frac{d}{s} \right)^i = \frac{1}{s} \sum_{i=0}^{d} \frac{1}{i!}
\]

(5)
For a given tolerance we can then solve Eq. 5 to determine the effective capacity of the cache in question. Thus, the effective capacity for a cache at level $k$ can be expressed as

$$E = (s_k, a_k, T)$$

where $s_k$ and $a_k$ are the size and associativity of the cache at level $k$ and $T$ is a tolerance term representing the conflict miss rate. The effective capacity models are built into our program analyzer and CREST and are invoked for each program that is tuned. The framework currently supports the following four resource constraints:

- **Effective Cache Capacity**: We use a probabilistic model to compute the effective cache capacity for all levels of cache. The effective cache capacity is used for tuning parameters for tiling, fusion, and loop interchange.
- **Effective Register Set**: The effective register set and register pressure are estimated using the model proposed by Carr. This parameter is used for loop unrolling, loop fusion and unroll-and-jam.
- **Effective TLB**: We develop our own model for computing effective TLB size. The model follows the same idea proposed by Hill and Smith but is adjusted for page size rather than cache lines. This model is used for all memory hierarchy transformations implemented in our framework.
- **Effective Instruction Cache**: The instruction cache constraint is dealt separately since we do not compute reuse distances for instructions and we are mainly concerned with capacity misses. We use the model proposed by Mueller et al. to derive an estimate of the instruction cache.

B. **Quantifying Memory Reuse**

When a new input program is received by the system, the source code is analyzed and the amount of reuse is quantified by the program analyzer. This reuse information is combined with architectural profiles to derive various estimates about program locality such as cache miss rates and number of register spills.

We make several standard assumptions about code shape in our reuse analysis framework. In our framework, a program is a collection of statements, each enclosed by zero or more loops. Loops are perfectly nested and loop bounds are affine expressions of loop iterators. All array references are considered to be uniformly generated. Without loss of generality, we assume that arrays are stored in column-major order. We assume loop bounds are known at compile time. Although this is not a realistic assumption for many applications, making such an assumption is not a problem in general. The issue of unknown loop bounds can be handled in several different ways. One approach is to simply assume loop bounds which are large enough so that we get no reuse at any of the outer levels.

C. **Constructing Search Spaces With Tolerance Values**

The Cartesian product of the sets of tolerance values forms the search subspace of memory hierarchy transformations. At each step in the search process, a tolerance value $t_i$ is chosen. The effective capacity of each architectural resource is derived based on the value of $t_i$. Once the effective capacities are formulated, each transformation is considered in turn and the optimization parameters are chosen so that maximum reuse is exploited based on the given constraints. The source code is transformed using the derived parameters and the new variant is sent for evaluation.

VI. **INTEGRATION OF MULTI-LEVEL CODE OPTIMIZATIONS**

A. **Search Space Representation**

When tuning for compiler optimizations, a simple bit-vector representation of all compiler flags is sufficient for describing the search space. In such a description, a bit being set indicates the transformation is enabled while a bit being unset represents the transformation is disabled. Similarly, when tuning for high-level transformations, the search space for transformation parameters can be represented using an n-dimensional Cartesian space, where each dimension represents the range of possible integral values for some transformation.

For adaptive tuning to be most effective, however, we need to be able to search and describe the space of both classes of optimizations in an integrated framework. Moreover, for many key transformations, such as loop fusion and distribution, the search space representation and parameterization is non obvious. Zhao et al. show that the search space for fusing $n$ statements into $m$ loops without any re-ordering can be as large as $(n-1)m$. Accurately enumerating these configurations is a difficult task. Further complicating matters is the fact that for many transformations, the feasible space is non-contiguous. For instance, the prefetch distance for software prefetching might be specified in multiples of the cache line size within a given range. In MATS, we address these issues and propose a search space representation scheme that is simple, flexible and can represent all classes of optimization in an integrated fashion.

For code transformations that take numeric parameters and for optimizations where a particular heuristic is to be chosen (e.g., instruction scheduling), we enumerate all feasible points and assign a bit position to each option. Therefore, for transformation $t$ with $N$ feasible parameters, we construct a bit vector of size $N$, with the constraint that only one of these bits is activated during a given instance of a search. Thus, the search dimension for such optimizations can be expressed as

$$D^m = \{x \text{ such that } \{0,1\} \text{ and } x_i = 1\}$$

Similarly, for the tolerance parameters (discussed in Section 7), we discretize the tolerance terms and create a bit vector where each position corresponds to a specific tolerance value and only one bit is set during a particular search instance.
D' = \{ x \mid x = 1 \}

For low-level compiler optimizations where the search algorithm needs to make an enable/disable decision we use a single bit to represent each optimization. Thus, for these optimizations we have,

D^b = \{ x \mid x = 1 \}, \text{where} |D^b| = 1

The bit-vectors for high-level optimizations, tolerance terms and low-level optimizations are combined to create the sequence used in each instance of the search process (and also during training in offline mode). Thus, if we have \( p \) high-level optimizations with numeric parameters, \( q \) tolerance parameters and \( r \) low-level optimizations with binary flags then the complete sequence is denoted as follows

O = \{ D^m_0 , ..., D^m_p , D^r_0 , ..., D^r_q , D^b_0 , ..., D^b_r \}

B. Semi-Automatic Tuning

To provide support for semi-automatic tuning, MATS allows users to manually specify the optimization search space. Fig. 4 shows an example search space configuration file used in MATS. 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 used when searching for the best phase-ordering. An enumerated type is a special case of the range type. It can be used to describe non-contiguous search dimensions. In addition, MATS 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 an algebraic inequality within the configuration file

<table>
<thead>
<tr>
<th>gcc</th>
<th># compiler for low-level optimization tuning [optional]</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td># maximum number of program evaluations</td>
</tr>
<tr>
<td>16</td>
<td># number of dimensions in the search space of high-level transformations</td>
</tr>
<tr>
<td>R 1 16</td>
<td># range: 1 ... 16</td>
</tr>
<tr>
<td>P 4</td>
<td># permutation: sequence length 4</td>
</tr>
<tr>
<td>E 2 8 16</td>
<td># enumerated: two possible value 8 and 16</td>
</tr>
</tbody>
</table>

Fig. 4 Example configuration file for semi-automatic tuning in MATS

Information specific to a search algorithm is specified elsewhere. For instance, for simulated annealing, the \( \alpha \) and \( \beta \) factors for each dimension are specified in a separate file. The parameters for the search algorithm have been deliberately kept separate to make the search space representation more general.

C. Extensibility

MATS is designed to be easily integrated with existing compilers and other performance enhancement tools. These tools vary in the way they accept input about how the transformations should be applied. For compilers, transformations are specified using command-line flags. For high-level code transformers the specification mechanism is typically source-code annotation \([46]\). More specialized code optimizers like POET \([47]\) supports an extensive set of parameterized transformations and communication occurs via scripts. To provide maximum flexibility in tuning applications, MATS is designed to support all three interfaces. At each step in the search process, we generate a search point that can be represented using command-line flags, source code annotation or transformation scripts.

MATS currently provides support for GCC, Open64, Intel and LLVM compilers. These compilers support hundreds of different code optimizations and this number is growing with each new release. Because the optimization set is large and certain optimization flags are subsumed by others (e.g., enabling optimization flag \( i \), might enable optimizations \( j \) and \( k \)), interfacing the tuning system with mainstream compilers proves 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 dimensionality and can potentially skew search results. On the other hand, applying the -O3 flag enables a wide range of loop transformations 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 several compilers, to isolate the command-line flags for each optimization and also discard flags that are ineffective or subsumed by others. 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 flags in the respective compiler. Since our approach only uses the command-line interface and is independent of the underlying compiler implementation, it provides a flexible and extensible method of interfacing compilers not just with our tuning framework but other autotuning systems as well. In particular, it can be easily integrated into the PERI autotuning system \([20]\).

VII. EXPERIMENTAL RESULTS

We conducted a series of experiments on several architectures to evaluate and demonstrate the effectiveness of our autotuning system. This section summarizes our findings.

A. Experimental Setup

1) Benchmarks:

We select 13 kernels from different domains in science and engineering. Table III provides a brief description and the source of each benchmark. The kernels are important in their respective domains and have wide applicability. Each kernel can be parallelized using data, task or pipelined methods. The kernels also exhibit many performance issues, often achieving only a small fraction of the peak
performance on current architectures \cite{3,52,53}, making them good candidates for evaluating an adaptive tuning system.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>Parallelization</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>swim</td>
<td>shallow water modeling</td>
<td>data-parallel</td>
<td>SPEC 2000</td>
</tr>
<tr>
<td>mgrid</td>
<td>multi-grid solver</td>
<td>data-parallel</td>
<td>SPEC 2000</td>
</tr>
<tr>
<td>genm</td>
<td>matrix-matrix multiply</td>
<td>data-parallel</td>
<td>BLAS \cite{48}</td>
</tr>
<tr>
<td>gemv</td>
<td>matrix-vector multiply</td>
<td>data-parallel</td>
<td>BLAS \cite{48}</td>
</tr>
<tr>
<td>ger</td>
<td>matrix-vector multiply</td>
<td>data-parallel</td>
<td>BLAS \cite{48}</td>
</tr>
<tr>
<td>knapsack</td>
<td>integer knapsack</td>
<td>pipelined</td>
<td>Morales \cite{49}</td>
</tr>
<tr>
<td>stencil</td>
<td>heat equation</td>
<td>pipelined</td>
<td>ParLab</td>
</tr>
<tr>
<td>advect3D</td>
<td>3D advection</td>
<td>pipelined</td>
<td>PARSEC</td>
</tr>
<tr>
<td>blacksholes</td>
<td>financial analysis</td>
<td>data-parallel</td>
<td>PARSEC</td>
</tr>
<tr>
<td>canneal</td>
<td>synthetic chip</td>
<td>unstructured</td>
<td>PARSEC</td>
</tr>
<tr>
<td>dedup</td>
<td>compression kernel</td>
<td>pipelined</td>
<td>PARSEC</td>
</tr>
<tr>
<td>freqmine</td>
<td>frequent pattern</td>
<td>data-parallel</td>
<td>PARSEC</td>
</tr>
<tr>
<td>streamcluster</td>
<td>data mining</td>
<td>data-parallel</td>
<td>PARSEC</td>
</tr>
</tbody>
</table>

2) Platforms:

Since the goal of adaptive tuning is to deliver portable performance across architectures, we have included four different platforms in our experimental evaluation. Table IV summarizes the software and hardware configuration for each platform. The chosen platforms display variation in the number of cores, processor speed and cache organization and serve as a good basis for evaluating our tuning framework. In the rest of this paper, we refer to these platforms using the names listed in the column headers in Table IV.

<table>
<thead>
<tr>
<th>Program</th>
<th>Core2</th>
<th>Quad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>2.33 GHz Intel Core 2 Duo</td>
<td>2.4 GHz Intel Core 2 Quad</td>
</tr>
<tr>
<td>L1</td>
<td>16 KB, 4-way</td>
<td>32 KB, 8-way</td>
</tr>
<tr>
<td>L2</td>
<td>4 MB, 4-way (shared)</td>
<td>2 x 4 MB, 8-way (shared/socket)</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.3.2</td>
<td>GCC 4.3.2</td>
</tr>
<tr>
<td>OS</td>
<td>Linux</td>
<td>Linux</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>Opteron</th>
<th>8Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>3 GHz AMD Opteron Dual-Core</td>
<td>2.33 GHz Xeon E2830</td>
</tr>
<tr>
<td>L1</td>
<td>32 KB, 4-way</td>
<td>32 KB, 4-way</td>
</tr>
<tr>
<td>L2</td>
<td>2 x 1 MB, 4-way (shared/socket)</td>
<td>8 x 256 KB, 16-way</td>
</tr>
<tr>
<td>Compiler</td>
<td>GCC 4.2.3</td>
<td>GCC 4.2.3</td>
</tr>
<tr>
<td>OS</td>
<td>Linux</td>
<td>Linux</td>
</tr>
</tbody>
</table>

B. Impact of Exposing Parallelization Parameters

As mentioned earlier, one unique feature of MATS is its ability to provide fine-grain control over transformations that affect the nature of the parallelism achieved. Here, we show how changing these parallelism parameters influenced the performance of the chosen programs.

1) Thread Granularity:

We first examine the search space of thread granularity as exposed by our framework. Although we conducted experiments with all benchmarks on four platforms, in the interest of space, in this section, we discuss the most interesting results. Figs. 5(a)-(d) show how changing the amount of work done per thread, impacts the performance of advect3d, dedup, freqmine and knapsack, respectively. Normalized execution times are reported for Core2 and Quad. The execution times are normalized with respect to a baseline version, in which the code is parallelized based on default granularity. For each of these applications, we modify thread granularity using a combination of strip-mining and loop interchange, loop fusion and loop selection. The achieved thread granularity for each program variant is linearized and reported on the x-axis of each chart.
We observe that there is significant variation in performance for all kernels, as a function of changing thread granularity. The most dramatic fluctuations occur for knapsack, where we notice a factor of five speedups when the granularity is increased to a sufficient level. We also observe that no clear performance pattern emerges for these benchmarks on either platform. *advect3d* exhibits a gradual decrease in normalized execution time when granularity is increased, while the execution time for *fourmide* is static for most levels of thread granularity with a few spikes in between. The performance of *dedup* remains flat on *Core2* but is rather erratic on *Quad*. We witnessed similar performance variations for the other kernels as well. These results reiterate the notion that on current architectures, parallel performance is highly influenced by thread granularity and that the performance patterns are not easy to predict. Thus, thread granularity is a parameter that deserves tuning and MATS is effective in extracting the right set of code transformations that exposes these performance variations to the search engine.

2) **Thread Count:**

The number of threads used in a parallel application can have a significant impact on performance. Here, we evaluate performance sensitivity to thread count for *mgrid*, *swim* and *stencil*. We use the strategy described in Section IV to generate program variants with different thread counts on all four platforms. Since thread count is closely related to the amount of parallelism extracted, for each program, we used two different parallelization schemes: one where only the outer loop in a loop-nest is parallelized and one where all loops are parallelized. Fig. 6 shows the performance of the three benchmarks with two different parallelization schemes using different thread counts. In the figure, \( P1TK \) refers to the first mode of parallelism, with \( K \) threads and \( P2TK \) refers to the second mode of parallelism.

We observe that none of the six parallelizing schemes stands out as a clear winner for any platform. On average we observe the best performance improvement on the 8Core platform but this can be attributed to the higher number of processing cores available on this platform. We do notice that in most cases using more threads than the available cores does lead to a loss in performance. However, this phenomenon is not applicable to all programs on all platforms. For example, on the *Opteron* with four cores, the best speedups for *stencil* are achieved when 8 threads are used instead of 4. More interestingly, setting the number of threads equal to the available cores can sometimes cause performance degradation, as we observe for *mgrid* on *Quad* and *swim* on *Opteron*. Thus, these results indicate that like thread granularity, the thread count parameters also warrant inclusion in the search space for achieving portable parallel performance.

C. **Impact of Search Space Pruning**

MATS implements a technique that significantly reduces the search space of memory hierarchy transformations. However, it is important to consider exactly how much performance is lost as a result of moving into the smaller search space. To this end, we perform a set of experiments, comparing our tolerance-based pruning strategy (*pruned*) against regular, in which only regular transformation parameters are searched. In both cases, we apply simulated annealing, *anneal*, as the search technique, as this was the strategy that proved most effective in our earlier experiments. In each version, we use gcc as the back-end compiler and programs are compiled with the `-O3` option turned on (i.e., back-end optimizations are not included in the search space). For both strategies, we allow the search algorithm to execute for 200 evaluations or until it converges.

Fig. 7 illustrates the trade-offs between performance and tuning time for pruned and regular. Numbers are reported for all benchmarks on *Core2* and *Quad*. The results show that on both platforms, pruned is able to find performance values that are comparable to the values found by regular. In a few cases, (e.g., *gemm* and *ger* on *Core2*), pruned discovers better values than those found by regular. On the other hand, regular significantly outperforms pruned on knapsack on both platforms. In post-mortem analysis of the search space, we discovered that for knapsack, the unroll-and-jam factor that works best creates a loop-body whose
register pressure is significantly larger than the available registers. Yet this choice of the unroll-and-jam factor does not lead to an increase in register spills. We speculate this may be due to a favorable interaction with one of the backend optimizations performed by gcc. Overall, we observe that regular has a slight advantage over pruned on both Core2 and Quad.

space for memory hierarchy transformations using tolerance values. In the context of autotuning, where number of program evaluations is a principal bottleneck, this savings in tuning time is particularly significant. In most scenarios, the savings in tuning time will make the small sacrifice in performance worthwhile.

D. Impact of Integration Multi-level Code Optimizations

A key feature of MATS is that it enables exploration of the search space of both data-locality and parallelization parameters in a non-orthogonal manner. Here, we present experimental results that demonstrate the need for this non-orthogonal exploration. We set up an experiment where we explore the combined search space of blocking and parallelization using both orthogonal and non-orthogonal search. We choose two orthogonal search methods: one where the blocking dimensions are explored first (blockfirst) and one where the parallelism dimensions are explored first (parfirst). For multi-dimensional search, we select simulated annealing (anneal) and direct search (direct), both of which are known to be effective in exploring the search space of transformation parameters [13]. To keep the comparison fair, individual dimensions in the orthogonal search are searched using simulated annealing. We allow each search algorithm to run for 100 iterations, or until it converges. We instrument the search algorithms to output the current best value every eight iterations.

Fig. 8 shows results of exploring the combined search space on all four platforms using both orthogonal and multi-dimensional search strategies. Results are reported for the first 64 iterations since no fluctuations were observed beyond this point. For the orthogonal search methods, numbers of evaluations are distributed evenly between blocking and parallel dimensions. That is, if the number of evaluations is 16, then for blockfirst, the first 8 evaluations explore the blocking dimensions, and the last 8 explore the parallelization dimensions. The code variant obtained using gcc -O3 is the baseline and the speedup reported is the geometric mean over all 13 benchmarks listed in Table III. Since we only display the current best value, the minimum speedup is always ≥ 1.

A consistent pattern emerges from the performance data presented in Fig. 8. It is clear that both direct and anneal have a significant advantage over orthogonal search methods on all four platforms when the number of evaluations goes beyond a threshold (near 32). For fewer iterations, parfirst is able to compete with the multidimensional strategies but in the long run does not yield the desired performance. Generally, blockfirst performs worse than parfirst. This indicates that capping the working sets, before selecting the loops to parallelize leads to suboptimal performance. Interestingly, blockfirst fares better on the Core2. This may be because the parallelization search space is smaller for this architecture (fewer affinity configurations and number of threads), and thus the block size has less influence on parallelization decisions. Overall, the poor performance of blockfirst, and less than average performance of parfirst indicate that there is indeed interaction between the blocking and parallelization...
dimensions that are not captured by orthogonal search methods, and hence, the combined search space merits a multi-dimensional search.

the baseline version is the fully optimized code generated by GCC (option -O3). The search algorithm used in all cases was anneal with no restrictions on the number of evaluations performed.

We observe that autotuning yields significant speedups for all benchmarks across all four platforms. On average, the highest speedups are achieved on 8Core. This speaks to the scalability of the proposed strategy and indicates that autotuning can be more effective for architectures where more parameters influence a program’s performance. We also note that our strategy is less effective in obtaining better performance out of the PARSEC benchmarks, canneal, dedup and freqmine. We attribute this to two factors. First, these programs use Pthreads for implementation. Since, MATS only provides limited support for Pthreads programs, we were not able to search for some of the more relevant parameters. Second, the PARSEC benchmarks provide fewer opportunities for tiling, which is a key transformation in our framework. Among the other benchmarks the speedups obtained for the dense-matrix kernels are particularly compelling. These numbers are comparable to the performances obtained by ATLAS, which uses specialized hand-coded assembly level optimizations in tuning the kernels.

VIII. CONCLUSIONS

This paper describes the design and implementation of MATS, a portable adaptive tuning system. The system includes several features that address current autotuning needs, including a framework for exposing tunable parameters for parallelization, a novel strategy for search space pruning, and a mechanism for integrating multi-level code transformations. We conduct a series of experiments to demonstrate the effectiveness of the system. These experimental results show that the tuning strategy employed by MATS can be effective in achieving high performance for scientific kernels across a range of architectures. The analytic modeling can keep the tuning times within manageable levels with a small penalty in performance.

REFERENCES


J. Hollingsworth, Experience with Automated Performance Tuning Using Active Harmony, in , Jul, Presented at the CSciADS Autotuning Workshop, 2008.

M. Stephenson and S. Amarasinghe, Predicting Unroll Factors Using Supervised Classification, in CGO, March, San Jose, CA, USA, 2005.


