Window-based Performance Tuning of Parallelized Applications

Dheya Mustafa and Rudolf Eigenmann

Paramount Research Group, Purdue University
West Lafayette, IN, USA
{dmustaf,eigenman}@purdue.edu
https://engineering.purdue.edu/paramnt

Abstract. In today's multi-core era, to exploit the increased computational power, automatic parallelization of sequential programs combined with tuning techniques is an alternative to manual parallelization. Automatic parallelization concentrates on finding any possible parallelism in the program, whereas tuning systems help identifying efficient parallel code segments and profitable optimization techniques. In this work, we present a generic automatic empirical tuning system that is easily customizable and generates a pruned search space. A key part of our system is an algorithm that partitions the search space in different dimensions into independent windows and tunes them simultaneously.
Our work is one of the first approaches delivering an auto-parallelization system that guarantees performance improvements for nearly all programs; hence it eliminates the need for users to "experiment" with such tools in order to obtain the shortest runtime of their applications. Experiments show that tuned applications perform equal to original serial programs in the worst case and sometimes outperform hand-parallel applications.

Keywords: Automatic Parallelization, Automatic Tuning, Empirical Tuning

1 Introduction

The advent of multi-core processors regenerated researchers' attention to automatic parallelization [31], [11], [7], [15], [27]. However, automatic parallelization, still didn't achieve the required performance to be considered a true alternative to manual parallelization. Parallelizing compilers are only finding partial parallelism in many programs. One reason is that compiler techniques are limited by insufficient static knowledge. Another important reason is that, even where compilers find parallelism, they cannot guarantee that the achieved parallelism results in increased performance. Users may experience speedup of less than one, unless they invest substantial time in tuning the parallel program. This lack of performance is despite the fact that compilers may achieve significant "parallel coverage" (i.e., high fraction of serial execution time enclosed by parallel constructs). In this work, we aim to overcome this issue and, hence, move automatic parallelization from being a research tool to an efficient production tool.

Some program sections, like small inner loops, are parallelizable but parallelizing them does not increase performance. Another goal of the work presented here is to integrate the automatic tuning system with an automatic parallelizing compiler, to make sure that the compiler-parallelized code performs at least as well as the original serial program.

Much research in the last few years has been devoted to both empirical and model-based auto-tuning software [4, 12]. Empirical tuning searches through many candidate program variants and chooses the best, based on execution time [18]. It provides accurate results but may involve long tuning times, as the search space can be large, to capture all potential interactions among optimization options and program sections. Automatic performance tuning also attempts to generalize the optimization process by first parameterizing the optimizations and then searching for the appropriate parameters [25, 29]. Model-based tuning reduces the search space of possible compilation variants, and thus tends to tune faster. The two methods complement each other. In this work, we use a model-based profitability test as a reference point.

This work presents an empirical optimization approach with pruned search space that reduces tuning time and achieves good performance. Our approach extends prior work [14] and partitions the search space in different dimensions into independent windows and tunes each window individually. The algorithm seeks to reach a tuned program from a set of independently tuned windows, since most important interactions are enclosed within a program section. It exploits system engineers' knowledge of the relationships between optimization techniques to further prune the space. It also reduces the tuning time by navigating search points of independent windows concurrently. Our generic tuning system can be easily customized to tune different optimization techniques.

Our customized system currently tunes profitable loop-parallelization [8] and function-inlining techniques by selectively applying these techniques to loop-nests when they improve performance. Inlining is often applied as a pre-pass to optimization, making it difficult to tune. Our system tunes inlining efficiently. It also tunes some architecture-dependent transformation techniques such as loop tiling and loop permutation.

This work makes the following contributions:

- We present a novel, fast tuning algorithm that partitions program into sections (program windows), and greedily tunes each code section independently, while fully accounting for interactions of optimization techniques within code sections. It also partitions Optimization options into set of groups (options' windows), and tunes each group independently. Independent optimization groups and code sections are tuned simultaneously.
- We propose a customizable generic empirical tuning system for parallelized applications. We implement the proposed tuning system using the CETUS source-to-source translation infrastructure [7], and customized it to tune the C-to-OpenMP translator implemented in CETUS. It guarantees performance improvements for nearly all programs.

– We present results of automatically parallelized programs using CETUS and ICC compilers, and empiricaly tuned versus model-based tuned, and hand-parallelized OpenMP programs from the NAS benchmarks.

The remainder of the report is organized as follows: The next section provides a system overview, including the tuning algorithm and search space pruning. Then, we present the customization process and implementation details of the window-based tuning system using the CETUS infrastructure in section 3. Section 4 evaluates the customized system using NPB benchmarks. We discuss related work in section 5. Section 6 presents ongoing and future work. Finally, section 7 draws conclusions.

2 Generic Window-based Tuning System

This section presents the automatic window-based tuning system for compiler generated code. Our generic approach is composed of three stages: Search Space Navigation, Version Generation, and Empirical Measurements, as depicted in Figure 1. The tuning system collects information from the compiler, target machine, and the user. Then, it uses the collected information to prune the search space. From now on, *user* refers to the tuning system customizer in this work. We describe each stage in detail in the following subsections. We also discuss its implementation using CETUS.

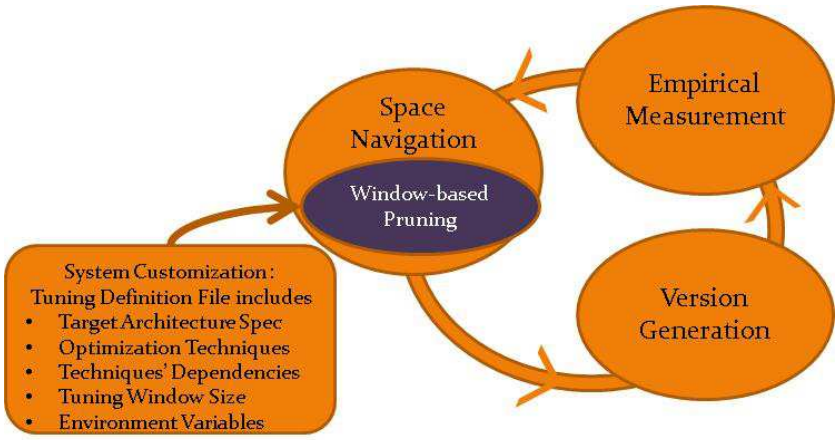

Fig. 1. High-level overview of the automatic tuning system. Search Space Navigation picks the next program version to be tried; Version Generation compiles this version; and Empirical Measurement evaluates its performance at runtime. Space pruner tool is automatically generated based on the information fed from the tuning system customizer.

2.1 Tuning System Specifications

The general search space is defined by a set of factors forming its dimensions: optimization techniques and all loop-nests in the program. Different terms are used in related work to refer to these factors, such as tuning options, optimization options, tunables, parameters, control points, and degree of freedom. we organize all options into three categories based on functionality:

- Category-A : includes options that define the scope of optimization techniques, *aka* tuning domain. For example, tuning window size is used to partition program into sections and tune each section individually.
- Category-B : includes binary and continuous optimization techniques, applied at section-level or program-level. All optimization options have discrete integer values.
- Category-C: includes environment variables, such as number of threads used in the machine.

The user provides the system, as part of the customization process, with a set of tuning options and dependencies between them. The raw search space size is the full factorial of the number of loops in the program and the number of tuning options. This large search space is needed because all optimization options and program sections can potentially influence each others. For example, a benchmark that has 90 loops with two binary tuning options (loop-paralleization and function-inlining), and two continuous tuning options (loop-tiling and permutation) with 10 values each, would have a raw search space of $2^{90} * 2^2 * 10^2 = 4.9 * 10^{29}$. Even if evaluating one program variant takes a tenth of a second and using parallel search on 100 nodes cluster simultaneously, the tuning process could take up to 15 quintillion years.

The proposed tuning algorithm narrows down the search space by eliminating optimization variants that may have similar effects or by deriving information related to interactions between different optimizations. The pruning algorithm partitions the search space into windows and tunes each window individually. First, it partitions the program into *program-type tuning windows*, abbreviated to *p-window*, since the interaction between program statements primarily affects close neighbors [14]. Then, it partitions optimization options into set of independent groups or *option-type tuning window*, abbreviated to *o-window*, based on user provided dependency information about optimization options. The Cartesian product of optimization options' groups (o-windows) and program sections (p-windows) are tuned independently using empirical exhaustive search. Exhaustive search becomes feasible with our pruning techniques, and it can demonstrate upper bounds of tuning time for finding the best. Faster search heuristics can be substituted in a straightforward way. For simplicity, we will discuss the two partitioning steps individually, though the two steps are integrated and compose single pruning algorithm.

2.2 Exploiting User knowledge: O-window Tuning

The proposed tuning system exploits some user-knowledge about tuning options' interactions to perform further pruning. The system customization process in-

cludes providing a tuning system definition with a set of options, number of arguments for each option, and a list of the arguments and range of values for each argument. Figure 2.a shows a hypothetical example of optimization options in the tuning definition file. To configure the system to tune parallelization and tiling, we would add lines "parallelize 0" and "tile 1 tileSize [32 : 256]" to the tuning definition file. User also provides the system with options' dependencies as depicted in Figure 2.b.

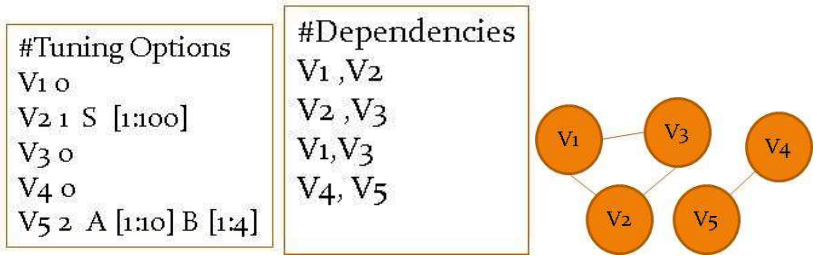

Fig. 2. Illustrative example for tuning options partitioning. Part a shows tuning options file. Part b shows pairs of dependent options. Part c shows the generated dependency graph by mapping options to vertices and dependencies to edges.

The system constructs a dependency graph by directly mapping optimization options to vertices and options' dependencies to edges. Figure 2.c shows the graph constructed from a and b. Then, the dependency graph is partitioned into a set of connected components, where each connected component represents an o-window of dependent optimization options. Optimization options within an o-window are tuned independently of all other o-windows without accounting for any possible interactions between them. Algorithm 1 shows the optimization space partitioning.

For the example in Figure 2, the optimization space partitioning generates two subspaces for o-windows $\{V_1, V_2, V_3\}$ and $\{V_4, V_5\}$. The customized system needs to consider $(2 * 100 * 2) + (2 * 10 * 4) = 480$ options' combinations, while the exhaustive search of raw space needs to consider $2 * 100 * 2 * 2 * 10 * 4 = 32000$ options' combinations in order to capture all possible interactions among options for a single loop-nest. Knowing that variants from different subspaces are independent and can be evaluated in a single run, the optimization space size can be further reduced to be the size of the largest subspace(i.e. 400 in our example).

```
GenOptimizationSubspaces()
input : tuning options {V1,V2,...,Vn}
      tuning options dependencies:
       {{V1,V2},{V1,V4},...,{Vi,Vj},...}
output: Set of optimization subspaces
begin
 Generate a forest graph G using options as
  vertices and dependencies as edges
 Partition G into set of connected components
 For each connected component G'
   Start a new optimization subspace S-G'
   for each vertex V in G'
    for each argument associated with V
      add a new dimension to S-G'
 return set of optimization subspaces.
end.
```

Algorithm 1: Generation of Optimization Subspaces. After partitioning options into o-windows based on dependencies, each o-window will have its own subspace that can be navigated independently of all other subspaces.

2.3 P-window-based Tuning and Program Partitioning Algorithm

Different loop-nests in the program interact with each others. P-window-based tuning concentrates in the interactions of neighboring loop-nests, and ignores possible interactions between distant loop-nests. The system allows user to set p-window size, that will be used to partition program into p-windows, and each p-window will be tuned individually. Algorithm 2 enumerates p-window based search subspace variants. For a program with 90 loop-nests, partitioned into p-windows of three loop-nests, and two on-off tuning options and two continuous tuning options with 10 values each, the search space would be $2^3 * 2^2 * 10^2 = 3200$ per p-window, while exhaustive search of raw space needs $2^{90} * 2^2 * 10^2 = 4.9 * 10^{29}$ variants to account for all possible interactions between loop-nests and optimization options. Since our approach tunes all p-windows independently, multiple measurements can be taken concurrently within a single program run. This search space can be small enough that our system searches it exhaustively.

The algorithm is greedy in that it composes a tuned program from the individually tuned p-windows. The possible interaction at p-window boundaries is not considered in this work and will be discussed in future work.

Selecting P-type Tuning Window Size Selecting p-window size is critical and directly affects both performance and tuning time of the tuning system. The semantics of p-window size is related to the interactions between different loop-nests in the program. A p-window size one means to tune each loop-nest individually without accounting for any interactions between different loop-nests.

```
EnumerateSearchSubspace()
input :TuningOptions,P-TuningWinSize
output:list of p-windows configurations
begin
 Call ProgramPartitioner (P-TuningWinSize)
 for each p-window W in program
  for all loop' combinations in p-window W
   for all options' combinations
    Gen optimization configuration for W
end.
```

Algorithm 2: Enumerating the Search Subspace of Optimization Variants. We consider four optimization options, loop parallelization, tiling, interchange, and inlining. Induction substitution, symbolic analysis, and privatization are tuned indirectly via parallelization. Each p-window variant is represented by a configuration string for each loop-nest in that window. Each p-window has a unique id attached to all its variants configurations.

The other extreme, where p-window size is ∞, considers all possible combinations of the loop-nests to exhaustively account for all interactions between different loop-nests in the program.

Two possible approaches to find the best p-window size considering the trade-off between tuning time and performance. One approach is to analyse data flow as source of interactions between different loop-nests, and partition program based on the analysis. This analytical approach requires detailed analysis and might result in irregular program partitioning and thus complicates tuning process. The other approach is empirical, and is based on the conjecture that most important interactions are within neighboring loop-nests. This conjecture leads to the assumption that performance will converge after a certain p-window size. With less than 10 experiments, we can find convergence point that represents the best p-window size. In the evaluation section, we discuss performance convergence for NPB benchmarks as function of p-window size.

3 Tuning System Customization: CETUS C-to-OpenMP Tuner

Our proposed tuning system can be customized to create a specific instant of tuning system. In this section, We present a customized tuning system for C-to-OpenMP translator using the CETUS compiler infrastructure. It tunes various set of optimization techniques. Some techniques are applied at loop-level, others require global analysis and are applied at program-level to make efficient decisions that influence performance. Also, implementation visibility affects at what level such techniques can be applied. We briefly describe these techniques together with their needs and opportunities for tuning in the following subsections.

3.1 CETUS : Automatic Parallelizing Compiler

The CETUS Compiler Infrastructure is a source-to-source translator for C programs [7]. Input source code is automatically parallelized using advanced static analysis such as scalar and array privatization, symbolic data dependence testing, reduction recognition and induction variable substitution. CETUS eliminates dead code. The translator supports identification of loop-level parallelism and OpenMP directive-based source code generation [1]. Internally, CETUS uses pragma annotations to pass information among different passes. Eager paralleization of small, inner loops could add significant overhead.

3.2 Function Inlining

In the absence of inter-procedural analysis, function inlining can help increase the chances of detecting more parallelism. However, aggressive inlining can lead to code with complex expressions, reducing the parallel coverage. CETUS provides a flexible support for inlining. For our purposes, the capability of inlining function calls inside *for* loop, has proved to be helpful in getting code with increased readability and parallel coverage. Inlining is applied at loop-level.

3.3 Loop Tiling

Tiling combines strip-mining and loop-permutation to partition a loop's iteration space into smaller chunks, to help the data stay in the cache until it is reused. Tiling strongly interacts with parallelization [16]. Because tiling introduces additional code and control overhead, performance degradation is expected if both techniques applied indiscriminately. Tiling is applied at loop-level.

3.4 Loop Interchange

This technique permutes the loops within a loop-nest to align array accesses in row-major order. Loop interchange aims to increase cache hit ratio. It interacts with parallelization, as moving a parallel loop to an inner position increases parallel loop overheads. Loop interchange is applied at loop-level.

3.5 Array Reduction Recognition

Reduction operations, used in many computational applications, commonly take the form of $rv = rv + expr$. Recognizing such operations is key to successfully autoparallelizing many loops. A data-dependence analyser will report a dependence on a reduction operation unless marked as a reduction operation. This transformation is sometimes expensive because of inefficient added code that affects back-end compiler and other techniques [2]. Array Reduction is applied at program-level, and tuned indirectly via loop-parallelization.

3.6 Privatization

Private variable serves as a temporary variable in a loop, one that is written first and used later in the loop. Array sections provide temporary locations for private array variables. These variables do not need to be exposed to the other threads at runtime, so the data-dependence analyser can safely assume these variables do not have dependencies. Privatization is applied at program-level, and tuned indirectly via loop-parallelization.

3.7 Induction Variable Substitution

An induction statement has a form, $iv = iv + expr$, and must be replaced by another form that does not induce data dependence. If the right-hand side in the preceding form can be expressed as a closed-form expression that does not contain iv, the dependence in the preceding statement will be removed. The close-form expression requires a multiplication operation instead of a less expensive addition operation. This technique is applied at program-level, and tuned indirectly via loop-parallelization.

3.8 Symbolic Analysis

This technique computes the ranges of integer variables' value at each program point and returns a map from each statement to a set of value ranges that are valid before each statement. This repository of ranges, together with utility functions, provides other passes with knowledge about symbolic terms, including the bounds of variables and expressions, and determines through symbolic comparison if one expression is semantically greater than another. A secondary effects of symbolic analysis are found when the technique helped recognize parallel loops that were too small to improve performance and introduced an overhead instead [2]. Symbolic Analysis is applied at program-level, and tuned indirectly via loop-parallelization.

3.9 Program Instrumentation

The Version Generation stage of the proposed tuning system uses automatic source code instrumentation. It measures application performance by annotating the source code before and after loop-nests with instrumentation calls. It provides enough timing information about each section in the program with negligible overhead [5, 13]. The instrumentation can be enabled/disabled by compiler flags.

3.10 Tuning System Workflow

This subsection describes the tuning system workflow as depicted in Figure 3. User provides the system with his knowledge about optimization options, p-window size, and environment variables. Based on that, Options partitioner divides options into a set of independent o-windows, and defines the optimization subspace for each o-window.

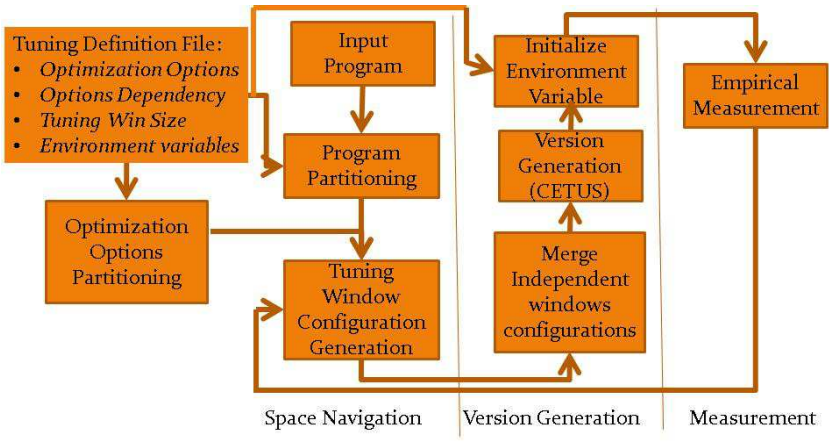

Fig. 3. Customized Tuning System Workflow.

The program partitioner divides each procedure in the program into a set of p-windows. The p-window size is the number of loop-nests it contains. Procedure calls work as borders between p-windows. After partitioning is completed, a detailed window-wise descriptive configuration file containing a list of loop-ids and tuning options applied to each loop is generated and fed to the Version Generation. Figure 4 shows an example of automatically generated configuration file for a program of four loop-nests and p-window size 2.

```
#loop level configuration: loopname, options, loop_flag
main#0, -tile=size=256  -interchnage=0 –parallelize=1 –inline=1, 1
main#1, -tile=size=256 -interchange=0 –parallelize=1 –inline=0, 1
main#2, -tile=size=256 -interchange=0 –parallelize=0 –inline=1, 1
main#3, -tile=size=256 -interchange=1 –parallelize=1 –inline=0, 1
#program level configuration
program, -reduction=2 -range=1 –privatize –induction, 1
#environment variables
env, NUM_OMP_THREAD=8
```

Fig. 4. Internal Configuration file is the interface between space navigation and version generation stages. Each line contains loop name, optimization options, and loop flag which enable/disable applying options to corresponding loop. Line starts by reserved word *program* includes options that can be applied at program-level. Environment variables is identified by reserved word *env*.

Version Generation is implemented in CETUS using SelectiveTransform generic pass controlled by command-line options, which reads the configuration file and generate corresponding program version. User provides the system with the implementation of transformation techniques such as loop-parallelization, function-inlining, loop-tiling, and loop-interchange. The output is generated by CETUS in the form of CETUS parallel annotations that are attached to corresponding loops in the Intermediate Representation (IR). Selective-Parallelize keeps or removes the OpenMP parallel annotations of selected loops. Selective inlining, tiling, and permutation are applied to the loop-nest when they are enabled in the configuration file. Program-level optimization techniques like array reduction and privatization are applied to the whole program by enabling the corresponding pass in command-line that invokes CETUS.

Using the independence of optimizations that the user has defined, the tuner evaluates these optimizations simultaneously in a single run to reduce the tuning time. Consider a program with four loop-nests and p-window size 2, and the number of optimization options' combinations is N . Program Partitioner generates two p-windows, each p-window has $2^2 * N$ variants. Total variants of both p-windows are $8 * N$. Combining independent variants of the two p-windows in a single run will drop the number of runs to $4 * N$.

At the end of the tuning process, a tuning log for each tried variant and it's performance record is generated. This will be used for identifying the best variants.

4 Performance Evaluation

In this section, we present a comprehensive experimental evaluation of our customized system using of speedup over the serial program and parallel coverage, mentioned in the introduction. Also we discuss performance convergence as a function of p-window size. This is an empirical approach to approximate p-window size. Tuning time is discussed at the end of this section.

4.1 Benchmarks Characteristics

The NAS Parallel Benchmarks(NPB) are a set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from Computational Fluid Dynamics (CFD) applications. They consist of five kernels and three pseudo-applications. Four classes of problems are used in the evaluation. These are Class W, Class A, Class B, and Class C. The classes of problems in NPB differ mainly in the sizes of principle arrays which generally affects the number of iterations of contained loops [3, 9]. Because of the availability of OpenMP program variants, NPB suite is ideal for evaluating parallelizing compilers. We use NPB suite implemented in C language [21].

4.2 Experiment Setup

We conducted experiments using a single-user x86-64 machine with two 2.5 GHz Quad-Core AMD 2380 processors and a 32GB memory. The running OS is Red Hat Enterprise Linux. We used the Intel ICC compiler version 11.1. Eight benchmarks from NPB are used to test our proposed system. We used the W dataset in the NPB benchmarks to train our tuning system. A, B, and C are used as production datasets.

4.3 Parallel Coverage

Figure 5 shows the parallel coverage of the NPB benchmarks using dataset A. We compare the parallel coverage for hand-parallel programs, and automatic parallel programs. High parallel coverage indicates efficient execution on an ideal parallel machine. The IS benchmark has low parallel coverage in both variants; it is not amenable to effective automatic parallelization.

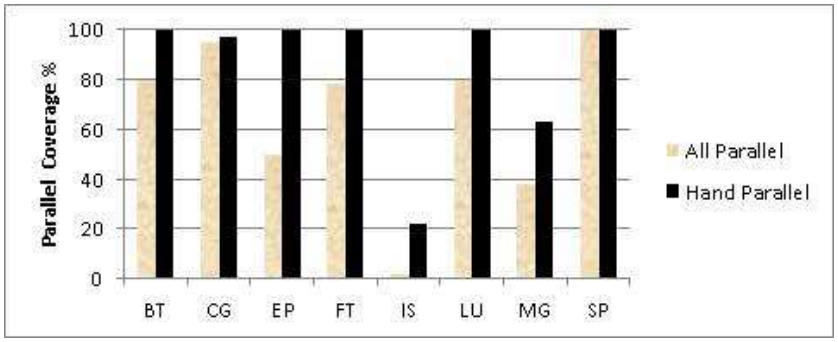

Fig. 5. Parallel coverage for NPB benchmarks using dataset A. Hand Parallel presents parallel coverage of the original benchmarks. All Parallel shows parallel coverage of the automatically parallelized code using CETUS.

4.4 Selecting P-type Tuning Window Size Based on Performance Convergence

We compare speed of tuned applications using our customized tuning system as a function of p-window size while keeping all other optimization options fixed. Figure 6 shows that performance of FT, IS, LU, and MG benchmarks are not affected by varying p-window size. These benchmarks do not benefit from automatic parallelization and the speedup is around one.

BT and EP are minimally affected by varying p-window size, one reason for that is the minimal interaction between loop-nests as in EP, which has one large loop-nest that consumes more than 90% of execution time. Another reason is

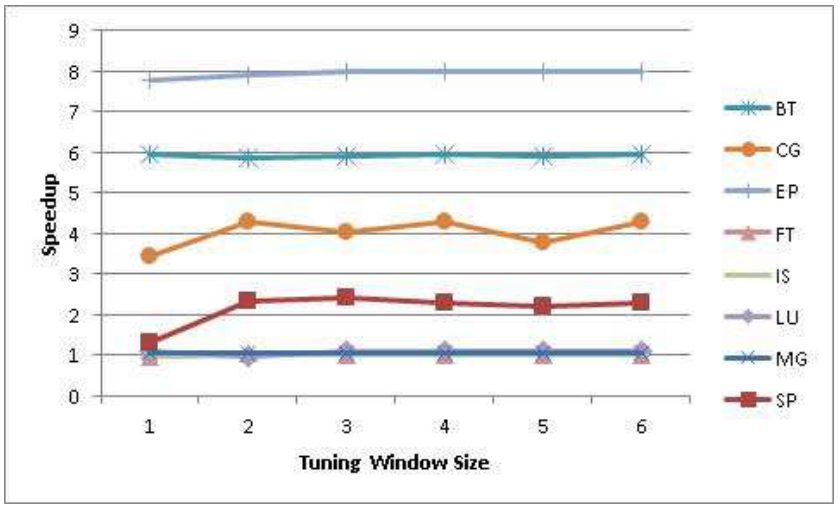

Fig. 6. Effect of p-type tuning window size on performance.

that tuning system tolerates varying p-window size and guarantees best performance; we found that performance is affected by varying p-window size using a production dataset. CG and SP shows convergence at p-window size 3. Another interesting observation is that odd size p-windows behave the same. We attribute this effect to the loop-nests' interactions on p-windows boundaries when even p-window size is used.

4.5 Speedup

We compare the speedup of our tuning system (Empirically Tuned) with the speedup of the serial code, hand parallel code, CETUS parallel code, and CETUS parallel code tuned by the model-based profitability test implemented in the CETUS infrastructure. We will name the later Model-Based.

CETUS has a simple built-in model for beneficial parallel loops. It symbolically computes the workload of each loop by counting loop iterations and the child statements. If the workload is smaller than the given threshold, the loop is serialized either at compile-time (if the workload is known integer) or at run-time (through OpenMP IF clause). The CETUS-user can also set threshold value via command line.

We produced the serial code timing from the NPB Benchmarks by disabling OpenMP flag in the back-end compiler(ICC). Hand Parallel code refers to the original parallel benchmarks. All Parallel code is the automatically parallelized code without tuning. Model-Based is generated by enabling profitability test option in CETUS. Figure 7 shows the speedup of the NPB benchmarks over 4 datasets W, A, B, and C.

For a training dataset, our approach guarantees that the output tuned code performs at least as good as the original serial code, and generally does better. But the effectiveness of our approach, like that of all profile-based approaches, is dependent on the representativeness of the training dataset. If the training dataset is not representative of the production dataset, the output tuned code may perform worse than the original serial code. Figure 8 shows speedup gained using ICC auto-parallelization compiler with '-parallel' option. Conservative ICC parallelizing compiler could not guarantee a non-degrading performance. CETUS outperforms ICC in best cases.

In CG and SP, the parallel coverage exceeds 90% and the results show a speedup on both the CETUS parallel and CETUS tuned versions; SP tuned outperforms the hand parallel version. One important reason is that hand parallel selects to parallelize loop at the second level while our system selects the outer loop to be parallel. Another reason is that our system parallelizes profitable small loops that were considered inefficient by hand-programmer.

While investigating the effect of inlining on parallel coverage, we found that the performance of BT and EP is significantly improved by inlining. For FT, and LU benchmarks, the parallel coverage exceeds 50%. However, performance is low since most parallel regions are inner loops and the tuner serializes them to avoid parallel-loop overhead. More advanced compiler techniques are required to discover parallelism in those benchmarks.

We found a degradation in parallel coverage due to inlining in LU; it reflects side effects of fixed inlining on automatic parallelization. This effect was eliminated by selective inlining in our tuning system.

4.6 Tuning Time

To capture all interactions between loops and optimization options, a large search space is required. For a program with N loops, and M optimization options, exhaustive space size $f(N, M) \in \omega(2^{(N+M)})$, assuming all options are binary. Our tuning system, using p-window size n, and o-window size m, prunes the search space to size $f(N, M) \in \Theta(2^{(n+m)})$. Although the pruned space size appears to be exponentially growing, it is relatively constant to the problem size (N,M).

For example, having four tuning options(Parallelize, Inline, Tile, Interchange), and p-window size 3, we compute space size when all options are dependent(i.e. o-window size 4). We also compute space size based on hypothetical assumption that {Paralleize,Inline} are independent of {Tile, Interchange}(i.e o-window size 2). We will call the two configurations PrunedI and PrunedII, respectively. Table 1 compares the search space size of the three configurations.

The tuning time of window-based system is a function of the window size, instead of being a function of the number of loops in the program. In our experience with empirical space navigators, such as Combined Elimination [8], we found that it considers 173 variants for LU and 558 variants for SP in tuning only loop-parallelization technique.

NPB	Space Size			#Loops
	Exhaustive	PrunedI	PrunedII	
BT	2^{70+4}	$2^{3+4} = 128$	$2^{3+2} = 32$	70
CG	2^{24+4}	128	32	24
EP	2^{10+4}	128	32	10
FT	2^{27+4}	128	32	27
IS	2^{15+4}	128	32	15
LU	2^{50+4}	128	32	50
MG	2^{32+4}	128	32	32
SP	2^{91+4}	128	32	91

Table 1. Tuning time in term of search space size. We compare exhaustive space to pruned space by our system using p-window size 3, and o-window sizes 4(PrunedI) and 2(PrunedII).

5 Related Work

A scalable auto-tuning system for compiler optimization [23] using Active Harmony [22] reduces tuning time by evaluating multiple search space variants concurrently using a cluster of nodes [24]. Such technique is known as parallel search. Using 64 nodes cluster, within 10 steps of tuning, they can exhaustively evaluate 640 variants. Their system tunes *permute, tile, unroll, data copy, split,* and *non-singular* on only kernels. Our system considers possible interactions between loops, and exploits user-knowledge to prune search space. Our tuning system evaluates multiple search space variants by combining them into single run on a single node. Parallel search could further reduce tuning time.

The POET (Parametrized optimizations for empirical tuning) [33] provides embedded scripting language for parametrizing complex code transformations, but cannot overcome an exponential search space. It empirically tunes *loop interchange, blocking,* and *unrolling* of two linear algebra kernels. We tune a wider range of transformations on a full benchmark suite.

Different research projects exploit user-provided information to enhance tuning. Dooley and Kale tune control points that affect the MPI architecture and applications characteristics at runtime in [10]. User is required to provide control point effect on performance(higher is better or lower is better). Tuning parameters used in their work fall under our third category of tuning options which does not require version generation.

Tuning System for Software-Managed Memory Hierarchies automatically tunes general applications to machines with software-managed memory hierarchies [20]. It reduces the search space dimensionality by grouping tunables(based on memory level affected) and searching each group separately. In our work, we group optimization options based on their dependencies provided by user.

Other research projects work on empirical optimization of linear algebra kernels and domain specific libraries. ATLAS [30] uses the technique to generate highly optimized BLAS routines. It uses a near exhaustive orthogonal search

(search in one dimension at a time by keeping rest of the parameters fixed). Instead, we search multiple dimensions within a group of parameters simultaneously with other independent groups.

The OSKI (Optimized Sparse Kernel Interface) [28] library provides automatically tuned computational kernels for sparse matrices. FFTW [6] and SPIRAL [32] are domain specific libraries. FFTW combines the static models with empirical search to optimize FFTs. SPIRAL generates empirically tuned Digital Signal Processing(DSP) libraries. Rather than focussing on one particular domain, our system aims at providing a general-purpose customizable compiler based approach tuning system.

Many compiler optimization space navigators use a feedback-directed approach, which iteratively uses information from the current step to decide the next experimental optimization combinations, until convergence criteria are reached. Optimization Space Exploration (OSE) iteratively constructs new optimization combinations using "unions" of the ones in the previous iteration [26]. Statistical Selection (SS) uses orthogonal arrays to compute the main effect of the optimizations based on a statistical analysis of profile information, which in turn is used to find the best optimization [19]. Combined Elimination iteratively identifies the harmful optimizations and removes them in a batch [17].

6 Ongoing and Future Work

We are working on customizing the proposed generic system to tune OpenMP-to-GPU transformer and OpenMP-to-MPI transformer. In the ongoing work, we are studying the interactions of adjacent p-type tuning windows, one approach is to overlap the two p-window boundaries with neighboring p-windows. Another approach is to apply inter-windows tuning. We will also extend our tuning system to include different space navigation algorithms for the pruned search space, such as Combined Elimination, and Parallel Search , further accelerating the tuning process.

7 Conclusions

We have presented an automatic *window-based* tuning system with pruned search space and applied it to a parallelizing compiler. The system clearly outperforms the state-of-the-art reference point, which makes static decisions about the profitability of parallelizing transformations. Our tuning system partitions both program and optimization options into windows. Then, it tunes each window independently, significantly reducing the search space of optimization variants and thus tuning time.

We also discussed the performance of our tuning system, and compare it with CETUS and ICC auto-parallelization compilers, model-based decision algorithms, and hand-parallelized OpenMP programs using the NAS benchmarks. The results show that the presented tuning techniques are able to efficiently navigate the large search space of parallelization techniques and individual loops on

which to apply them. The combined parallelizer-tuning system is able to guarantee performance greater or equal to the serial execution in all programs and outperforms the hand-parallelized program in one benchmark.

References

1. Openmp [online]. available: http://openmp.org/wp/.
2. Hansang Bae and Rudolf Eigenmann. Performance analysis of symbolic analysis techniques for parallelizing compilers. In *Proc. of the Workshop on Languages and Compilers for Parallel Computing(LCPC'02)*, pages 280–294, August 2002.
3. E. Barszcz, J. Barton, L. Dagum, P. Frederickson, T. Lasinski, R. Schreiber, V. Venkatakrishnan, S. Weeratunga, D. Bailey, D. Bailey, D. Browning, D. Browning, R. Carter, R. Carter, S. Fineberg, S. Fineberg, H. Simon, and H. Simon. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.
4. Chun Chen, Jacqueline Chame, Yoonju Lee Nelson, Pedro Diniz, Mary Hall, and Robert Lucas. Compiler-assisted performance tuning. *Journal of Physics: Conference Series*, 78(1):012024, 2007.
5. William Y. Chen, Scott A. Mahlke, Nancy J. Warter, Richard E. Hank, Roger A. Bringmann, Sadun Anik, and Wen mei W. Hwu. Using profile information to assist advanced compiler optimization and scheduling, 1992.
6. I-Hsin Chung and Jeffrey K. Hollingsworth. Using information from prior runs to improve automated tuning systems. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, SC '04, pages 30–, Washington, DC, USA, 2004. IEEE Computer Society.
7. Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Computer*, 42(12):36–42, 2009.
8. Chirag Dave and Rudolf Eigenmann. Automatically tuning parallel and parallelized programs. In *LCPC '09: Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, 2009.
9. Rob F. Van der Wijngaart. Nas parallel benchmarks version 2.4. Technical report, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, 2002.
10. Isaac Dooley and Laxmikant V. Kale. Control points for adaptive parallel performance tuning. November 2008.
11. Rudolf Eigenmann, Jay Hoeflinger, and David Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. Parallel Distrib. Syst.*, 9:5–23, January 1998.
12. Sylvain Girbal, Nicolas Vasilache, Cdric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34:261–317, 2006.
13. Oscar Hernandez, Fengguang Song, Barbara Chapman, Jack Dongarra, Bernd Mohr, Shirley Moore, and Felix Wolf. Performance instrumentation and compiler optimizations for mpi/openmp applications. In *Proceedings of the 2005 and 2006 international conference on OpenMP shared memory parallel programming*, IWOMP'05/IWOMP'06, pages 267–278, Berlin, Heidelberg, 2008. Springer-Verlag.

14. Dixie Hisley, Gagan Agrawal, and Lori Pollock. Evaluating the effectiveness of a parallelizing compiler. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, volume 1511, pages 195–204, December 1998.
15. Chunhua Liao, Oscar Hernandez, Barbara Chapman, Wenguang Chen, and Weimin Zheng. Openuh: an optimizing, portable openmp compiler. *Concurrency and Computation: Practice and Experience*, 19(18):2317–2332, 2007.
16. Zhelong Pan, Brian Armstrong, Hansang Bae, and Rudolf Eigenmann. On the interaction of tiling and automatic parallelization. In *First International Workshop on OpenMP*, pages 24–35, 2005.
17. Zhelong Pan and Rudolf Eigenmann. Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *The 4th Annual International Symposium on Code Generation and Optimization (CGO)*, pages 319–330, 2006.
18. Zhelong Pan and Rudolf Eigenmann. Peak—a fast and effective performance tuning system via compiler optimization orchestration. *ACM Trans. Program. Lang. Syst.*, 30(3):1–43, 2008.
19. R. P. J. Pinkers, P. M. W. Knijnenburg, M. Haneda, and H. A. G. Wijshoff. Statistical selection of compiler options. In *Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 494–501, 2004.
20. Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed memory hierarchies. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 280–291, New York, NY, USA, 2008. ACM.
21. S. Satoh. NAS Parallel Benchmarks 2.3 OpenMP C version [Online]. Available: `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp`, 2000.
22. Cristian Tapus, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active harmony: Towards automated performance tuning. In *In Proceedings from the Conference on High Performance Networking and Computing*, pages 1–11, 2003.
23. Ananta Tiwari, Chun Chen, Cha Jacqueline, Mary Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–12, Washington, DC, USA, 2009. IEEE Computer Society.
24. Ananta Tiwari, Jeffrey K. Hollingsworth, Chun Chen, Mary W. Hall, Chunhua Liao, Daniel J. Quinlan, and Jacqueline Chame. Auto-tuning full applications: A case study. *IJHPCA*, 25(3):286–294, 2011.
25. Georgios Tournavitis, Zheng Wang, Björn Franke, and Michael F. P. O'Boyle. Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. *SIGPLAN Not.*, 44(6):177–187, 2009.
26. Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler optimization-space exploration. In *In Proceedings of the international symposium on Code generation and optimization*, pages 204–215, 2003.
27. Hans Vandierendonck, Sean Rul, and Koen De Bosschere. The paralax infrastructure: automatic parallelization with a helping hand. In *PACT*, pages 389–400, 2010.
28. Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Institute of Physics Publishing*, 2005.
29. Zheng Wang and Michael F.P. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *Proceedings of the 14th ACM SIGPLAN*

symposium on Principles and practice of parallel programming, PPoPP '09, pages 75–84, New York, NY, USA, 2009. ACM.

30. R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *CONFERENCE ON HIGH PERFORMANCE NETWORKING AND COMPUTING*, pages 1–27. IEEE Computer Society, 1998.
31. Blu William, Ramon Doallo, Rudolf Eigenmann, John Grout, Jay Hoeflinger, Thomas Lawrence, Jaejin Lee, David Padua, Yunheung Paek, Bill Pottenger, Lawrence Rauchwerger, and Peng Tu. Parallel Programming with Polaris. *Computer*, 29:78–82, December 1996.
32. Jianxin Xiong, Jeremy Johnson, Robert Johnson, and David Padua. Spl: A language and compiler for dsp algorithms, 2001.
33. Q. Yi, K. Seymour, H. You, R. Vuduc, and D. Quinlan. Poet: Parameterized optimizations for empirical tuning. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1 –8, march 2007.

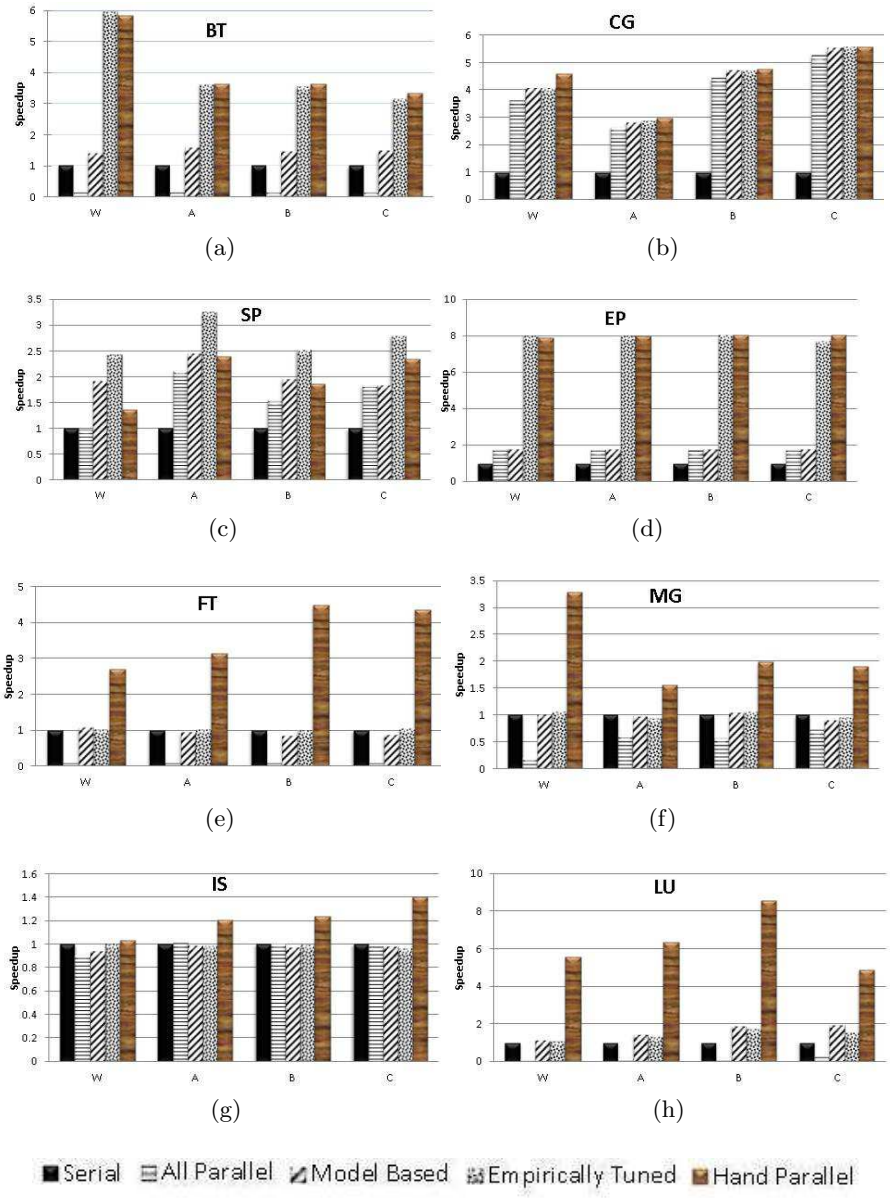

Fig. 7. Tuning results for NAS Parallel suite. Training is performed with W dataset. All Parallel shows performance for compiler-parallelized code. Model-Based represents the speedup for the automatically parallelized programs using model-based profitability test available in CETUS. Empirically tuned shows speedup for tuned programs using our system. Hand Parallel shows speedup for original NPB programs.

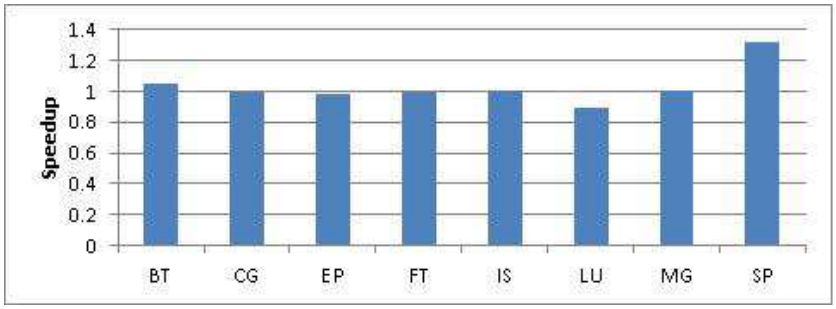

Fig. 8. Speedup for ICC auto-parallelization using '-O3 -parallel' flags and dataset A for NPB benchmarks.