Portable Section-level Tuning of Compiler Parallelized Applications

Dheya Mustafa and Rudolf Eigenmann
Electrical and Computer Engineering
Purdue University
West Lafayette, USA
Email: {dmustaf,eigenman}@purdue.edu

***Abstract*—Automatic parallelization of sequential programs combined with tuning is an alternative to manual parallelization. This method has the potential to substantially increase productivity and is thus of critical importance for exploiting the increased computational power of today's multicores. A key difficulty is that parallelizing compilers are generally unable to estimate the performance impact of an optimization on a whole program or a program section at compile time; hence, the ultimate performance decision today rests with the developer. Building an autotuning system to remedy this situation is not a trivial task. This work presents a portable empirical autotuning system that operates at program-section granularity and partitions the compiler options into groups that can be tuned independently.**

To our knowledge, this is the first approach delivering an autoparallelization system that ensures performance improvements for nearly all programs, eliminating the users' need to experiment with such tools to strive for highest application performance.

I. Introduction

Despite the enormous potential and importance of automatic parallelization for increasing software productivity in today's multicore era, current compilers do not use this capability as their default behavior. As we will show, both state-of-the-art commercial and advanced research compilers, while capable of improving the performance of many programs, may also *degrade* the performance of individual codes. This behavior would be unacceptable as the default of a compiler. The primary reason is that there is insufficient knowledge at compile time to make adequate optimization decisions. Even though many advanced parallelization techniques exist today, including several runtime decision methods, three decades of autoparallelization research have not yet delivered the needed, consistent compilation tools for multicores [1]–[5]. In this paper, we will show that by building on advances in automatic tuning, and by further improving this technology, compilers can deliver automatically parallelized applications that *ensure* performance greater or equal to their source programs. Our results will demonstrate that this can be done with current commercial as well as with research compilers.

We advance the employed, automatic tuning methods in two ways. First, our new techniques support the optimization of *individual program sections*. Doing so is both important and challenging. We will show that section-level tuning can outperform whole-program tuning by up to 82%. The challenge is in managing the drastic growth in the search space of optimization variants. Empirical tuning methods work by evaluating many optimization variants and choosing the one that performs the best at runtime [6]–[10]. During this process, they need to consider interactions between the optimization variants. This issue is already severe in existing, whole-program tuning methods. The effect of an optimization may depend significantly on the presence of another (e.g., unrolling and vectorization influence each other). Therefore, a large number of combinations of program optimizations need to be evaluated. In our section-based tuning system, interactions also occur between optimizations applied to different program sections. Therefore, combinations of program sections in different optimization variants will need to be explored, further growing the already large search space. We use a key observation to manage this additional growth: Interactions between program sections happen primarily among near neighbors. We will consider potential interactions within *windows* of program sections, but not across them. We will show that the window size can be kept small, significantly limiting the increase in program optimization time of our section-level tuning method.

Second, we introduce a *portable* tuning framework and demonstrate its application to *all* parallelizers that we could obtain. Creating portable tuning systems is critical, as the development effort and thus cost is large. Re-usability in tuning has not been addressed, so far; the many proposed systems and techniques use substantially different frameworks and components [11]–[16] . In particular, different approaches use specialized optimization search spaces, which are custom-built and navigated with specialized tuning engines. Our approach is two-fold. We use a framework that conceptualizes the essential elements of empirical optimization, and we employ a tuning definition system, through which the user (typically the compiler developer) customizes the framework for the task a hand. We will evaluate the implementation of this approach for a range of parallelizing compilers and show that the customization efforts are small.

This paper makes the following contributions.

- We introduce a novel, fast tuning algorithm that is able to optimize individual program sections. To keep tuning time low, it partitions the program into *windows*; it fully

This work was supported, in part, by the National Science Foundation under grants No. CNS-0720471, 0707931-CNS, 0833115-CCF, and 0916817-CCF.
SC12 November 10-16, 2012, Salt Lake city, Utah, USA
978-1-4673-0806-9/12/$31.00 ©2012 IEEE

accounts for the interaction of optimizations of different sections within windows, but not across windows. Independent optimization groups and code sections are tuned simultaneously. We will show that section-based tuning improves performance significantly, that a small window size is sufficient, and that tuning times are reasonable.
- We present a portable, empirical tuning framework for automatic parallelization, which we implement and evaluate on our CETUS source-to-source translator [3] as well as the OpenUH [28], Rose [17], PGI [18], and ICC compilers [19]. We will show that our tuning system improves the performance of all these compilers, and the porting effort to each compiler is small.
- Applying our tuning method turns all compilers into tools where automatic parallelization can be used as the *default behavior*. We will show that the performance of all programs in our test suite (the NAS Parallel Benchmarks) for all compilers mentioned above is at least as good as the performance of the serial programs. Hence, users can benefit from the performance gains of automatic parallelization without risk of performance degradation. This eliminates the need for users to experiment with such tools. We will also show that, relative to the hand-parallelized programs, tuned automatic parallelization gains significant performance in half of our benchmark suite and in one case exceeds the manually optimized code.

The remainder of the paper is organized as follows: The next section provides a system overview, including the tuning model, system definition process, tuning algorithm, and search space pruning method. Section III applies this tuning framework to five compilers, Cetus, ICC, PGI, Rose, and OpenUH, demonstrating its portability. Section IV evaluates our work using the NAS parallel benchmarks. We discuss related work in Section V. Section VI draws conclusions and discusses future work.

II. Portable Tuning System

One aim of creating a portable tuning system is to make it easy for compiler developers to plug their existing compiler into a framework that applies empirical tuning. In this way, compilers can be enhanced to find the best combination of their optimization capabilities, using an offline process that is similar to profile-based compilation. The system should allow the compiler developer to define compiler `make` information and all possible optimization options. From this information, the tuning system determines the best-performing program variant for each code section. Recall a key challenge in doing so: Because of the interactions of compiler optimizations within and across program sections, the sheer number of possible optimization combinations would make a brute-force approach infeasible – leading to extremely long tuning times. This paper pursues an approach that exploits *locality of optimization interactions* [20] – distant program sections do not influence each others significantly, allowing substantial pruning of the number of optimization variants that need to be explored.

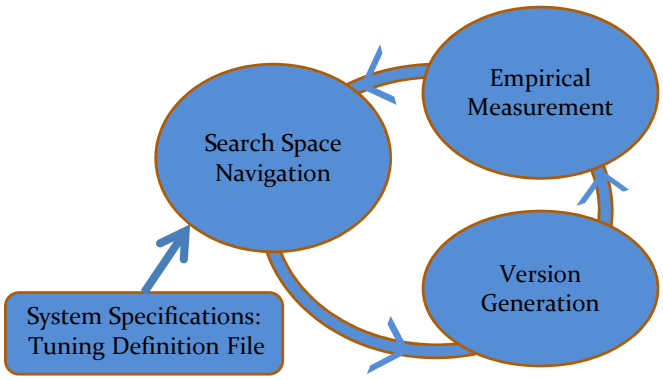

Fig. 1. High-level model of the automatic tuning system. Search Space Navigation picks a next program version to be tried; Version Generation compiles this version; and Empirical Measurement evaluates its performance at runtime. The search space is generated and pruned based on the specifications from the tuning definition file.

The focus of this paper is on developing such a portable tuning system for improving automatic parallelization. To this end, the following introduces a generic tuning model and prototype that are applicable to a wide variety of scenarios. We will demonstrate the portability to two commercial and three research parallelizing compilers.

A. Tuning Model

Figure 1 shows the high-level model of our portable tuning system. Program tuning proceeds in three steps. *Search Space Navigation* iteratively traverses the space of all feasible combinations of program optimizations, until a suitable "best" variant has been found. In each iteration, the current combination is passed on to the *Version Generator* – typically a compiler – which produces the code. Next, the performance gets evaluated by the *Empirical Measurement* component. The performance results are fed back to the Search Space Navigator, which decides on a next iteration or terminates tuning.

This model can be configured through a *System Specification* by means of a *Tuning Definition File (TDF)*, which defines all optimization options. The specifications include compiler options for global as well as for section-level optimizations and allow the tuning system developer to indicate which optimizations may interact with each other. In defining these interactions, the tuning developer can be conservative; independent optimizations will speedup the tuning process, whereby specification inaccuracies may affect the performance outcome but not correctness. The specification also expresses strategies for partitioning programs into *windows* (typically one to a few loops) that will be optimized independently. Available strategies are the choice of a fixed or an adaptive window size. A window size of one would amount to ignoring all interactions between program sections. A window size of ∞ would consider all interactions, even between distant sections, leading to long tuning times. We will show that a small window size is sufficient. The TDF also defines the `make` file information for compilers and runtime environments. The portability of the proposed model lies in this configurability

and in the applicability to many tuning scenarios of compilers and other areas.

The *Automatic Search Space Builder* processes the TDF specifications and creates the search space of optimization variants. The raw search space would be the full factorial of all tuning option values and program sections. Pruning is crucial. The key information enabling pruning is the program windows (or *p-windows*), splitting the code into independently tunable sections and the declared optimization dependences, allowing groups of non-interacting optimizations to be formed (called *o-windows*). The Search Space Builder creates an independent search space for each p-window. Across o-windows, only the sum of all optimization variants need to be explored, rather than their Cartesian product.

For a program with N loops, and M binary optimization options, the exhaustive search space size is $2^{(N+M)}$. Our tuning system, using p-window size n, prunes this space to size $2^n * OptSpace$, where 2^n is independent of the program size (i.e. 8 in our prototype), and $OptSpace$ is the size of the optimizations search space. To define $OptSpace$, let us assume that the tuning system has L o-windows, with the *ith* o-window size m_i. Then, $OptSpace$ equals $\sum_{i=1}^{L}(2^{m_i})$. Knowing that 2^M equals $\prod_{i=1}^{L}(2^{m_i})$, it is clear that $OptSpace$ is less than 2^M, as it represents the number of interacting techniques combinations. In practice, the resulting search space is small enough to allow efficient navigation.

An example shows the importance of pruning: a program that has 6 loops with two binary tuning options (loop-parallelization and function-inlining), and two discrete tuning options (loop-tiling and permutation) with 4 values each, would have a raw search space of $(2^6 - 1)(2^2)(4^2) = 4032$. With p-window size 3, only 448 variants are navigated during tuning in the worst case, which is about 11% of the raw search space. For larger applications, the raw search space is exponentially growing, while the number of variants being navigated (i.e. tuning time) remains constant.

The Search Space Navigator traverses all independent search spaces concurrently and determines, for each p-window, a next optimization variant that will be empirically evaluated. It calls the Version Generator to create the needed program code. Once a final "best" program version has been found, the Search Space Navigator calls the Version Generator one more time to create the final combination of all best-performing p-windows. Note that this final version may be different from all program variants made during the tuning process, as it combines the individually best-performing program sections.

The Version Generator creates the program variant as requested by the Search Space Navigator. The necessary make information for the needed compilation steps and the mapping of optimization variants to command line flags is taken from the TDF. Not all compilers are able to apply optimizations to specific program sections. Our model includes a postprocessing step, allowing compilers that generate OpenMP output to enable or disable individual parallel loops. This is one of the most important section-level tuning decision. Version Generation also instruments each p-window with timing calls (or, in general, performance counters). For compilers that do not provide such instrumentation, the postprocessor provides this feature as well.

```
# this line is a comment
# new section begins with ! sign
!Loop-Level Optimization Options
loop_tile   1  tile_size   [4:256:*=4]
loop_unroll 1  unroll_size [2:16:*=2]
loop_parallelize 0
vec   1 vec_threshold [50:100:+=10]
!Program level Optimization Options
reduction 0
!Options' Dependencies
loop_tile     loop_parallelize
!Windowing Strategy
fixed 3
!Environment Variables
OMP_NUM_THREADS   [1:8]
!Make Definition
```

Fig. 2. Example of a Tuning Definition File (TDF). The TDF is composed of several sections for specific sets of tuning options and their formats. Each line describes one optimization technique. Dependencies are specified by indicating the involved pair of techniques.

The *Empirical Measurement* step executes the program variant and evaluates its performance. It sets runtime parameters either as fixed values (given by TDF make information) or tunable parameters chosen by the Search Space Navigator (e.g., the number of threads). Independent measurements are taken for each of the n p-windows. Thus, in a single program run, an optimization variant is evaluated for n program sections, leading to an additional reduction of the tuning time. The n measurements are passed on to the Search Space Navigator for deciding on the next iteration of the tuning process.

The following Section presents our prototype tuning system, based on the described model.

B. Tuning System Prototype

This Section describes the implementation of the above model in our portable tuning system. We will discuss TDF, Search Space Generation and Navigation, Version Generation, and Empirical Measurements.

1) TDF and Search Space Generation: Figure 2 shows an example TDF. The TDF language contains the following categories of information:

- Category-A defines binary and discrete optimization techniques (*aka* tuning options, tunables, parameters, control points, or degree of freedom in related work). The optimizations may be applicable at the section level or program level. All optimization options have discrete integer values. Most common are binary options, switching a technique on or off. An example of a multi-valued option is loop unrolling, which may have a parameter of 2, 4, 8

or 16. Optimizations that are prerequisites for others are combined into one multi-valued option. For example, if the *on/off* option *forward-substitution* needs to be *on* for the option *substitute-only-integers/substitute-all* to have an effect, the two are combined into a three-valued option *off/substitute-integers-only/substitute-all*. Techniques that belong to this category are represented by *loop-level/program-level optimization options* sections in the TDF.

- Category-B defines the possible interactions among optimization techniques as a list of pairs. This category is represented by *Options' Dependencies* section in the TDF.
- Category-C describes the windowing strategy to partition a program into sections and tune each section individually. It can be either fixed or adaptive. This category is represented by *Windowing Strategy* section in the TDF.
- Category-D: includes two groups of options:
  - Tunable environment variables, such as the number of threads used in the machine.
  - Make definitions and fixed back-end compiler options as well as fixed environment variables. We use a generic parametrized MakeFile.

  This category is represented by *Environment Variables* and *Make Definition* sections in the TDF.

The Automatic Search Space Builder processes this information. Its key algorithm prunes the potentially combinatorial number of optimization variants to a manageable size. To this end, it partitions the search space into windows that will be tuned individually. The partitioning happens for both the program space (p-windows) and the optimization space (o-windows). Our tuning prototype currently supports a fixed p-window size. We will show that, in practice, a small size of up to three loops is sufficient. Subroutines are boundaries for windows; thus the partitioned programs may include p-windows of one, two, and three loops. O-windows are formed based on TDF-provided dependence information for the optimizations, as described below. The optimizations within an o-window are dependent; all combinations of options need to be explored. The optimizations of different o-windows are independent; the combined search space of two o-windows is only the sum of the individual search spaces, not the product.

Figure 3 shows the automatic search space builder. For simplicity, we will discuss the two partitioning steps separately; in the implementation, the two steps form a single pruning algorithm.

Algorithm 1 shows the optimization space partitioning. The algorithm constructs a dependency graph, where optimizations represent vertices and options' dependencies form edges. Next, the graph is partitioned into a set of connected components, where each component represents an o-window of dependent optimization options.

In practice, most generated graphs are not strongly connected, which means not every pair of techniques within an o-window are possibly interacting. Instead of trying exhaustively all techniques' combinations, the algorithm uses graph coloring to decide the minimum number of combinations that captures all interactions. Graph coloring finds the minimum number of colors such that no two adjacent vertices share the same color. Independent techniques are mapped to vertices with the same color. For a graph with n colors, it needs 2^n combinations of techniques composing that graph. For example, if V1 interact with V2, and V2 interact with V3, two colors are sufficient for marking the resulting connected component (graph). Thus, only four combinations are needed ($\overline{V1}\overline{V2}\overline{V3}$, $V1\overline{V2}V3$, $\overline{V1}V2\overline{V3}$, $V1V2V3$) instead of eight (assuming V1,V2, and V3 are binary techniques). This algorithm guarantees a window search space size equal to the size of the largest strongly connected component in the graph (i.e. (V1,V2) or (V2,V3) in our example).

After partitioning is completed, each search space variant is written to a file in the form of a list of loop-id, command line tuning options applied to that loop, and p-window id. This file

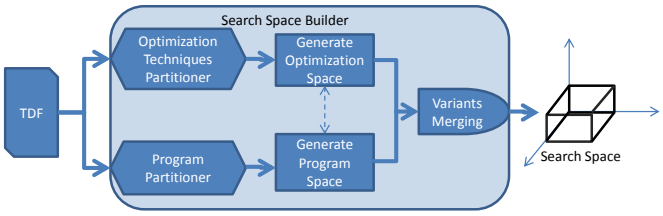

Fig. 3. The Automatic Search Space Builder reads the TDF information provided by the tuning developer and generates a pruned search space. Pruning exploits the fact that distant program sections have little influence on each other and non-interacting optimizations, per the TDF, can be evaluated independently.

Algorithm 1 Generation of Optimization Space. 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.

1: $GenOptimizationSpace()$
2: **input**:tuning options $(V1, V2, ..., Vn)$
3: tuning options dependencies:$(Vi, Vj) \mid i, j \in [1:n]$
4: **output**:Set of optimization subspaces
5: **begin:**
6: Map options to vertices
7: Map dependencies to edges
8: Generate a forest graph G
9: Partition G into connected components
10: **for** each connected component $G^{cc} \subset G$ **do**
11: Start a new optimization subspace S-G^{cc}
12: Color the vertices in G^{cc}
13: **for** each vertex V in G^{cc} **do**
14: **for** each argument associated with V **do**
15: add a new dimension to S-G^{cc}
16: **end for**
17: Merge same colored dimensions in S-G^{cc}
18: **end for**
19: **end for**
20: **return** set of optimization subspaces.
21: **end.**

is passed on to the Search Space Navigator.

2) Search Space Navigation: All p-windows have separate search spaces. The Search Space Navigator traverses them independently, picking a next variant for each window to be evaluated next and calling the Version Generator with this information. After each evaluation step, the measured points in the search space are annotated with their performance results; in addition, the best performance and corresponding point is recorded for each p-window.

Because of the extensive pruning, the resulting search spaces in our experiments were small enough, so that exhaustive search of all points was feasible. Exhaustive search guarantees to find the best version and demonstrates upper bounds of tuning time. Heuristic algorithms are known that navigate faster, but do not guarantee optimality [21]–[23]. These algorithms can be substituted in a straightforward way.

Because windows can have different sizes, their search spaces differ in size as well. After the navigation of smaller p-windows has completed, their best version is used for the remaining calls to the Version Generator.

For final version generation, the Search Space Navigator simply passes the recorded best variants for each p-window to the Version Generator. Whole-program optimizations are the same for all p-windows.

3) Version Generation: A Program version is expressed by the combination of all its p-windows variants and the whole-program optimizations. Optimization options are expressed as command line options; a loop is identified by an automatically generated loop ID, which is composed of the subroutine name containing that loop and a serial number reflecting the loop position.

We extended the Cetus translator to support section-level tuning by adding a Selective Transform Pass that applies transformations to individual loops, as specified by the p-window variants. To supports enabling and disabling of OpenMP parallel loops, we implemented a postprocessor in Cetus that selectively removes OpenMP parallel loop directives based on the variant description obtained from the Space Navigator. Other compilers do not support section-level tuning. We will show later how these capabilities are also used to support section-level tuning and enabling/disabling individual parallel loops for the Rose compiler.

The proposed tuning system uses automatic source code instrumentation implemented in Cetus. It measures application performance by annotating the source code before and after loop nests with timer calls. It provides enough timing information about each section in the program with negligible overhead.

4) Empirical Measurement: The code made by the Version Generator is already instrumented for taking performance measurements. These may be overall timing measurements for compilers that only support whole-program optimizations or there may be section-level instrumentation for evaluating individual p-windows. The Empirical Measurement step sets the environment parameters according to the information contained in the TDF and executes the program. Performance results are gathered and passed on to the Search Space Navigator. Empirical Evaluation measures all p-windows in one program run; the largest p-window dominates tuning time.

To minimize the effect of measurement variations, the system can be configured via TDF to set the number of measurements repetitions. The system chooses the minimum value of the repeated measurements, since delays are usually caused by undesirable system effects. Performance results are reported to the Search Space Navigator as a vector of loop IDs and corresponding execution times.

Through the TDF, our prototype system can be configured to create a specific instant of a tuning system. The following section describes how this has been done for five different parallelizing compilers.

III. Case Studies in Tuned Autoparallelization

We applied the described portable tuning system to five compilers. ICC and PGI are commercial compilers, while Cetus, OpenUH, and Rose are research compilers.

A. Tuning the Cetus Autoparallelizer

This section presents a customized tuning system for the C-to-OpenMP translator that is part of the Cetus compiler infrastructure. We briefly describe these techniques together with their needs and opportunities for tuning.

The Cetus compiler infrastructure is a source-to-source translator for C programs. 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. The translator supports the detection of loop-level parallelism and the generation of C code annotated with OpenMP parallel directives [24].

Cetus uses a set of optimization techniques, some of which are applied at the loop level, others are global optimizations applied at the program level. We include three categories of such techniques: techniques that assist program analysis include symbolic analysis and subroutine inlining; parallelism-enabling techniques include data privatization, reduction parallelization and induction variable recognition; locality-enhancement techniques include loop interchange and tiling. Specifics of these techniques are described in [3].

There are many tuning opportunities in Cetus. Eager parallelization of small, inner loops could add significant overhead. Aggressive inlining can lead to code with complex expressions, reducing the parallel coverage. Tiling strongly interacts with parallelization [25]. Because tiling introduces additional code and control overhead, performance degradation is expected if both techniques are applied indiscriminately. Loop interchange also interacts with parallelization, as moving a parallel loop to an inner position increases parallel loop overheads. Reduction transformation is sometimes expensive because of inefficient added code that affects back-end compilers and other techniques. The closed-form expression produced by induction variable substitution introduces more costly operations. Even though symbolic analysis may help recognize

parallel loops, they may be too small to improve performance and introduce overhead instead [26].

We define the portable system to tune profitable loop-parallelization [27] 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. Cetus is tuned at section-level granularity.

B. Tuning the OpenUH Compiler

The OpenUH compiler is a branch of the Open64 compiler maintained by the High Performance Computing Tools (HPC-Tools) group of the University of Houston. The OpenUH is a robust, optimizing, portable OpenMP compiler, which translates OpenMP 2.5 (www.openmp.org) directives in conjunction with C, C++, and FORTRAN 77/90 (and some FORTRAN 95). The OpenUH compiler's major optimization components are the inter-procedural analyzer, the loop nest optimizer and global optimizer. In order to achieve portability while preserving most optimizations, the compiler includes an IR-to-source translators to produce compilable code immediately before the code generator [4], [28].

We tune auto parallelization (*apo*), loop unrolling (*unroll*), loop nest optimization (*interchange, blocking, prefetch*), function inlining (*inline*), optimization level (O_n), inter procedural analysis (*alias, constant propagation, dead function elimination*). OpenUH does not support section-level optimization; it is tuned at the program level.

C. Tuning the ICC Compiler

The auto-parallelization feature of the Intel C++ Compiler, invoked by the '-parallel' option, automatically translates serial portions of the input program into equivalent multithreaded code. The autoparallelizer analyzes the dataflow of the program's loops and generates multithreaded code for those loops that can be safely and efficiently executed in parallel. This enables the potential exploitation of the parallel architecture found in symmetric multiprocessor (SMP) systems [19].

The ICC compiler also includes a heuristic for deciding when a parallel loop benefits performance. Via a user-defined threshold, ICC tries to balance the overhead of creating multiple threads versus the amount of work available to be shared among the threads.

ICC's Compiler options included in our tuning system are auto-parallelization (*parallel*), loop unrolling (*unroll*), loop blocking (*blocking*), vectorization (*vec*), data prefetch (*prefetch*), scalar replacement (*scalar-rep*), data alignment (*align*), optimization level (O_n), and function inlining (*inline*). The ICC compiler does not support section-level optimization; it is tuned at the program level.

D. Tuning the PGI Compiler

The PGI compiler incorporates global optimization, vectorization, software pipelining, and shared-memory parallelization capabilities. Vectorization transforms loops to improve memory access performance and makes use of packed SSE instructions, which perform the same operation on multiple data items concurrently. Unrolling replicates the body of loops to reduce loop branching overhead and provide better opportunities for local optimization, vectorization and scheduling of instructions. Performance for loops on systems with multiple processors may also improve using the parallelization features of the PGI compilers [18], [29]. The PGI compiler is tuned at the program level.

Compiler options being tuned include auto-parallelization (*concur*), vectorization (*vect*), function inlining (*inline*), loop unrolling (*unroll*), inter-procedural analysis and optimization (*ipa*), cache aligning (*align*), partial redundancy elimination (*PRE*), and optimization level (O_n). For a more detailed description of these options, we refer to the PGI user guide [18].

E. Tuning the Rose Compiler

Rose is an open-source compiler infrastructure to build source-to-source program transformation and analysis tools for large-scale Fortran 77/95/2003, C, C++, OpenMP, and UPC applications [17]. Rose provides several optimizations including autoparallelization, loop unrolling, loop blocking, loop fusion, loop fission, and inlining. Loop optimizations and autoparallelization are separate projects under Rose that are not yet integrated. Currently, we tune autoparallelization only. We used the Cetus postprocessor to support section-level tuning of the Rose autoparallelizer. Rose successfully compiles five out of eight NAS Parallel benchmarks.

F. Tuning System Building Time

Most of the effort and time spent when building the tuning systems for these compilers was in reading the compilers' documentations, understanding the behaviour of the underlying techniques, and determining the possible interactions among them. On average, building a tuning system consumed around five working days, from downloading and installing the compiler, to the beginning of the tuning process. Writing the tuning definition files consumed minor time, as the TDFs include a few tens of lines of code. We expect this to be the typical customization time for a developer who is not yet familiar with the underlying compiler.

All compilers discussed previously have rich sets of options affecting their performance. Playing the role of compiler writer, we have chosen the most important options and corresponding argument ranges to include in the tuning system specifications. We excluded options that have a consistent negative effect. To do so, we performed a preliminary step using the following algorithm: Starting from the base case, where all techniques are turned on, we turned off one technique at a time. Techniques with consistent negative performance effects were not included as tuning options. This algorithm has a complexity of $O(n)$, where n is the number of optimizations.

IV. Performance Evaluation

This section presents a comprehensive experimental evaluation of our portable tuning system using the metric "speedup

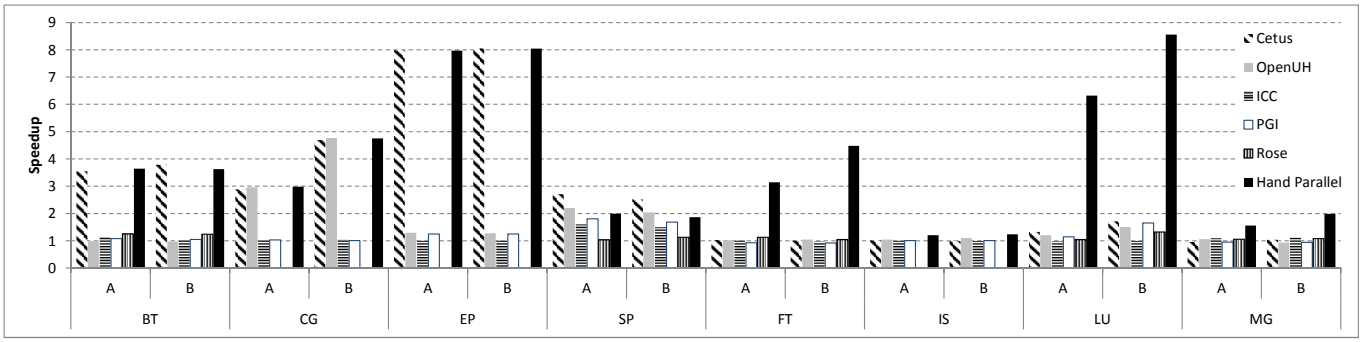

Fig. 4. Speedup of tuned Cetus, OpenUH, PGI, Rose, and ICC compilers over the serial version of NAS Parallel Benchmarks. We use Class W for training, while Class A and B are production datasets. Hand Parallel refers to the speedup of the original NAS parallel benchmarks. Rose speedups are not available for CG, EP, and IS.

over the serial program". The results shows that the tuning system ensures non-degrading performance in all cases and often leads to significant speedup. Section-level tuning improves substantially over whole-program tuning and over untuned optimization. The results also quantify characteristics of the tuning system itself. They show that our system feasibly navigates large search spaces and that a small p-window size is sufficient.

A. Benchmarks Characteristics

The NAS Parallel Benchmarks (NPB) are designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from Computational Fluid Dynamics (CFD) applications. Four classes of problems are used in the evaluation. These are Class W, Class A, Class B, and Class C. The classes differ mainly in the sizes of principle arrays, which generally affects the number of iterations of contained loops [30], [31]. Because of the availability of OpenMP program variants, the NPB suite is favorable for evaluating parallelizing compilers, providing hand-optimized reference points. We use the NPB suite implemented in the C language [32].

B. Experiment Setup

We conducted the experiments on 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, UHCC version 4.2, PGCC version 9.0-3 64 bit, and Rose compiler version 0.9.5a. We used all eight NPB programs. The W data set is used during tuning; A, B, and C are used as production datasets.

C. Overall Evaluation of Optimizing Compilers

This section evaluates the five tuned compilers described in Section III. These include Cetus, two important commercial compilers (Intel's ICC and the PGI compilers) as well as the Rose and OpenUH research compilers. The hand-parallelized versions of the programs are used as reference points. The serial versions of NPB were obtained by compiling the codes with the serial compiler options. Hand Parallel code refers to the original parallel benchmarks. Figure 4 shows the overall results of the tuned compilers.

Cetus shows significantly better speedups over serial execution than ICC ad PGI compilers. This indicates that not yet all of the advanced parallelization techniques developed in research compilers have been transferred to industrial products. In four of the programs, the best Cetus version matches, and in one of these cases (SP) outperforms, the hand-parallelized codes. We find that, in roughly 50% of scientific/engineering codes, parallelizers yield substantial improvements.

One important reason for the superior performance of the tuned SP code is that Hand Parallel chooses to parallelize an important 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 the programmer of the benchmark.

OpenUH reaches hand parallel performance in CG, and outperforms it in SP. PGI and ICC show a noticeable speedup –close to hand parallel– in SP. The Rose autoparallelizer we downloaded was unable to compile three of the benchmarks: CG, EP, and IS.

D. Efficiency of Section-Level Tuning

This section compares the performance of tuning at the program level, which is the common practice of all previous work, versus the proposed section-level tuning. To do that, we created instances of our tuning system to tune the Cetus and Rose Compilers at the program level, using the same optimization techniques as described in Section III. Figure 5 compares the average speedup of Cetus and Rose tuned at section-level versus program-level tuning, relative to the NAS serial programs for datasets W, A, B, and C.

The detailed results revealed that section-level tuning clearly outperforms program-level tuning in two benchmarks: LU and SP. Section-level tuning improves performance by 42% over program-level tuning on the best case.

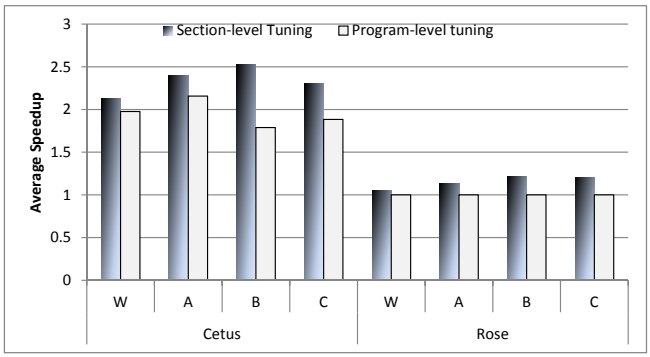

Fig. 5. Speedup of Tuned programs at section-level versus program-level, relative to the NAS serial programs using the Cetus and Rose compilers. Measurements are averaged over the NAS suite. Class W is used for training, while Class A, B, and C are used as production datasets

E. Tuning Impact on Performance

This section quantifies the impact of tuning for the different compilers. We compare the average performance of *tuned* NAS benchmarks with *untuned* and *profile-based tuned* programs supported by some of the compilers under study. Untuned programs are generated by turning on all optimizations. The Profile Based Tuned versions are generated using a compile-time profitability test in Cetus, profile feedback optimization in the PGI compiler, feedback directed optimization in OpenUH, and profile guided optimization in ICC. Rose does not include a profitability test; we used the described postprocessor to tune profitability in Rose. Figure 6 shows the results.

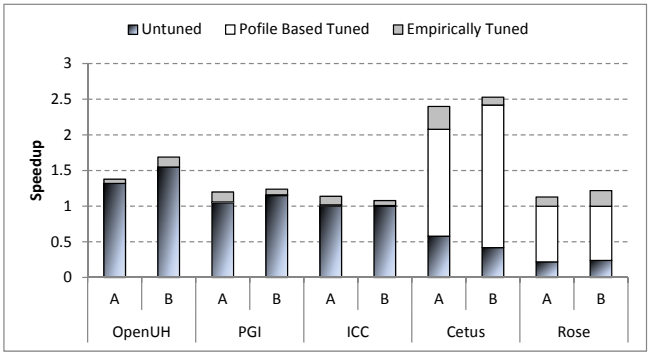

Fig. 6. Speedup contribution percentage of our tuning system (*Tuned*) versus *Untuned* programs and *profile Based* tuned for Cetus, OpenUH, PGI, Rose, and ICC compilers on average over the NAS suite. Profile based tuning is shown for the compilers that support it.

Detailed results revealed that, without tuning, all compilers degrade performance in some of the benchmarks. For example, the ICC compiler degrades performance to 82% in LU, OpenUH degrades performance to 78% in BT, the PGI compiler degrades performance to 83% in MG, Rose degrades performance to 25% in FT, and Cetus degrades performance to 21% in LU.

Untuned performance does not degrade in the PGI, ICC, and OpenUH compilers on average, which we attribute to their conservative optimizations. OpenUH uses a built-in profitability test for its techniques, explaining the minor effect of profile based optimization. On the other side, aggressive optimization capabilities in Cetus provide ample opportunity for the tuner to improve performance as shown in Figure 4.

Comparing results in Figures 6 and 5, we found that the gain of tuning over profiling seems less than over whole-program tuning. This means that whole-program tuning performs less than profile-based optimization. Profile-based optimization lies between program-level and section-level tuning. It is similar to section-level tuning, with a p-window size of 1, and a constant parallelization threshold for all loops in the program. This optimization method does not account for interactions among loops.

F. Tuning Window Size

The choice of the program window size is important. We provide empirical evidence that a small p-window size of three is a good choice, which we have used in all implementations. Figure 7 compares the speedup of Cetus-tuned applications as a function of the p-window size, while keeping all other optimization options the same. The average performance of the NAS suite reaches a maximum value at p-window size three, validating our approach.

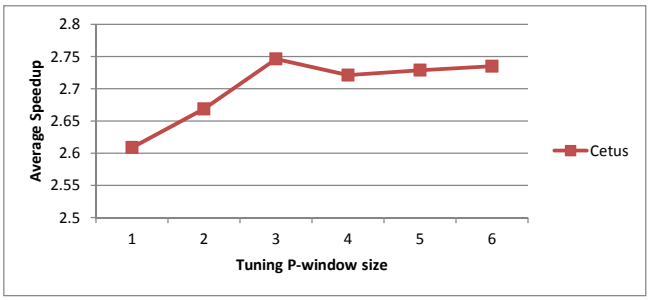

Fig. 7. Effect of p-window size on performance for Cetus on average over NAS benchmarks using dataset A.

G. Tuning Time

Table I compares the number of variants considered in tuning the five compilers under study. It also shows the ratio of measured variants to the raw search space size. Rose and Cetus are tuned at the section level with p-window size three, while the other compilers are tuned at the program level.

TABLE I
TUNING TIME IN TERMS OF MEASURED SEARCH SPACE SIZE. CETUS AND ROSE ARE TUNED AT THE SECTION LEVEL; ALL OTHER COMPILERS ARE TUNED AT THE PROGRAM LEVEL. COLUMN 3 SHOWS THE RATIO OF THE PRUNED SEARCH SPACE TO THE RAW SEARCH SPACE.

Compiler	Pruned Search Space Size	% of Space Considered	Tuning Granularity	Avg Raw Space Size
OpenUH	36	1%	Program-level	3600
PGI	60	3.9%	Program-level	1536
ICC	540	1%	Program-level	311040
Cetus	705	2%	Section-level	35250
Rose	8	<1%	Section-level	900

Tuning process for ICC, PGI, and OpenUH compilers consumed less than 15 minutes on average over all benchmark

suite. Cetus and Rose required more than an hour in the worst case; this is due to the high parallelization overhead of some space variants as well as the high source-to-source translation times for some techniques, such as inlining.

V. Related Work

A lot of research considers autoparallelization and tuning. To our knowledge, portability and re-usability across different applications have not been addressed before. Our system is the first to support section-level tuning and ensures performance no less than the original programs.

A scalable auto-tuning system for compiler optimization [11] using Active Harmony [33] reduces tuning time by evaluating multiple search space variants concurrently using a cluster of nodes [34]. This technique is known as parallel search. Using a cluster of 64 nodes, 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 prunes the search space. Our tuning system evaluates multiple search space variants by combining them into a single run on a single node. Parallel search could further reduce tuning time.

The POET (Parametrized optimizations for empirical tuning) provides an embedded scripting language for parametrizing complex code transformations [12], 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.

Recent work uses POET to tune the Rose loop optimizer [35]. All optimizations were re-implemented to support parameterization. The fine-grain parameterization allows to capture optimization interactions. Our system does not require optimization re-implementation. A new optimization can be added to our portable system by means of a tuning definition file.

Different research projects exploit user-provided information to enhance tuning. Dooley and Kale tune control points that affect the MPI architecture and application characteristics at runtime [36]. The user is required to define a control point effect on performance (higher is better or lower is better).

The Tuning System for Software-Managed Memory Hierarchies automatically tunes general applications to machines with software-managed memory hierarchies [13]. It reduces the search space dimensionality by grouping tunables (based on memory level affected) and searching each group separately. Our work assumes no prior knowledge of optimization options, and tunes a wide range of techniques, beyond memory related optimizations. Our system groups optimization options based on their dependencies provided by the compiler writer. Our system supports portability across a wide range of applications.

Iterative compilation aims at selecting the best parameterization of the optimizations options for a given program or for a given application domain. It typically affects optimization flags (switches), parameters (e.g., loop unrolling, tiling), phase ordering, the heuristic itself, or the hybridation of multiple heuristics [37]–[43]. Our generic system is built based on iterative tuning.

Some tuning systems targeting large-scale applications use outlining to extract hot code sections (kernels) and tune them off-stage, then plug them back in to the code [44]–[46]. This is an orthogonal issue and could be combined with our approach.

Other research projects work on empirical optimization of linear algebra kernels and domain-specific libraries. ATLAS [14] 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 the rest of the parameters fixed). The order of options being tuned may affect performance if a following option interacts with a previous one. Instead, our method searches multiple dimensions within a group of parameters simultaneously with other independent groups. Our work tunes faster, which makes it feasible to tune a wide range of applications.

The OSKI (Optimized Sparse Kernel Interface) [15] library provides automatically tuned computational kernels for sparse matrices. FFTW [47] and SPIRAL [48] 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 to tuning.

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 a convergence criteria is reached. Optimization Space Exploration (OSE) iteratively constructs new optimization combinations using "unions" of the ones in the previous iteration [21]. 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 [22]. Combined Elimination iteratively identifies the harmful optimizations and removes them in a batch [23]. All these approaches are orthogonal to our work and can be incorporated to navigate the pruned space, further reducing tuning time.

Combined empirical tuning of loop fusion and model-based tuning of loop tiling, vectorization and parallelization in an automatic parallelization framework is proposed [16]. Kernels were used in their evaluation. In contrast to our work, this system does not address portability and tunes a specific set of techniques. Our work tunes all techniques empirically. Also, we performed an extended evaluation using the full suite of NAS benchmarks.

Loop tiling and unrolling are simultaneously tuned in [49] using iterative tuning. One of the search algorithms used was named window search. Initially, the window is the whole 2D space, after each iteration, a smaller window is defined based on samples taken. The validity of this algorithm is biased to architectural attributes of both techniques. Our window-based tuning is a different concept. Their window navigation

algorithm can replace exhaustive search within windows in our approach.

VI. Conclusions and Future Work

We have presented an automatic portable *window-based* tuning system and applied it to a set of research as well as commercial parallelizing compilers. 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. 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 systems are able to ensure performance greater or equal to the serial execution in all programs and outperform the hand-parallelized programs in one benchmark.

We found parallelizers to be reasonably successful in about half of the given science-engineering programs. On average over all compilers, tuning improves performance by 170% over the untuned applications. Section-level tuning improves performance by 20% over program-level tuning on average.

We are working on porting the proposed generic system to tune more compilers, such as OpenMP-to-GPU and OpenMP-to-MPI translators. We are studying the interactions of adjacent p-windows, and supporting more windowing strategies. We will plug-in different space navigation algorithms for the pruned search space, such as Combined Elimination, and Parallel Search, further accelerating the tuning process.

References

[1] B. William, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel Programming with Polaris," *Computer*, pp. 78–82, 1996.

[2] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the Automatic Parallelization of the Perfect Benchmarks," *IEEE Trans. Parallel Distrib. Syst.*, pp. 5–23, 1998.

[3] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A source-to-source compiler infrastructure for multicores," *IEEE Computer*, pp. 36–42, 2009.

[4] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: an optimizing, portable OpenMP compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

[5] H. Vandierendonck, S. Rul, and K. D. Bosschere, "The Paralax infrastructure: automatic parallelization with a helping hand," in *PACT*, 2010, pp. 389–400.

[6] C. Chen, J. Chame, Y. L. Nelson, P. Diniz, M. Hall, and R. Lucas, "Compiler-assisted performance tuning," *Journal of Physics: Conference Series*, p. 012024, 2007.

[7] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *International Journal of Parallel Programming*, pp. 261–317, 2006.

[8] Z. Pan and R. Eigenmann, "Peak—a fast and effective performance tuning system via compiler optimization orchestration," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 3, pp. 1–43, 2008.

[9] G. Tournavitis, Z. Wang, B. Franke, and M. F. P. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.*, pp. 177–187, 2009.

[10] Z. Wang and M. F. 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*, ser. PPoPP '09, 2009, pp. 75–84.

[11] A. Tiwari, C. Chen, C. Jacqueline, M. Hall, and J. K. Hollingsworth, "A scalable auto-tuning framework for compiler optimization," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–12.

[12] 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*, march 2007, pp. 1 –8.

[13] M. Ren, J. Y. Park, M. Houston, A. Aiken, and W. J. Dally, "A tuning framework for software-managed memory hierarchies," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 280–291.

[14] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Conference on High Performance Networking and Computing*. IEEE Computer Society, 1998, pp. 1–27.

[15] R. Vuduc, J. W. Demmel, and K. A. Yelick, "Oski: A library of automatically tuned sparse matrix kernels," in *Institute of Physics Publishing*, 2005.

[16] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan, "Combined iterative and model-driven optimization in an automatic parallelization framework," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10, 2010.

[17] Daniel Quinlan and Chunhua Liao and Thomas Panas and Robb Matzke and Markus Schordan and Rich Vuduc and Qing Yi, "ROSE user manual: A tool for building source-to-source translators," 2012. http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf

[18] "PGI® compiler user's guide," 2012. http://www.pgroup.com/doc/pgiug.pdf

[19] "Intel® c++ compiler for linux systems user's guide." http://denali.princeton.edu/intel_cc_docs/c_ug/index.htm

[20] D. Mustafa, Auranzeb, and R. Eigenmann, "Performance analysis and tuning of automatically parallelized OpenMP applications," in *Proc. of the International Workshop on OpenMP, IWOMP*, 2011, pp. 150–164.

[21] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D. I. August, "Compiler optimization-space exploration," in *In Proceedings of the International Symposium on Code Generation and Optimization CGO03*, 2003, pp. 204–215.

[22] 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*, 2004, pp. 494–501.

[23] Z. Pan and R. Eigenmann, "Fast and effective orchestration of compiler optimizations for automatic performance tuning," in *The 4th Annual International Symposium on Code Generation and Optimization (CGO)*, Mar. 2006, pp. 319–330.

[24] "OpenMP. http://openmp.org/wp/."

[25] Z. Pan, B. Armstrong, H. Bae, and R. Eigenmann, "On the interaction of tiling and automatic parallelization," in *First International Workshop on OpenMP*, 2005, pp. 24–35.

[26] H. Bae and R. Eigenmann, "Performance analysis of symbolic analysis techniques for parallelizing compilers," in *Proc. of the Workshop on Languages and Compilers for Parallel Computing(LCPC'02)*, Aug. 2002, pp. 280–294.

[27] C. Dave and R. Eigenmann, "Automatically tuning parallel and parallelized programs," in *LCPC '09: Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing*, 2009.

[28] "OpenUH compiler suite : User guide," 2007. http://www2.cs.uh.edu/~openuh/OpenUHUserGuide.pdf

[29] "PGI® compiler reference manual: Parallel fortran, c and c++ for scientists and engineers," 2012. http://www.pgroup.com/doc//pgiref.pdf

[30] 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,"

The International Journal of Supercomputer Applications, Tech. Rep., 1991.

[31] R. F. V. der Wijngaart, "NAS Parallel Benchmarks version 2.4," Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division, Tech. Rep., 2002.

[32] S. Satoh, "NAS Parallel Benchmarks 2.3 OpenMP C version. http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp," 2000.

[33] C. Tapus, I.-H. Chung, and J. K. Hollingsworth, "Active harmony: Towards automated performance tuning," in *In Proceedings from the Conference on High Performance Networking and Computing*, 2003, pp. 1–11.

[34] A. Tiwari, J. K. Hollingsworth, C. Chen, M. W. Hall, C. Liao, D. J. Quinlan, and J. Chame, "Auto-tuning full applications: A case study," *IJHPCA*, vol. 25, no. 3, pp. 286–294, 2011.

[35] Q. Yi, "Automated programmable control and parameterization of compiler optimizations," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*, april 2011, pp. 97 –106.

[36] I. Dooley and L. V. Kale, "Control points for adaptive parallel performance tuning," November 2008.

[37] F. Bodin, T. Kisuki, P. Knijnenburg, M. O'Boyle, and E. Rohou, "Iterative compilation in a non-linear optimisation space," 1998.

[38] L. Almagor, K. D. Cooper, A. Grosul, T. J. Harvey, S. W. Reeves, D. Subramanian, L. Torczon, and T. Waterman, "Finding effective compilation sequences," *SIGPLAN Not.*, pp. 231–239, 2004.

[39] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones, "Fast and efficient searches for effective optimization-phase sequences," *ACM Trans. Archit. Code Optim.*, pp. 165–198, 2005.

[40] J. Cavazos, J. E. B. Moss, and M. F. P. O?oyle, "Hybrid optimizations: Which optimization algorithm to use?" in *IN 15TH INTERNATIONAL Conference on Compiler Construction (CC 2006)*, 2006.

[41] P. Kulkarni, W. Zhao, S. Hines, D. Whalley, X. Yuan, R. V. Engelen, K. Gallivan, J. Hiser, J. Davidson, B. Cai, and M. Bailey, "Vista: Vpo interactive system for tuning applications," *ACM Transactions on Embedded Computing Systems*, vol. 5, p. 2006, 2005.

[42] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. P. O'Boyle, "Portable compiler optimisation across embedded programs and microarchitectures using machine learning," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42, 2009, pp. 78–88.

[43] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *SIGPLAN Not.*, pp. 177–187, 2009.

[44] C. Liao, D. J. Quinlan, R. W. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization." in *LCPC'09*, 2009, pp. 308–322.

[45] Y.-J. Lee and M. W. Hall, "A code isolator: Isolating code fragments from large programs." in *LCPC'04*, 2004, pp. 164–178.

[46] P. Zhao and J. N. Amaral, "Ablego: a function outlining and partial inlining framework: Research articles," *Softw. Pract. Exper.*, vol. 37, no. 5, pp. 465–491.

[47] I.-H. Chung and J. K. Hollingsworth, "Using information from prior runs to improve automated tuning systems," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ser. SC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 30–.

[48] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for DSP algorithms," 2001.

[49] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle, "Combined selection of tile sizes and unroll factors using iterative compilation." in *IEEE PACT'00*, 2000, pp. 237–248.