Int J Parallel Prog
DOI 10.1007/s10766-014-0307-8

PETRA: Performance Evaluation Tool for Modern Parallelizing Compilers

Dheya Mustafa · Rudolf Eigenmann

Received: 18 July 2013 / Accepted: 28 February 2014

Abstract This paper describes PETRA: a portable performance evaluation tool for parallelizing compilers and their individual techniques. Automatic parallelization of sequential programs combined with performance tuning is an important alternative to manual parallelization for exploiting the performance potential of today's multicores. Given the renewed interest in autoparallelization, this paper aims at a comprehensive evaluation, identifying strengths and weaknesses in the underlying techniques. The findings allow engineers to make informed decisions about techniques to include in industrial products and direct researchers to potential improvements. We present an experimental methodology and a fully automated implementation for comprehensively evaluating the effectiveness of parallelizing compilers and their underlying optimization techniques. The methodology is the first to (1) include automatic tuning, (2) measure the performance contributions of individual techniques at multiple optimization levels, and (3) quantify the interactions of compiler optimizations. The results will also help close the gap between research compilers and industrial compilers, which are still far behind. We applied the proposed methodology using PETRA on five modern parallelizing compilers and their tuning capabilities, illustrating several use cases and applications for the evaluation tool. We report speedups, parallel coverage, and the number of parallel loops, using the NAS Benchmarks as a program suite. We found parallelizers to be reasonably successful in about half of the given science-engineering programs. An important finding is also that some techniques

This work was supported, in part, by the National Science Foundation under grants No. CNS-0720471, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

D. Mustafa (✉) · R. Eigenmann
Purdue University, West Lafayette, IN 47907, USA
e-mail: dmustaf@purdue.edu

R. Eigenmann
e-mail: eigenman@purdue.edu

Published online: 19 March 2014

substitute each other. Furthermore, we found that automatic tuning can lead to significant additional performance and sometimes matches or outperforms hand-parallelized programs. Advanced versions of some of the techniques identified as most successful in previous generations of compilers are also most important today, while other techniques have risen significantly in impact. Finally, we analyze specific reasons for the measured performance and the potential for improvement of automatic parallelization.

Keywords Automatic parallelization · Optimization techniques · Performance evaluation · Evaluation methodology · NPB benchmarks

1 Introduction

The advent of multicore processors has put a renewed focus on automatic parallelization [7,10,15,16,36,39–41]. Autoparallelizers still do not achieve the consistent performance necessary to be considered true alternatives to hand parallelization. In many programs, parallelizing compilers are only finding partial parallelism. One reason is that parallelism detection techniques are limited by insufficient static knowledge. Another important reason is that, even where compilers find parallelism, they cannot guarantee that the generated parallel code results in increased performance. Users may experience performance degradation, unless they invest substantial time in tuning their parallel programs. Recent work has therefore proposed automatic tuning techniques to be integral parts of autoparallelization tools [8,20,38].

A thorough understanding of such tools and their underlying techniques is critical for both researchers and engineers. Trading off the costs of implementing a compiler technique versus the performance loss of excluding it is key to the design of a commercial compiler. At the same time, research can be guided toward more important techniques and away from less effective ones. To the best of our knowledge, no such study has been conducted on modern parallelizing compilers for today's multicore systems.

A study on a previous generation of compilers had discussed the effectiveness of automatic parallelization and restructuring techniques on the Perfect Benchmarks [2, 9]. The techniques analyzed included *reduction substitution, recurrence substitution, induction variable elimination, scalar expansion, forward substitution, stripmining*, and *loop interchange*. The study found that three out of the ten programs showed respectable improvements and that the scalar expansion technique proved to be the most effective, followed by reduction substitution. Other work aimed at evaluating the overall performance of parallelizing compilers in terms of improved loop timings [13, 22,37] and the number of loops that can be parallelized [3]. Such metrics are found to be insufficient for modern architectures; more parallelized loops do not necessarily mean higher performance, due to growing parallelization overhead with increasing numbers of cores. Also, different techniques may have similar effects on a program. We need to understand if one technique can replace another, and quantify the degree to which such substitution may occur.

Many techniques and options of today's compilers are not simple on/off choices. For example, data privatization is a *multi-level* technique; scalars and/or arrays can be

privatized. Tiling is an example of a *multi-value* technique; the tile size parameter can take many different values. Understanding the performance effects of such parameters is of interest, but has not been evaluated in previous studies. Furthermore, as the performance behavior of different programs varies significantly, it is important to consider the techniques' average performance as well as their effect on individual programs. Doing so involves large-scale experimentation, which increases further, as techniques tend to interact; thus, their effects need to be considered in many possible combinations.

Building on the methodologies applied in previous studies [29,30], the present paper describes a comprehensive performance evaluation tool for modern autoparallelizers and their underlying techniques for today's multicores. We present overall program performance and identify the contributions of individual techniques. We evaluate both the parallelization and the tuning capabilities. Our methodology requires extensive measurements; automating this process is important. To this end, we make use of a *portable empirical tuning system* implemented in prior work [21]. As a tuning tool, this system can navigate through a large search space of optimization variants and pick the best. Used for our evaluation study, the same tool supports a comprehensive, automatic exploration of compiler techniques and their effects on individual loops in a given program suite. Furthermore, by exploring combinations of techniques in many variants, this method can identify interactions among techniques.

We applied the proposed methodology to measure the effectiveness of two commercial compilers, Intel's ICC [12] and PGI [27], and three research parallelizing compilers, Cetus [7], OpenUH [16], and Rose [34], including an automatic tuner. For measuring individual techniques and their parameters, we used the Cetus compiler, which proved to be the most advanced. ICC performed better than GCC and thus was selected as the backend compiler for the source-to-source translation. As a test suite, we used the NAS Parallel Benchmarks (NPB). We chose this suite because it includes OpenMP program variants that are hand optimized, which serve as reference points for automatic parallelization. We measured the overall effectiveness of the parallelizers in terms of the achieved program speedup, the number of parallelized loops, and parallel coverage, which we compare to the hand-parallelized program variants. We measured individual techniques in terms of their contributions to the overall speedup and execution overhead. Finally, we discuss the reasons behind the gap between automatic and hand-parallelized programs. We identify both challenges faced by current autoparallelizers and opportunities for future improvements.

In summary, we make the following contributions:

- We introduce an experimental methodology for quantitatively evaluating the effectiveness of parallelizing compilers and their underlying optimization techniques as well as their interactions.
- We describe PETRA, an automatic evaluation tool that implements the proposed methodology in a way that facilitates portability and re-usability across different architectures and compilers.
- We apply PETRA on both industrial and research compilers. We comprehensively evaluate the performance of modern autoparallelizers, including automatic tuning capabilities. We evaluate the performance contribution of individual optimization techniques and discuss their interactions.

The remainder of the paper is organized as follows: The next section presents the measurement methodology and evaluation tool implementation. Section 3 provides a brief description of parallelizing compilers and their techniques being evaluated in this paper. Overall performance results of parallelizing compilers as well as individual performance contributions and interactions of techniques are presented in Sect. 4. Section 5 provides recommendations for potential improvements of autoparallelizers. Related work is discussed in Sect. 6, followed by conclusions in Sect. 7.

2 Measurement Methodology

The proposed evaluation methodology is motivated by the needs of engineers and researchers to better understand parallelizing compilers and the performance behavior of their constituent techniques, as described in the introduction. One wants to know the overall success of autoparallelizers, the performance contributions of individual techniques, the techniques' impact on each other, the effect of the many parameters the techniques can take, and the compilation time increase due to individual techniques.

The key part of the methodology is a set of well-defined metrics that are able to put the answers to these questions in quantitative terms. As gathering the data for quantitative answers involves large-scale experimentation, another essential part of the methodology is the automation of its implementation in a way that can be easily applied to a new compiler. The following two sections describe these parts.

2.1 Metrics

Three metrics quantify the overall success of a parallelizer: speedup, number of parallelized loops, and parallel coverage. Speedup shows the performance achieved over the serial program. The number of parallel loops demonstrates the compiler's ability to detect parallelism, while parallel coverage is the fraction of serial execution time enclosed by parallel constructs—indicating the performance potential on an idealized parallel machine.

Several metrics quantify individual techniques. One metric is the increase in program translation time due to including a technique in the compilation process.

We also introduce the *Performance Drop Ratio* ($\Phi_{(T)}$). This metric measures the performance reduction when technique T is turned off while all other optimizations remain the same.

$$\Phi_{(T)} = \left(\frac{Speedup_{(All\text{-}Tuned)} - Speedup_{(T\text{-}Off)}}{Speedup_{(All\text{-}Tuned)}} \right) \tag{1a}$$

$$\overline{\Phi}_{(T)} = 1 - \Phi_{(T)} \tag{1b}$$

Some optimization techniques will take over when others get disabled. We call this situation *substitution*. We define A and B as substituting techniques if the combined effect of disabling both, $\overline{\Phi}_{(A,B)}$, is significantly less than the combined individual performance $\overline{\Phi}_{(A)} * \overline{\Phi}_{(B)}$. For example, assume that $\Phi_{(A)}$ is 10 % (i.e. performance

drops 10 % from the best tuned version, when switching A off) and $\Phi_{(B)}$ is 15 %. Then, the two techniques are substituting if $\Phi_{(A,B)}$ is substantially more than $1-(1-10\,\%)*(1-15\,\%)=23.5\,\%$.

Several variants of speedup are also used to quantify the performance contribution of individual techniques. We turn off one technique at a time and report the speedup [Speedup(T_{off})]. We also report the worst-case possible speedup when tuning the compiler using the poorest combination of optimization parameters (from within reasonable value ranges). For multi-level techniques we report the speedups of the different levels in stacked bars. Furthermore, we count the number of parallelized loops before and after each technique is turned off.

2.2 System Workflow

The implementation of our methodology includes three phases: Preliminary Pruning (PP) [31], Version Exploration (VE), and Data Visualization. The PP phase excludes techniques with a consistent negative effect. The VE phase navigates over a large set of program variants in short time. It also generates a performance database about each program variant that was measured. We use this database to evaluate the effectiveness of the parallelizing compilers and their underlying techniques. We provide several ways to visualize the data. The major components of the methodology are automated in a prototype that is portable across different compilers and scalable to large numbers of optimization techniques.

The prototype is able to keep the tuning time low. Generally, tuning times can be very long, as optimization techniques tend to interact and thus all combinations of techniques on all loops would need to be tried for finding the best combination. The key point is that code sections in close proximity and optimization techniques are likely to interact. The proposed system tunes at code section granularity. It includes a feature for the user to customize the search space by defining optimization techniques and their interactions that should be tuned; this usually happens at compiler setup time. We also use this customization feature to exhaustively explore the optimization techniques studied in this paper. Specifics of this method are described in [21].

The negative effect of one optimization can be represented by its Φ introduced earlier. The PP phase measures Φ for each technique and removes those with $\Phi \leq 0$ in all programs. This algorithm has a complexity of $O(n)$, where n is the number of optimizations. This phase considers all optimization options to be binary (i.e. a multi-value optimization will be either turned off or on with the default value).

The VE phase allows users to define architectural characteristics and compiler options of the system being evaluated. The user can also define the measurement granularity (program level or section level). Based on this information, VE automatically and efficiently navigates the space of optimizations and their combinations, recording the measured performance. The fully optimized and tuned programs are selected as the baseline of our measurements for the VE phase. Starting from these programs, VE disables one compiler technique at a time, and measures the performance of the resulting parallelized programs. The measured values (Φ) represent the performance loss due to the disabled technique. VE includes a set of features that facilitate comparing the

resulting, transformed code with the base code and the original parallel code to examine the differences. For example, one can query the large database for the best performance configuration, the worst performance of a specific technique, and sort the configurations based on performance. Other features help count the number of loops and explore the extensive results, presenting them in a well-readable form in tables and graphs.

3 Compilers and Techniques Being Evaluated

We applied PETRA to evaluate five parallelizers—three research compilers (Cetus, Rose, and OpenUH), and two commercial compilers (Intel's ICC and the PGI parallelizer). For each compiler, we evaluated the techniques that are accessible through command line options. The available options differ significantly across compilers. This is because the compilers employ different techniques, but also because of the developers' choices of which options may be of interest to the user. Such variety provides a sufficient test environment for our proposed tool. PETRA can easily handle this diversity. For each compiler, we will show the available options that have the highest impact. To obtain the best possible performance for each of the compilers, we used an automatic tuning system (see Sect. 3.6). For the detailed study of individual techniques, we used Cetus, as it is the most successful parallelizer among the five compilers. The following subsections introduce these compilers.

3.1 Cetus Compiler

The Cetus compiler infrastructure is a source-to-source translator for C programs. The translator supports the detection of loop-level parallelism and the generation of C code annotated with OpenMP parallel directives [23].

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 [7].

3.2 OpenUH Compiler

OpenUH is a branch of the Open64 compiler maintained by the High Performance Computing Tools (HPCTools) group of the University of Houston. OpenUH 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 translator to produce compilable code immediately before the code generator [16,24].

The list of optimizations evaluated and presented are autoparallelization (*apo*), loop unrolling (*unroll*), loop nest optimization (*interchange, blocking, prefetch*), function inlining (*inline*), optimization level (*O*, *Ofast*), inter-procedural analysis (*alias, constant propagation, dead function elimination*) and variable reduction (*reduction*). The terms in parentheses will be used to refer to the corresponding techniques in the presented figures later.

3.3 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 [12].

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. The selected compiler options included in our study 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*).

3.4 PGI Compiler

The PGI compiler incorporates global optimization, vectorization, software pipelining, and shared-memory parallelization capabilities. Performance for loops on systems with multiple processors may improve using the parallelization features of the PGI compiler. Vectorization transforms loops to improve memory access performance and make 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 to provide better opportunities for local optimization, vectorization and scheduling of instructions [26,27].

Compiler options included in the evaluation are auto-parallelization (*concur*), vectorization (*vect*), function inlining (*inline*), loop unrolling (*unroll*), inter-procedural analysis and optimization (*ipa*), cache align (*align*), partial redundancy elimination (*PRE*), SSE flush (*flushz*) and optimization level (*O*). For more detailed descriptions of these options, we refer to the PGI user guide [27].

3.5 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 [34]. Rose provides several optimizations including

Table 1 List of compilers and techniques considered in the evaluation study

Compiler	Techniques
PGI	*-concur*, *-vect*, *-inline*, *-unroll*, *-ipa*, *-align*, *-PRE*, *-flushz*, *-O3, -O2, -O1*
ICC	*-parallel*, *-unroll*, *-blocking*,*-vec*, *-prefetch*, *-scalar-rep*, *-align*, *-inline*, *-O3, -O2, -O1*
OpenUH	*-apo*, *-unroll*, *-lno*, *-interchange*, *-blocking*, *-prefetch*, *-inline*, *-Ofast*, *-ipa*, *-alias*, *-cp*, *-dfe*, *-reduction*, *-O3, -O2, -O1*
Rose	*parallelize*
Cetus	*-alias*, *-ddt*, *-parallelize-loops*, *-privatize*, *-range*, *-reduction*, *-induction*, *-tinline*

Specifics of these techniques and their arguments are describes in corresponding compilers' programmer guide

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 evaluate autoparallelization only. Rose successfully compiles five out of the eight NPB. A summary of all compilers and techniques considered in the evaluation is listed in Table 1.

3.6 Automatic Tuning

As mentioned in the introduction, a key challenge of optimizing compilers is to decide where and when to apply their techniques, given limited knowledge about the program's input data and runtime environment. Among these decisions are the best use of loop interchange and tiling, which interact strongly with parallelization [28]. Also, function inlining and reduction parallelization may cause overheads if applied indiscriminately. In general, eager parallelization of small, inner loops is not beneficial; the user may see performance degradation instead of speedup.

Improved runtime decision making can be done through an automatic tuning capability that searches through the space of all (or a customizable set of) optimization techniques and loops, and finds the combination that performs the best at runtime. This process is done in an offline manner, similar to profile-based compilation. Like profile-based optimization, the tuning process uses a *training data set*, which is representative of but different from production data sets. We provide an automatic tuning system [20,21] that enables all five compilers to maximize performance in this way.

4 Results

We have applied the described methodology for comprehensively evaluating five compilers. After a brief description of the benchmarks and experiment setup, we discuss several use cases and applications of the PETRA evaluation tool using several measurements defined by the methodology. First, we show the overall performance evaluation of optimizing compilers and their individual techniques. Then we take a deeper look into the Cetus evaluation to illustrate more capabilities of the PETRA tool, such as multi-level techniques, contributions to speedup, and translation time.

4.1 Benchmarks and Experiment Setup

The NPB are designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from Computational Fluid Dynamics (CFD) applications. We use four classes of problems in the evaluation: Class W, A, B, and C. The classes differ mainly in the sizes of principle arrays, which generally affects the number of iterations of contained loops [6]. We use the NPB suite implemented in the C language [35].

The correctness of generated code by all compilers was verified using tools and pre-generated output shipped with the NAS Suite. In all compilers, the common errors are those introduced by floating point optimizations. Such errors are not tolerated.

Several of the parallelizers we study are source-to-source translators. We used ICC as a backend code generator. Comparing GCC to ICC, we found that ICC consistently performs better. We only show ICC results for space limitations.

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 OS is Red Hat Enterprise Linux. We used the Intel ICC compiler version 11.1, OpenUH compiler version 4.2, PGI compiler version 9.0-3 64 bit, Cetus compiler version 1.3 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 for our measurements. To compute speedups, we created the serial program versions by compiling with option O3, only.

4.2 Overall Evaluation of Optimizing Compilers and Their Techniques

This section presents speedups and number of parallelized loops for all compilers being studied. We also discuss the performance drop ratio for several techniques in all compilers. Finally, we present the techniques with multiple values and the worst possible speedup for each compiler, which provides an estimate of potential degradation that a compiler could cause when used eagerly.

4.2.1 Speedups

We measured speedups of the parallelized and tuned programs over the sequential versions generated by ICC with option -O3. Figure 1 shows the speedups for all compilers using the NAS benchmarks. Cetus shows significant performance for BT, CG, EP, and SP. OpenUH shows performance improvement for CG, and SP. The PGI and ICC compilers show performance improvement for SP. Cetus improves performance by 61 % over the OpenUH compiler, 162 % over the ICC compiler, 154 % over the Rose autoparallelizer, and 151 % over the PGI Compiler.

4.2.2 Number of Parallel Loops

Table 2 shows the number of parallelized and vectorized loops by the five compilers. Cetus results are discussed in Sect. 4.3 in more detail. We counted individual loops (i.e. a doubly nested loop is counted as two loops).

The PGI, ICC, and Rose compilers attempted to parallelize/vectorize as many loops as possible, while OpenUH, Cetus and hand programmers tried to parallelize the outer-

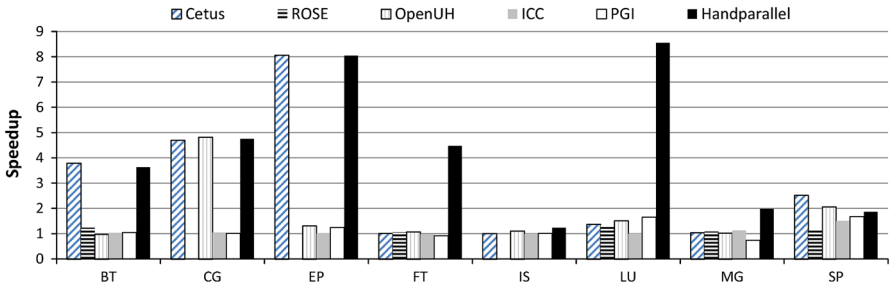

Fig. 1 Speedups of the NAS benchmarks optimized with the *Cetus*, *OpenUH*, *Rose*, *ICC* and *PGI* compilers. Hand parallel shows the speedups of the original benchmarks. Cetus achieves high performance in *BT*, *CG*, *EP* and *SP* relative to hand parallel. The commercial compilers (ICC, PGI) are unable to gain significant performance from automatic parallelization

Table 2 Number of parallelized and vectorized loops using Rose, ICC, PGI, Cetus, and OpenUH

Loop type	Vectorized/parallelized					Hand parallel	Total number of loops
	PGI	ICC	OpenUH	Rose	Cetus		
BT	0/105	18/20	0/30	0/155	0/63	54	223
CG	10/0	31/12	0/24	NA	0/14	24	46
EP	4/0	9/2	0/1	NA	0/7	1	11
FT	0/0	6/0	0/7	0/5	0/5	6	51
IS	3/0	11/4	0/6	NA	0/8	1	4
LU	19/93	50/0	0/15	0/143	0/39	29	173
MG	14/14	26/5	0/1	0/39	0/10	12	83
SP	0/130	149/50	0/73	0/249	0/80	70	314
Total	226/342	300/93	0/157	0/591	0/226	197	905

Hand Parallel represents the number of parallel loops in the original code. Vectorization is not available in the OpenUH, Cetus and Rose compilers

most loop within a nest. This explains the noticeable difference between the two groups of compilers in terms of the number of loops. The PGI and ICC compilers also vectorize loops.

Relating speedup to the number of parallel loops, we find that in the hand-parallelized NPBs, only 22 % of the loops are parallel. This is consistent with the common execution profile, where a small number of program sections consume most of the execution time; tuning is most effective in these sections. Rose parallelizes the highest number of loops, even though its performance is the lowest. Most of these loops are found to be small inner loops within a big loop nests. Such loops achieved low parallel coverage.

4.2.3 Performance Drop Ratio

Figure 2 shows the average performance drop ratio Φ of the optimization options for the Cetus, ICC, OpenUH, and the PGI compilers across the NAS suite. $\Phi_{(T)}$ measures

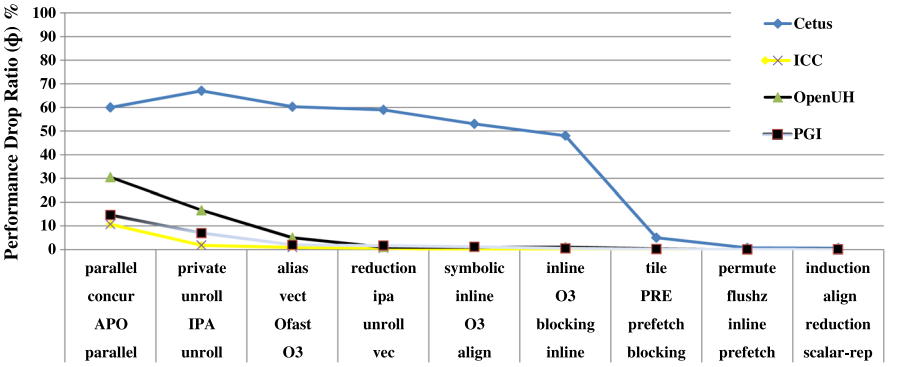

Fig. 2 Performance Drop Ratio (Φ) for *Cetus*, *OpenUH*, *PGI*, and *ICC* compilers on average over the NAS suite. To measure $\Phi(T)$, technique T is turned off while all other options are set to the best tuned configuration. Techniques to the left are most important

the performance impact of technique T. We discuss techniques shown in the first three columns, as they have the highest effects.

The first column shows the overall impact of autoparallelization. Inter-procedural analysis enables other techniques; inlining takes the place of IPA in Cetus. Privatization, alias analysis and reduction substitution are very important techniques; they show around 50 % performance drop in Cetus. Loop unrolling helps sometimes produce suitable loads for the powerful cores in modern architectures, and may lead to more instruction cache misses. Memory optimizations such as blocking, tiling, permutation, prefetching and aligning show no or minor influence on performance. Several factors could contribute to this result, including the benchmark, the architecture, and the compilers themselves. We included dataset C in our tests, which is large and thus requires locality enhancement techniques to be applied. We do not believe the architecture to be a factor, as the used X86 platform is common. We produced the serial versions of the NPB suite by removing all openmp pragmas; locality might have already been enhanced in the NPB. Finally, we found that compilers give higher priority to parallelization over locality enhancement when these techniques interact.

Similar and even identical optimization techniques for different compilers have different performance contributions. We attribute this primarily to differences in implementations. Induction variable substitution in Cetus, and *O*3 in the PGI compiler show low or no influence on performance. We attribute this result to the fact that most induction variables had already been eliminated in the NPB. It is common that compilers switch on certain optimization levels when advanced options are chosen. In the PGI compiler, unrolling enables O2 or O3, hence these techniques are substituting. Other substituting techniques include Ofast, IPA, and O3 in the OpenUH compiler.

4.2.4 Techniques with Multi-Value Arguments

Many optimization techniques are controlled by one or more command-line arguments that can take a range of values. Table 3 lists all multi-value techniques and the range of values considered in the evaluation. The combination of these techniques and their

Table 3 Multi-value techniques with their range of values considered in the evaluation of the ICC, PGI, Cetus, and OpenUH compilers

Compiler	Technique	Test range
PGI	Optimization level	[1:3]
ICC	Optimization level	[1:3]
	Parallelization threshold	[0:100]
	Loop unroll	[0:8]
	Block factor	[0:256]
	Vectorization threshold	[0:100]
	Optimization prefetch	[0:4]
	Inline level	[0:2]
OpenUH	Optimization level	[1:3]
	Blocking size	[0:256]
	Loop unroll level	[0:2]
Rose	Parallelize	Nested, not nested
Cetus	Tile size	[0:256]
	Permute	[0:32]

Table 4 Worst-case performance for Rose, ICC, PGI, Cetus, and OpenUH compilers

This measures the potential performance drop when tuning naively. Each row represents the speedups over serial execution of the corresponding benchmark for all compilers. The "Average" column represent the average worst speedups across all compilers, while the row "Average" represents the average worst speedups across benchmarks

Benchmark	Compilers worst speedup					Average
	PGI	ICC	OpenUH	Rose	Cetus	
BT	0.6	0.04	0.83	0.09	0.09	0.33
CG	0.46	0.29	1.01	NA	1.97	0.81
EP	1.23	0.75	1.02	NA	0.19	0.80
FT	0.65	0.26	0.96	0.96	0.04	0.57
IS	0.94	0.50	0.97	NA	1.0	0.85
LU	0.47	0.11	0.91	0.05	0.95	0.50
MG	0.55	0.07	0.81	0.08	0.15	0.33
SP	0.51	0.18	0.77	0.08	1.15	0.54
Average	0.67	0.27	0.91	0.26	0.69	

corresponding range of values compose a huge optimization search space, which is efficiently navigated by PETRA.

Recall that Φ measures the performance impact of an optimization technique. To understand the potential negative effect of a technique when used eagerly, we measured the worst-case performance within these ranges, shown in Table 4. Several industrial compilers apply their techniques conservatively, not choosing the advanced options available in research compilers. The main reason is to avoid unexpected performance drops. This choice shows the necessity for tuning. The IS benchmark shows the highest worst-case speedup of 0.85 on average for all compilers. This code is not amenable to optimization.

4.3 A Closer Look at an Advanced Parallelizer

The Cetus parallelizer has shown the best performance. We use this compiler to discuss a number of advanced topics. Several of the optimization techniques include multiple

Table 5 Optimization options that include multiple levels

Option	Level0	Level1	Level2
Alias analysis *(alias)*	Conservative	Advanced	Assume no aliases
Data dependency test *(parallel)*	OFF	Banerjee test	Range test
Data privatization *(privatize)*	OFF	Scalar privatization	Scalar and array privatization
Symbolic analysis *(range)*	Minimal	Local analysis	Inter-procedural analysis
Reduction parallelization *(reduction)*	OFF	Scalar reduction	Scalar and array reduction
Subroutine inlining *(inline)*	OFF	Eager	Selective inlining
Induction variable recognition *(induction)*	OFF	Simple	Generalized IV

Most levels are self descriptive. Banerjee and Range tests are well known data dependency tests

levels; we measure the impact of the different levels. We also assess the degree to which some techniques may be substituted by others. Furthermore, we measure parallel coverage of the benchmarks, indicating the speedup of the best parallelized programs on an ideal parallel machine. Last but not least, we measure the cost of individual optimization techniques in terms of their compilation run times.

4.3.1 Evaluating Individual Techniques with Multiple Levels of Optimization

We studied the performance contributions of individual techniques with multiple levels of optimizations in the Cetus compiler. Table 5 shows a list of optimization techniques in Cetus and their levels. Three classes of techniques are measured: Program analysis, parallelism enabling, and locality enhancing techniques.

Figure 3 shows the contribution of each technique for all levels for the NAS benchmarks. A technique with lower L0 value (the black column) has a higher impact on performance. The figure shows averages across all benchmarks.

The transformation technique with the highest impact is Scalar and Array Privatization. This is consistent with earlier studies which found Scalar Privatization to be important as well [2,9]. Similarly, Reduction Parallelization is the second most important transformation technique, affecting two of the programs.

All analysis techniques are important. Symbolic analysis (range test) affects half of the programs. Inlining affects all programs, excepts those where parallelization makes no difference. The same holds for alias analysis. After thorough analysis of the results, we find that:

- The Advanced forms of parallelism-enabling transformation techniques that were found important in the earlier performance studies are most important today. They will be the first to implement for an effective parallelizing compiler.
- The analysis techniques tell a slightly different story. Forward substitution can be considered a simple version of symbolic analysis. While it had no significant impact in the early Kap compiler [2], it is important now. We attribute this to two reasons: (1) the technique needed to mature to a certain point, where it begins to

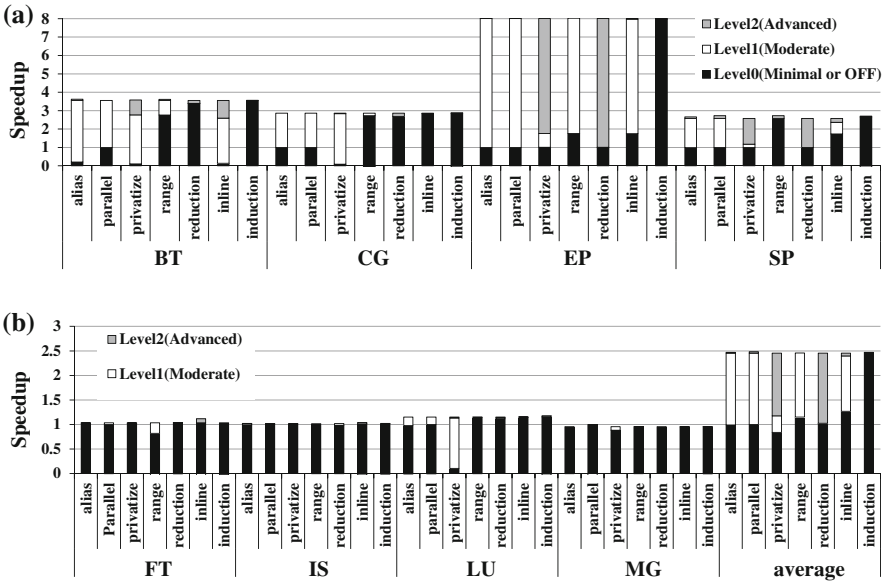

Fig. 3 Stacked detailed performance contributions of different optimization levels for Cetus techniques using NAS suite. Three levels of optimizations are evaluated for each technique as explained in Table 5. We presented dataset A results for NPB benchmarks; all other datasets provide coherent results. Array privatization and reduction are more important than the scalar ones. The advanced levels have a minor effect for alias, parallel and inline

make a difference, and (2) the simpler optimization techniques in Kap did not need advanced analysis, whereas the more aggressive techniques in Cetus do.

– Inlining takes the place of inter-procedural analysis. Even though inlining was available in Kap, it was not included in the earlier study. Indiscriminate inlining can have a negative impact on performance. Therefore, Cetus subjects inlining to tuning as well; it is applied only where of benefit and, in this form, contributes significantly to the obtained performance in two programs.
– Alias analysis was not present in Kap, as it was a FORTRAN compiler; the FORTRAN language defines that subroutine parameters can be safely assumed as non-aliased by the compiler. This is not so in C programs. Not having alias information available would force the compiler to make too many conservative assumptions about data dependencies.
– Loop interchange was measured in the study of Kap, but did not make a difference. Because of the higher processor-to-memory speed ratio today, we expected this technique, as well as tiling, to make a bigger difference. One possible explanations for the small measured impact is that the NAS benchmarks have already been optimized for locality.

4.3.2 Substituting Techniques

Table 6 shows the performance drop ratio for loop interchange (LI), and loop tiling (LT). For SP, $\Phi_{(LI)}$ is 4.4 %, while $\Phi_{(LT)}$ 1.8 %. $\Phi_{(LT,LI)}$ is 24.7 %, which means

Table 6 Effect of loop tiling (LT) and interchange (LI) on performance

Both techniques are clearly substituting in the SP benchmark

	BT	CG	EP	FT	IS	LU	MG	SP
$\Phi_{(LT,LI)}$	0	6.3	0	1.9	0	1.6	4.4	24.7
$\Phi_{(LT)}$	0	2.5	0	0	0	23	29	1.8
$\Phi_{(LI)}$	0	0	0	0.9	0	1.6	11	4.4

that both techniques substitute each other significantly. This is not surprising, as both enhance locality. Tiling is a more advanced technique and includes loop interchanging, even though the techniques don't aim at the same type of locality. Apparently, SP exhibits simple locality patterns, which can be handled by both techniques. In general, the effects of tiling and interchange are small. BT, EP, FT, and IS are not affected by either techniques.

Loop autoparallelization and vectorization are potential substituting techniques as well. Turning both techniques on will leave the decision of parallelizing or vectorizing a candidate loop to the profitability model in the compiler. Turning one technique off, the compiler will try to apply the other technique to all loops.

Another form of substitution happens with group options (i.e. an option that enables multiple techniques). For example, inter-procedural analysis implicitly enables many techniques, including inlining. Another example is optimization level O3, which implicitly switches on several techniques. This form of substitution is described in the PGI compiler manual, where both O3 and inline techniques are substituting [26].

4.3.3 Parallel Coverage

To measure parallel coverage, the parallelized loops need to be instrumented in the translated source code. Our experiments support this feature for source-to-source compilers; Cetus and Rose. Figure 4 shows the parallel coverage of the NPB benchmarks using dataset A. We compare the parallel coverage for hand-parallel programs, and automatic parallel programs for Cetus and Rose. 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. BT, CG, and SP show high parallel coverage. Despite good parallel coverage for LU and FT, the performance is low as depicted in Fig. 1.

4.3.4 Translation Time

The *cost* of advanced optimization techniques is the added translation time. Figure 5 compares the translation time of individual techniques at advanced and basic levels as defined in Table 5. We measure one technique at a time, while keeping all other techniques set to the best tuned configuration.

On average, inlining consumes the most translation time. One reason being the programs' complex call trees. Also, inlining adds extra work to other techniques; it increases code size by 130 % on average and 250 % in the worst case.

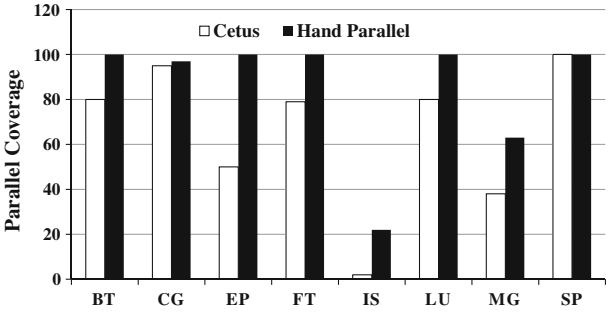

Fig. 4 Parallel coverage for NPB benchmarks using dataset A. *Hand Parallel* shows parallel coverage of the original benchmarks. *Cetus* and *Rose* show parallel coverage of the automatically parallelized codes. Higher parallel coverage indicates potential for higher performance

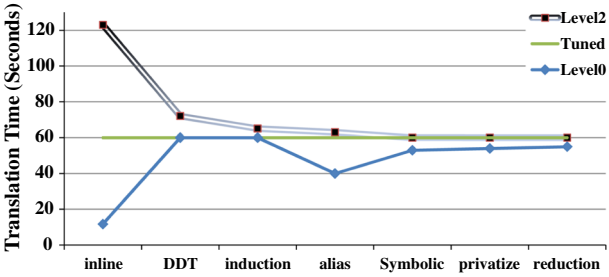

Fig. 5 Cetus average translation time for NAS parallel suite. *Level2* is the translation time when using the most advanced level of the optimization, while *Level0* is the translation time for the basic level. *Tuned* is the translation time for the best-tuned optimizations

4.3.5 Number of Parallelized Loops

We also quantify the effectiveness of individual techniques using a static metric: the number of parallel loops. Starting from the base case, where all optimization techniques are on, we counted the number of autoparallelized loops in the transformed programs after turning off one technique at a time. We count a loop nest as a single loop. We also report the number of hand-parallelized loops as a reference point. Figure 6 shows the results for the Cetus compiler over the NAS suite.

Parallel coverage is the ratio of the circular area to the rectangular area. The intersection area is the compiler efficiency from a hand programmer point of view. One can see that the importance of the technique being turned off increases as the number of serial loops increases, and the intersection area shrinks. In some cases, performance drops severely despite the fact that the number of parallel loops are still relatively high. However, the intersection shrinks severely, indicating that the parallel loops are not important.

5 Enhancing Autoparallelizers

Section 4 has shown that autoparallelizers are successful in about 50 % of scientific applications. For such applications, our main concern is to tune compiler techniques to

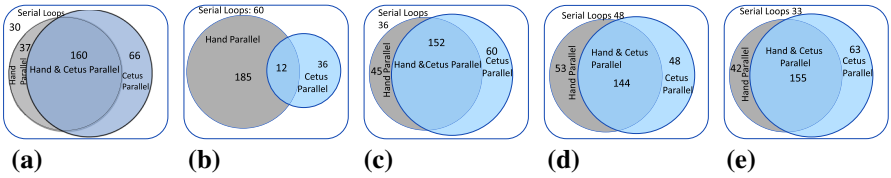

Fig. 6 Effect of individual techniques on the number of parallel loops. One optimization is turned off at a time, and the number of loops were counted. Tuned is the number of loops for the best tuned version. A Technique is more important if the inter-section shrinks more when turned off. **a** Fully Tuned. **b** Privatize Off. **c** Reduction Off. **d** Symbolic Off. **e** Induction Off

maximize performance. Applications that are not amenable to autoparallelization are also not influenced by tuning. Developing more powerful parallelization techniques, such as autoparallelization for irregular applications, is a complementing research direction [14,32,42]. This section is dedicated to study this kind of applications.

In this section, we discuss lessons learned from PETRA to improve autoparallelization. We have seen a few cases where autoparallelizers fail due to code complexity despite the fact it is dependence free. Specifically, we study the reasons behind the poor performance of the parallelized FT, LU, and MG benchmarks. These codes have large numbers of parallelized loops and relatively high parallel coverage. We found that most of the important loops in the three hand-parallelized programs have similar code style, as depicted in Listing 1. The main feature is a declared array inside an *omp parallel* construct, which is later used inside an *omp for* statement. This array is privatized in the hand-optimized code. The arrays' data types could be a primitive or user-defined structure. They also could be statically or dynamically allocated. Autoparallelizers could not privatize these arrays. This is an implementation issue, passing arrays as pointers added more complexity to compile-time analysis.

Listing 1 Template of hand-parallel code that compilers failed to parallelize. The *datatype* could be a primitive or user-defined structure. Array *a* could be statically or dynamically allocated.

```
main(){
.
.
 #pragma omp parallel{
   datatype a[];//could be dynamically allocated.
   foo(a);
 }
.
.
}
foo(datatype * a){
  #pragma omp for
  for(..)
  {
    a[]=f(a independent data access);
    .
    .
    r[]=f(a[]);//r is independent of a
  }
}
```

The following subsections discuss the most important code segments that compilers failed to parallelize, the reasons for such failure, and possible remedies.

Listing 2 Simplified hotspot hand-parallel code segment in FT:cfftsx loop style

```
#pragma omp for
for(k=0;k<N2;k++) {
  for(jj=0;jj<=N1-blk;jj+=blk){
    for(j=0;j<blk;j++)
      for(i=0;i<N0;i++){
        y0[i][j].re=Xin[k][j+jj][i].re;
        y0[i][j].re=Xin[k][j+jj][i].im;
      }
    cfftz(is, logd[0],N0, y0);

    for(j = 0; j < blk; j++)
      for(i = 0; i < N0; i++)
        Xout[k][j+jj][i].re=y0[i][j].re;
        Xout[k][j+jj][i].im=y0[i][j].im;
  }
}
```

5.1 Parallelizing FT

The most important loops compilers failed to parallelize are three loops in the functions cffts1, cffts2, and cffts3. The three loops consume 66 % of the serial execution time. All hand-parallel loops in the input code have the same structure, as depicted in Listing 2.

The first obstacle faced by the compiler is function call *cfftz()* as shown in Listing 2; It was removed by inlining. Cetus still could parallelize only the j-loop and i-loop. It reported a dependency on array *y0*. The benchmark programmer used *y0* as a scratch pad. It is declared inside the *omp parallel* region as an array of complex numbers and passed as an argument to the *cfttsx* functions. They all have the same style described in Listing 1. The Cetus compiler could not privatize y0. Alias analysis found that Xin and Xout are aliases. However, the benchmark developer selects to load a segment of X (Xin) into the scratch pad, apply computation on the scratch pad, and store the results back to X (Xout) sequentially in the inner loops. This benchmark could be autoparallelized by (1) providing user information about aliases to the compiler, and (2) improving privatization and range analysis to handle structures.

5.2 Parallelizing LU

The most important loops the compiler failed to parallelize are two loops in functions *blts* and *buts*. Both hand-parallel loops consume 40.5 % of the program's serial execution time, and have the same loop structure shown in Listing 3.

Listing 3 Simplified hotspot hand-parallel code segment in LU:blts and buts loops style. K is passed as argument

```
#pragma omp for
for(i=ist;i<=iend;i++){
  for(j=jst;j<=jend;j++){
    for (m=0;m<5;m++)
      v[i][j][k][m]+=f(v[i+1][j][k][m],v[i][j+1][k][m],
                       v[i-1][j][k][m], v[i][j-1][k][m]);
    tmp=....
    v[i][j][k][?]+=v[i][j][k][?]*tmp;
  }
}
```

Both hand-parallel loops follow the same style described in Listing 1. The right hand side expression of the statement inside the loop nest is a function of the four elements around V(i,j). It is a stencil loop, where each element is computed as a function of neighboring elements. The compiler reported a dependency on array *V*. The programmer was not concerned about reading old or new values of array V, since these values will eventually converge, permitting race conditions. He/She chose to parallelize the loop. *Semantics pragmas* could be added by the user to help the compiler resolve such issues. Also, loop skewing could be applied to parallelize this loop.

5.3 Parallelizing MG

The most important hand-parallel loops compilers failed to parallelize are in the functions *resid, rprj3, interp,* and *norm2u3*. These hand-parallel loops consume 66 % of the overall serial time. Loops in *interp, resid, and rprj3* contain array dependences. The *rprj3* loop has an array subscript as a complex function of the loop index, which is computed at each nested loop level. Loop *resid* has a dependency on array r which is a write-only array, as depicted in Listings 4. However, r is dynamically allocated and passed as a pointer to the resid function, which leads a conservative dependency analyzer to assume a dependency. The *interp and norm2u3* loops have the same code style described in Listing 1.

Listing 4 Simplified hotspot hand-parallel code segment in MG:resid loop

```
resid(double ***u,double ***r,...){
double a[M];
#pragma omp for
for(i3=0;i3<N3;i3++)
  for(i2=0;i2<N2;i2++){
    for(i1=0,i1=N1;i1++)
      a[i1]=f(u[i3,i2,i1])
    for(i1=0,i1=N1;i1++)
      r[i3,i2,i1]=f(u[i3,i2,i1],a[i1])
  }
}
```

5.4 Summary

Some of the reasons behind the compilers' failure discussed earlier are beyond static analysis or runtime tuning, and require interactive parallelization. Other reasons indicate weaknesses in some of the techniques and the need for advancing those techniques. We also found cases where an algorithm is dependence free, but the complex coding style leads to a conservative analysis, reporting dependences.

6 Related Work

Several studies were conducted on performance evaluation in the last two decades. Our work is the first to: (1) propose a portable tool that can accommodate diverse

compiler techniques and their attributes, (2) include tuning in the evaluation,and (3) evaluate the performance of individual techniques within a compiler.

The effectiveness of parallelizing compilers and their underlying restructuring techniques is studied in [2]; the compiler used for this study is a modified version of Kap, developed by Kuck and Associates. The authors found that scalar expansion had the greatest effect on performance, followed by reduction replacement. We repeat the same plus an extended analysis on modern parallelizing compilers. We found that scalar and array privatization, the advanced version of scalar expansion, is the most effective technique, followed by array reduction transformation.

Two different approaches for evaluating parallelizing compilers are discussed in [1]. One approach counts the number of program segments that can be parallelized, and the other approach measures the execution time of the parallelized code. The authors present several references that include experimental evaluations of compilers following these approaches. In [22], several vectorizing compilers are compared. Our results are consistent with their finding that compilers that vectorize more loops do not necessarily produce faster code. We use a set of derived metrics based on execution time as well as the number of automatically and hand-parallelized loops for accurate evaluation.

A detailed survey about optimizing compilers for parallel machines is presented in [18]. The study compares six compilers available at that time, discusses their features and lists the different optimization techniques applied by them. This is complementary to our study, which experimentally and quantitatively evaluates parallelizing compilers and their underlying techniques and compares the compilers' performance with hand-parallel program variants.

In other research [25], automatic compiler techniques are compared with manual techniques for the development of the parallel implementation of a serial FORTRAN program, and a method is suggested for parallel program development, based on overhead measurement and analysis. This also complements our study, which performs extended measurements over the NAS Parallel suite to evaluate compiled versus hand-parallel code and considers contributions of individual techniques.

In an early paper, Cytron et.al. [5] studied the performance degradation of the EISPACK algorithms after disabling various restructuring techniques of the Parafrase compiler. Of the measured analysis and transformation steps, scalar expansion was the most effective, followed by conversion of control dependence into data dependence, a data-dependence test analysis pass, and the recurrence recognition and substitution pass. Our work is a more comprehensive study on a larger program suite, using modern compilers and architectures.

Recent work evaluates the efficiency of vectorizing compilers using a synthetic benchmark consisting of 151 loops, two applications from Petascale Application Collaboration Teams (PACT), and eight applications from Media Bench II [17]. The paper evaluated three compilers: GCC, ICC and XLC. The authors found that 45–71 % of the loops in the synthetic benchmark and only a few loops from the real applications are vectorized by the compilers they evaluated.

Using machine learning to build models based on compiler heuristics has been studied quite extensively in previous research [4,11,19,33]. Some of these efforts focused on specific optimizations, such as inlining and loop unrolling, for a single compiler. By contrast, PETRA provides a portable empirical tool that makes no prior

assumptions about the architecture, compiler, or application. We used five compilers to demonstrate the capabilities of our tool.

7 Conclusions and Future Work

We have presented a methodology and a tool, PETRA, for evaluating the performance of parallelizing compilers. Using this infrastructure, we have studied the effectiveness of five modern parallelizing compilers and the underlying optimization techniques on the NPB. Cetus, OpenUH, Rose, PGI, and ICC were selected as representatives of the available parallelizing compilers in research and industry. We reported overall performance as well as the impact of individual optimization techniques.

An element not included in previous compiler evaluation studies is *substituting techniques*, where a technique becomes effective in the absent of one or more other techniques. We also evaluated techniques at multiple levels. Another element used for the first time in an evaluation study is the automatic tuning component; it overcomes some of the most severe limitation of an optimizing compiler, which is the lack of runtime knowledge. At the same time, this component supported the automated implementation of our evaluation methodology. We found parallelizers to be reasonably successful in about half of the given science-engineering programs. Advanced versions of the techniques identified earlier as most successful are also most important today, while other techniques have increased in relevance. Cetus achieves 73 % of the Hand Parallel performance on average over the NAS benchmarks.

Currently, we are using the same portable methodology for new evaluation studies, such as measuring the performance of novel architectures, such as Intel Xeon Phi and IBM BlueGene clusters. The study focuses on speedup, scalability, and memory access time. This experimental investigation will help understand the behavior of current and future generation computer systems; lessons will be learned to produce more efficient scientific code. We include SPEC OMP2001 and SPEC OMP2012 benchmarks in the evaluation.

References

1. Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.: Automatic program parallelization. In: Proceedings of the IEEE, pp. 211–243 (1993)
2. Blume, W., Eigenmann, R.: Performance analysis of parallelizing compilers on the perfect benchmarks programs. IEEE Trans. Parallel Distrib. Syst. **3**, 643–656 (1992)
3. Callahan, D., Levine, D.: Vectorizing compilers: a test suite and results. In: SC Conference, pp. 98–105 (1988)
4. Cavazos, J., O'Boyle, M.F.P.: Automatic tuning of inlining heuristics. In: Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference, pp. 14–14 (2005). doi:10.1109/SC.2005.14
5. Cytron, R., Kuck, D.J., Veidenbaum, A.V.: The effect of restructing compilers on program performance for high-speed computers. Comput. Phys. Commun. **37**, 39–48 (1985)
6. der Wijngaart, R.F.V.: NAS Parallel Benchmarks Version 2.4. Technical Report, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division (2002)
7. Dave, C., Bae, H., Min, S.J., Lee, S., Eigenmann, R., Midkiff, S.: Cetus: a source-to-source compiler infrastructure for multicores. IEEE Comput. **42**(12), 36–42 (2009)

8. Dave, C., Eigenmann, R.: Automatically tuning parallel and parallelized programs. In: LCPC '09: Proceedings of the 22nd International Workshop on Languages and Compilers for Parallel Computing (2009)
9. Eigenmann, R., Blume, W.: An effectiveness study of parallelizing compiler techniques. In: ICPP (2)'91, pp. 17–25 (1991)
10. Eigenmann, R., Hoeflinger, J., Padua, D.: On the automatic parallelization of the perfect benchmarks. IEEE Trans. Parallel Distrib. Syst. **9**, 5–23 (1998)
11. Haneda, M., Knijnenburg, P.M.W., Wijshoff, H.A.G.: Automatic selection of compiler options using non-parametric inferential statistics. In: Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on, pp. 123–132 (2005). doi:10.1109/PACT.2005.9
12. Intel C++ Compiler for Linux Systems User's Guide. http://denali.princeton.edu/intel_cc_docs/c_ug/index.htm
13. Kim, S.W., Voss, M., Eigenmann, R.: Performance analysis of compiler-parallelized programs on shared-memory multiprocessors. In: Proceedings of CPC2000 Compilers for Parallel Computers, p. 305 (2000)
14. Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., Casçaval, C.: How much parallelism is there in irregular applications? SIGPLAN Not. **44**(4), 3–14 (2009). doi:10.1145/1594835.1504181
15. Larsen, P., Ladelsky, R., Lidman, J., McKee, S.A., Karlsson, S., Zaks, A.: Parallelizing more loops with compiler guided refactoring. In: Proceedings of the 2012 41st International Conference on Parallel Processing, ICPP '12, pp. 410–419. IEEE Computer Society, Washington, DC, USA (2012). doi:10.1109/ICPP.2012.48
16. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: an optimizing, portable OpenMP compiler. Concurr. Comput. Pract. Exp. **19**(18), 2317–2332 (2007)
17. Maleki, S., Gao, Y., Garzaran, M., Wong, T., Padua, D.: An evaluation of vectorizing compilers. In: Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on, pp. 372–382 (2011). doi:10.1109/PACT.2011.68
18. McKinley, K.S., Singhai, S.K., Weaver, G.E., Weems, C.C.: Compiling for Heterogeneous System: A Survey and an Approach. Technical Report. Amherst, MA, USA (1995)
19. Monsifrot, A., Bodin, F., Quiniou, R.: A machine learning approach to automatic production of compiler heuristics. In: Artificial Intelligence: Methodology, Systems, Applications, pp. 41–50. Springer, Berlin (2002)
20. Mustafa, D., Auranzeb, Eigenmann, R.: Performance analysis and tuning of automatically parallelized openmp applications. In: Proceedings of the International Workshop on OpenMP, IWOMP (2011)
21. Mustafa, D., Eigenmann, R.: Portable section-level tuning of compiler parallelized applications. In: SC'12: Proceedings of the 2012 ACM/IEEE Conference on Supercomputing. IEEE Press (2012). http://engineering.purdue.edu/paramnt/publications/SC12_DHEYA.pdf
22. Nobayashi, H., Eoyang, C.: A comparison study of automatically vectorizing fortran compilers. In: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, pp. 820–825 (1989)
23. OpenMP [Online]. Available: http://openmp.org/wp/
24. OpenUH Compiler Suite : User Guide (2007). http://www2.cs.uh.edu/openuh/OpenUHUserGuide.pdf
25. O'Boyle, M.F.P., Bull, J.M.: Expert programmer versus parallelizing compiler: a comparative study of two approaches for distributed shared memory. Sci. Program. **5**(1), 63–88 (1996)
26. PGI Compiler Reference Manual: Parallel Fortran, C and C++ for Scientists and Engineers (2012). http://www.pgroup.com/doc//pgiref.pdf
27. PGI Compiler user's Guide (2012). http://www.pgroup.com/doc/pgiug.pdf
28. Pan, Z., Armstrong, B., Bae, H., Eigenmann, R.: On the interaction of tiling and automatic parallelization. In: First International Workshop on OpenMP, pp. 24–35 (2005)
29. Pan, Z., Eigenmann, R.: Rating compiler optimizations for automatic performance tuning. In: SC2004: High Performance Computing, Networking and Storage Conference, (10 pages) (2004). http://engineering.purdue.edu/paramnt/publications/sc04.pdf
30. Pan, Z., Eigenmann, R.: Fast and effective orchestration of compiler optimizations for automatic performance tuning. In: The 4th Annual International Symposium on Code Generation and Optimization (CGO), pp. 319–330 (2006). http://engineering.purdue.edu/paramnt/publications/PE06_Orchestration.pdf
31. Pan, Z., Eigenmann, R.: Peak—a fast and effective performance tuning system via compiler optimization orchestration. ACM Trans. Program. Lang. Syst. **30**, 1–43 (2008)

32. Pingali, K., Nguyen, D., Kulkarni, M., Burtscher, M., Hassaan, M.A., Kaleem, R., Lee, T.H., Lenharth, A., Manevich, R., Méndez-Lojo, M., Prountzos, D., Sui, X.: The tao of parallelism in algorithms. In: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI '11, pp. 12–25. ACM, New York, NY, USA (2011). doi:10.1145/1993498.1993501
33. Pinkers, R.P.J., Knijnenburg, P.M.W., Haneda, M., Wijshoff, H.A.G.: Statistical selection of compiler options. In: Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings. The IEEE Computer Society's 12th Annual International Symposium on, pp. 494–501 (2004). doi:10.1109/MASCOT.2004.1348305
34. ROSE User Manual: A Tool for Building Source-to-Source Translators (2012). http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf
35. Satoh, S.: NAS Parallel Benchmarks 2.3 OpenMP C version [Online]. Available: http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp (2000)
36. Schulte, W., Tillmann, N.: Automatic parallelization of programming languages: past, present and future. In: Proceedings of the 3rd International Workshop on Multicore Software Engineering, IWMSE '10, pp. 1–1. ACM, New York, NY, USA (2010). doi:10.1145/1808954.1808956
37. Shen, Z., Li, Z., Yew, P.: An empirical study of fortran programs for parallelizing compilers. IEEE Trans. Parallel Distrib. Syst. **1**, 356–364 (1990)
38. Tiwari, A., Chen, C., Jacqueline, C., Hall, M., Hollingsworth, J.K.: A scalable auto-tuning framework for compiler optimization. In: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing, pp. 1–12. IEEE Computer Society, Washington, DC, USA (2009). doi:10.1109/IPDPS.2009.5161054. http://portal.acm.org/citation.cfm?id=1586640.1587552
39. Tournavitis, G., Wang, Z., Franke, B., O'Boyle, M.F.: Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping. SIGPLAN Not. **44**(6), 177–187 (2009). doi:10.1145/1543135.1542496
40. Vandierendonck, H., Rul, S., Bosschere, K.D.: The paralax infrastructure: automatic parallelization with a helping hand. In: International Conference on Parallel Architectures and Compilation Techniques, pp. 389–400 (2010)
41. William, B., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T., Lee, J., Padua, D., Paek, Y., Pottenger, B., Rauchwerger, L., Tu, P.: Parallel programming with Polaris. Computer. **29**(12), 78–82 (1996)
42. Yoo, S., Lee, H., Killian, C., Kulkarni, M.: Incontext: simple parallelism for distributed applications. In: Proceedings of the 20th International Symposium on High performance distributed computing, HPDC '11, pp. 97–108. ACM, New York, NY, USA (2011). doi:10.1145/1996130.1996144