electronics

MDPI

Article

Automatic and Interactive Program Parallelization Using the Cetus Source to Source Compiler Infrastructure v2.0

Akshay Bhosale *, Parinaz Barakhshan, Miguel Romero Rosas and Rudolf Eigenmann

Department of Electrical & Computer Engineering, University of Delaware, Newark, DE 19711, USA; parinazb@udel.edu (P.B.); miguelro@udel.edu (M.R.R.); eigenman@udel.edu (R.E.)
* Correspondence: akshay@udel.edu

Abstract: This paper presents an overview and evaluation of the existing and newly added analysis and transformation techniques in the Cetus source-to-source compiler infrastructure. Cetus is used for research on compiler optimizations for multi-cores with an emphasis on automatic parallelization. The compiler has gone through several iterations of benchmark studies and implementations of those techniques that could improve the parallel performance of these programs. This work seeks to measure the impact of the existing Cetus techniques on the newer versions of some of these benchmarks. In addition, we describe and evaluate the recent advances made in Cetus, which are the capability of analyzing subscripted subscripts and a feature for interactive parallelization. Cetus started as a class project in the 1990s and grew with support from Purdue University and from the National Science Foundation (NSF), as well as through countless volunteer projects by enthusiastic students. While many Version-1 releases were distributed via the Purdue download site, Version 2 is being readied for release from the University of Delaware.

Keywords: automatic parallelization; subscripted subscript analysis; interactive parallelization

Citation: Bhosale, A.; Barakhshan, P.; Rosas, M.R.; Eigenmann, R. Automatic and Interactive Program Parallelization Using the Cetus Source to Source Compiler Infrastructure v2.0. *Electronics* **2022**, *11*, 809. https://doi.org/10.3390/electronics11050809

Academic Editors: Manuel E. Acacio and George Angelos Papadopoulos

Received: 29 December 2021
Accepted: 1 March 2022
Published: 4 March 2022

1. Introduction

Cetus [1] (available online: https://sites.udel.edu/cetus-cid/, accessed on 21 February 2022) is a source-to-source translator for programs written in the C language. The primary use is as a parallelizing compiler, translating C programs to equivalent C code annotated with OpenMP parallel directives. Cetus is a research platform to study parallelization techniques and related program transformations. As such, the design has been kept lean and easy to learn. Cetus has never been extended to handle other languages, for that reason. Cetus was created out of a need for a state-of-the-art automatic parallelizer for multi-cores, written in a modern language and capable of performing analyses and transformations for today's architectures. The effect of the various optimization techniques in Cetus on the performance of prior generations of benchmarks and machines is well documented [2,3]. One of the contributions of the present paper is to determine which of these effects have remained invariant and where do they differ with the evolution of the benchmarks and architectures. In addition, we present and evaluate two new Cetus capabilities, namely subscripted subscript analysis and interactive compilation. The goal of this paper is to extensively evaluate the current Cetus techniques on current versions of the benchmarks and summarize the recent advancements in Cetus, some of which were discussed in detail in our previous contribution [4].

Subscripted subscript analysis:

A long standing challenge for automatic parallelizers is the compile-time parallelization of subscripted subscript patterns. A significant number of loops in a class of irregular applications that we analyzed could not be automatically parallelized, as they exhibited subscripted subscript patterns. If an algorithm can prove the presence of a property, such

as monotonicity, for the subscript array, the enclosing loops can be parallelized. The information required to prove the existence of these properties is often present in the application code itself, which makes it feasible to automatically detect the properties. Our work was motivated by two observations: (i) in a class of programs, such as adaptive mesh refinement applications and sparse matrix computations, subscripted subscript patterns are the main impediment to more successful automatic parallelization. (ii) Currently, there are no known compile-time analysis techniques that can automatically parallelize such loop patterns. In this paper, we briefly describe a compile-time algorithm, which makes use of *symbolic range aggregation* to prove the monotonicity of the subscript array and parallelize important loops in application codes. We then demonstrate the effectiveness of the algorithm in improving the performance of benchmark applications. The algorithm is described in detail in [4].

Interactive compilation using iCetus:

While the ability to deal with subscripted subscript patterns significantly advances the state of the art of optimizing compilers, more is needed. Typical parallelizers are able to substantially improve about one in two science/engineering applications. This is a success from a scientific viewpoint, but is still unsatisfactory for the end user. It is especially aggravating for the engineer of novel applications, which may not exhibit the regular data structures that the parallelization technology learned to optimize well. What is more, even where the tools succeed in detecting parallelism, mapping this parallelism to a given architecture may introduce overheads that offset the gain of automatic optimization. The result is that users see large performance variations across programs and architectures, ranging from nearly ideal speedup to slowdown of the original program.

Our aim for the iCetus tool (available at icetus.ece.udel.edu/cetusWeb/, accessed on 1 November 2021) is to involve the user in the decisions that compilers struggle with. User feedback is factored into program parallelization. To that end, iCetus provides the user with information about how the compiler analyzes, transforms, and parallelizes the program, as well as displaying the speedup gained from applying such optimization to the code. It offers a user interface for controlling program parallelization, based on this information. Doing so combines user knowledge and classical compiler capabilities.

This paper briefly describes the underlying system design of iCetus, the functionalities supported by the first version of the tool, and features that will be added in upcoming versions. We present the results of an iCetus user survey, assessing the usefulness and importance of the features in the current tool version as well as anticipated features of the next version.

In summary, this paper makes the following contributions:

1. We present a brief overview of the analysis and transformation techniques in Cetus and measure the impact of individual techniques on the overall performance of 7 programs from the latest NAS Parallel benchmark suite [5] v3.3 and 30 programs from the PolyBench benchmark suite [6] v4.2. Cetus achieved a maximum performance improvement to the order of 1.22–20 times over the serial execution for the benchmarks evaluated from the NAS Parallel benchmark suite and about 28–108 times improvement for the benchmarks evaluated from the PolyBench suite.
2. We briefly describe a compile-time algorithm for detecting monotonic subscript arrays, which is adequate for automatically parallelizing a class of programs that demonstrate subscripted subscript patterns. We present the performance results after applying the algorithm by hand to key loops from two real scientific applications, and discuss the overall impact on the performance of the applications.
3. We present a new interactive parallelization tool called iCetus (interactive Cetus), which supports the user in determining performance bottlenecks in application codes and helps to resolve them in an interactive, menu-driven way. In addition, we present the results of a user survey, quantifying the importance and usefulness of the implemented and proposed features of iCetus for optimizing scientific codes.

The remainder of the paper is organized as follows. Section 2 provides an overview of the analysis and transformation passes in Cetus. Section 3 briefly describes the new compile-time algorithm to automatically parallelize subscripted subscript patterns. Section 4 describes the interactive Cetus tool. Section 5 evaluates the presented techniques and capabilities. Section 6 presents related work, followed by conclusions in Section 7.

2. Overview of the Analysis and Transformation Passes in Cetus

The Cetus translator is implemented in Java. The internal program representation (IR) uses a Java class hierarchy and is syntax oriented, allowing the Cetus output to resemble the source code closely. Internally, the translator is organized into three distinct parts—the front end (lexer and parser), the IR, and a set of analysis and transformation passes that enable automatic parallelization. The "base" represents the IR in the form of the said class hierarchy, which provides and implements the functions needed by pass writers for creating, analyzing, modifying, and printing programs. The passes build on the base, providing program analysis and transformation functionality. The Cetus driver invokes the passes and creates the user interface, such as the Cetus command line and its options. The base contains 12,000 lines of Java, while the passes for the parallelization functionality contain some 32,000 lines, with more passes being added. A driver is typically a small function of a few 10 to 100 lines. The most common driver is that for the parallelization functionality. Many other drivers have been added by researchers, using the Cetus platform for creating tools for program instrumentation, OpenMP-to-CUDA translation [7], OpenMP-to-MPI conversion [8], and more.

We categorize the Cetus passes into program analyses, parallelism-enabling transformations, and architecture-mapping transformations. This is not strict. Some analysis passes also apply certain transformations to bring the code into a form that is easier to understand, and many transformation passes perform some analyses as well. However, all program analyses gather information about the program that will enable the later transformation techniques. Parallelism-enabling transformations remove data dependencies, bringing loops into a form that can be parallelized. Data privatization and reduction parallelization are two of the most important parallelism-enabling transformations. Cetus creates only fully parallel loops. That is, there are currently no techniques that exploit partial parallelism in the presence of data dependencies that cannot be removed. Cetus also contains a range of techniques that map the detected parallel loops onto specific architectures. The first two categories are generic, enabling the exploitation of parallelism on all architectures. They are explained briefly below, and their impact is measured in Section 5. Among the architecture-mapping techniques are loop interchange, tiling, the conversion to CUDA, and the translation to the MPI form. We briefly describe and evaluate the impact of the loop interchange pass in Cetus. This paper does not further elaborate on the rest of the architecture-mapping transformations. A detailed description of the various aforementioned analysis and transformation passes in Cetus is mentioned in [2,9].

2.1. Program Analysis

- Cetus includes classical data-dependence and pointer/alias analyses. For data-dependence detection, it makes use of the range test [10] and Banerjee–Wolfe [11] test. Pointer/alias analysis uses an inter-procedural points-to-analysis technique [12] with some variations from the original algorithm.
- A key distinguishing technique of the Cetus platform is its symbolic range analysis capability, which enables other passes to manipulate and reason about symbolic expressions [13]. Several transformation passes make use of this capability. It also represents an important basis for the array property analysis described in Section 3.
- Most Cetus techniques work intraprocedurally, that is, they neither gather information from other subroutines, nor apply transformations across subroutine boundaries. The compiler relies on subroutine inlining to overcome this limitation. Inlining enables all

other techniques to see and operate across subroutine boundaries. Section 5 shows where subroutine inlining makes a difference.

- Cetus performs several normalization steps that allow its analyses to make simplifying assumptions. Examples are loop normalization, which converts all loops to a form that iterates from 0 to an upper bound in steps of 1, and statement normalization, ensuring that all statements contain only one assignment. We will not further elaborate on these techniques.

2.2. Parallelism-Enabling and Architecture-Mapping Transformations

Cetus includes four techniques in this category, which were previously evaluated to be the most important [2]: data privatization, reduction parallelization, induction variable substitution and loop interchange.

- Data privatization identifies scalar and array variables whose values are produced *and* consumed within the same loop iteration. Such variables can be declared *loop private* , using OpenMP's private clause [14]. Doing so eliminates *anti-dependencies* that would otherwise arise [15].
- Reduction parallelization and induction variable substitution deal with mathematical reduction operations and with induction sequences, respectively. The original form of both patterns contain data dependencies that would inhibit parallelization. Cetus marks identified reduction variables, using OpenMP directive clauses (in certain OpenMP-unsupported cases, Cetus rewrites the reduction directly in a parallel form). Induction sequences are rewritten as closed-form expressions.
- The loop interchange pass in Cetus is capable of determining the best permutation of loops in a loop nest for effectively exploiting data locality and parallelism in concert. In addition to the legality test for interchange, the pass implements a memory model, described in [16], to determine cache line reuse from multiple accesses to the same memory location and from consecutive memory accesses.

3. Subscripted Subscript Analysis

This section briefly describes a compile-time algorithm capable of automatically parallelizing loops with subscripted subscript patterns. We also present a brief discussion of the subscript array properties required to prove independence in loops containing subscript arrays and to automatically parallelize such loops.

3.1. Motivation

Recall from Section 1 that subscripted subscript patterns are the main impediment preventing Cetus from matching hand-parallelized performance. Similar experiments with other optimizers, such as Rose [17] and Intel's ICC compiler [18], yielded the same result. In order to parallelize subscripted subscript patterns, a compiler must be able to determine possible values of the subscript array and formulate subscript array properties. A key observation was that the information required to do so was often present in the application code itself, specifically in loops that modify the content of the subscript array. Figures 1 and 2 illustrate this situation.

```
1: for (i = 0; i < n; i++) {
2:     for(j = pntr[i]; j < pntr[i+1]; j++){
3:         x[j] = c[j] * diag[d[j]];
4:     }
5: }
```

Figure 1. Subscripted subscript pattern in an example loop. The subscript expression of array x, that is, j, derives its values from array *pntr*.

In the example code shown in Figure 1, values of array *pntr* appear at the subscript of array *x* on line 3. The loop on line 1 can be parallelized if array *pntr* is monotonically increasing. The sequence of loops shown in Figure 2 appears before the loop in Figure 1 and produces monotonic values for array *pntr*. In general scientific applications, complex loop patterns are used to define and modify the subscript array. However, in our analysis, we found that ***in many such programs, the necessary and sufficient information showing that the involved loops can, in fact, be parallelized was present in the program code and was not dependent on the program input data.*** While investigating this information can be complex, the opportunity *exists* to develop compile-time analyses that perform such detection.

```
1: for(i = 0; i < n; i++){
2:       pntr[i] = 0;
3: }
```

(a)

```
1: for(i = 0; i < n; i++){
2:     if(condition) pntr[i]++;
3: }
```

(b)

```
1: for(i = 1; i < n; i++){
2:     pntr[i] = pntr[i - 1] + pntr[i];
3: }
```

(c)

Figure 2. Subscript array *pntr* derives monotonic values in three steps: loop in (**a**) initializes *pntr* to zero; loop in (**b**) conditionally increments *pntr*; (**c**) sum recurrence.

3.2. Analyzing Subscript Array Properties

We analyzed the application codes listed in Table 1 for subscripted subscript patterns. We looked at all codes from these suites, except in SPEC CPU 2006, where we considered the seven most promising of the 17 applications. Subscripted subscript patterns were found in 14 of the 32 inspected application codes. In all of these patterns, the loops were parallel, comprising single or multiple array write references with subscript expressions that contain the value of another array, but there was no read reference of the written array. Such loops can be parallelized if the compiler can prove that there is no *self output dependence* [19]. In doing so, the array properties described below are of interest:

1. **Injectivity:** An array is said to be injective if $a[i] \neq a[j], \forall i \neq j$. The array accesses $x[a[i]]$ and $x[a[j]]$ are independent if $i \neq j$ in this case.
2. **Monotonically increasing or decreasing:** An array is monotonically increasing if $a[i] \leq a[j], \forall i < j$ and monotonically decreasing if $a[i] \geq a[j], \forall i < j$. This implies non-strict monotonicity.
3. **Strictly monotonically increasing or decreasing:** An array is strictly monotonically increasing if $a[i] < a[j], \forall i < j$ and strictly monotonically decreasing if $a[i] > a[j], \forall i < j$. Strict monotonicity implies injectivity.

Table 1. Benchmarks codes analyzed for the presence of subscripted subscripts.

Benchmark Suite	Total Benchmarks Analyzed	Benchmarks with Subscripted Subscripts
NPB3.3 [5]	10	3
SuiteSparse 5.4.0 [20]	10	7
SPEC CPU 2006 [21]	7	2
The Mantevo Project [22]	5	2

A representative example of the creation of these array properties is shown in Figure 2. Common intermediate properties that are captured by our algorithm are *positive* or *non-negative*, as shown in Figure 2b.

Our previous work [23] mentions key loops from benchmark applications that can be parallelized due to each of the above-mentioned properties. Recall again that the above properties were found to be present in the program code itself and independent of the program input data. An advanced programmer would be able to determine the properties and thus parallelize the enclosing loops.

3.3. Compile-Time Algorithm for Subscript Array Analysis

Our algorithm proceeds in program order, analyzing loop nests and determining the properties described in the previous section. Loops in each nest are analyzed from inside out. At each loop level, the algorithm analyzes the values of the loop variant variables: integer scalars and integer arrays with simple subscripts. For the current algorithm, "simple subscript" refers to a subscript expression of the form $i + k$, where i is the iteration number and k is a constant. Loops with a single subscripted-subscript array write reference can be analyzed by the algorithm. We assume that all eligible loops are normalized, with at most one assignment per statement. In addition, we assume that induction variables are replaced with the appropriate closed-form expressions. Normalized loops are characterized by iteration spaces that begin at 0 and are stride 1. The loop variable represents the iteration number.

The algorithm follows two key observations:

1. Recurrence relationships generate monotonic arrays by assigning to a current array element the summation of the immediately preceding element and a positive value for strict monotonicity or a non-negative value for non-strict monotonicity.
2. Positive values are often created by starting with 0 and conditionally incrementing it an arbitrary number of times.

Our algorithm proceeds as follows. Phase 1 performs symbolic analysis of the loop body and captures the effect of one iteration on the value of the variables of interest. Phase 2 then aggregates this expression across all iterations, determining the effect of the entire loop on the variable, and testing for array properties. After Phase 2, the loop is substituted by the set of aggregate expressions, which represent the effect of the loop. The algorithm then proceeds with the next outer loop. We described the algorithm in detail in our most recent contribution [4].

4. Interactive Cetus (iCetus)

The iCetus tool (available at icetus.ece.udel.edu/cetusWeb/, accessed on 1 November 2021) is a new interactive web interface to Cetus, providing users with a range of capabilities for the source-to-source transformation of C programs using OpenMP directives on shared memory machines.

An early version of iCetus is implemented as a web application for easy access, eliminating the need for user installation and updates. The tool supports the user through all phases of the program transformation process, including program analysis and optimization. The program analysis phase includes static and dynamic analyses, pointing out loops that represent performance bottlenecks and should be improved. The optimization phase offers diverse options to cater to different levels of user skills. While the tool can parallelize code fully automatically for non-experts, power users can steer the parallelization process in a menu-driven way. By interactively displaying compiler analysis results, iCetus supports the user in pinpointing parallelization impediments and resolving them. The programmer can apply successive improvements by editing the input program and the parallel version of the code, evaluating the performance, and comparing it to that obtained by previous program versions. In addition, iCetus can be used as a learning tool to understand the usage of parallel constructs and to write higher-quality code.

4.1. Automatic Parallelization Challenges and Opportunities for Interactive Tools

Automatic parallelizers face challenges that interactive parallelizers can address. Our discussion below covers the major challenges of automatic parallelizers and the potential of interactive parallelizers.

4.1.1. Correctness and Conservative Assumptions

Parallelization techniques are highly complex, and user code may obscure parallelism. Furthermore, we expect that compilers perform their optimizations correctly on *all* programs. The strict demand for correctness makes parallelizers conservative, bypassing many opportunities for optimization.

For example, two key compiler capabilities for identifying parallelism are data dependence and private variable analysis. If a compiler cannot prove that data accesses are dependence free or variables are private, it conservatively assumes that they are not. This is similarly true for other techniques, such as alias analysis, reduction parallelization, and induction variable recognition.

The opportunity for an interactive tool is to present the results of these analyses and then let the user decide what is acceptable. In this way, a data dependence that the compiler cannot disprove or a variable that the compiler cannot privatize can be tagged as such by the user. This is especially useful in the fairly common case of a loop, where only a few hard-to-detect data dependence or private variable patterns remain that can be recognized by the user. Cetus' optimization report will be of help in this situation. By selectively showing the remaining dependencies of a loop and allowing the user to drill down into the analysis details, an interactive tool can help parallelize key loop patterns that batch-oriented compilers are unable to.

4.1.2. Overheads and Profitability

Program transformations may introduce overhead. Estimating this overhead is highly complex and depends on the characteristics of both the program and the target architecture. Performance models usually include parameters that are only known once the program executes, making it often infeasible for the compiler to decide whether or not an applicable technique is beneficial. The dilemma is that not applying the technique forgoes the optimization opportunity; applying it may introduce overhead that offsets the gain or, worse, degrades performance. For instance, a major reason an automatically parallelized loop may execute more slowly than the sequential version is that the loop is too small, so the cost of invoking and terminating the parallel activity dominates. Transformations that add substantial code to the program, such as reduction parallelization and loop tiling, are especially prone to low profitability.

The opportunity for an interactive tool lies in informing the user about loops where profitability is borderline or needs run-time information. The tool can also disclose high-overhead transformations that are applied, allowing the user to be the judge on profitability.

Another tool opportunity is to offer run-time measurements gained through program execution. The values of critical variables may be evaluated (e.g., the number of iterations of a loop), the execution time of a loop may be measured, or the performance of a serial and parallel code version may be compared. An advanced scenario would be to "auto-tune" a code section or the entire program. That is, the interactive tool would execute applicable optimization variants and determine the best.

4.2. iCetus Features

Besides basic tool functions, such as inputting and uploading programs or program sections and browsing the serial or parallel code, iCetus offers the following key features:

- The results of compiler analyses and transformations, such as the values of variables, data dependencies, privatized and reduction variables, can be inspected to understand the parallelization process and identify potential problems.
- A menu-driven interface for customizing parallelization options is provided along with help functions.
- User optimizations can be applied by modifying the input code as well as the parallel code.
- Information about speedup and efficiency gained by the optimization is provided.
- A variety of examples are given to illustrate key concepts in parallel programming, transformations, and the tool's capabilities, along with the possibility of making changes to those examples and experimenting with what-if scenarios.

4.3. iCetus Implementation Overview

The iCetus tool is implemented as a dynamic web application generating the pages/-data in real time, as per the user's request. The web server, upon receiving a request for a dynamic page, passes the page to the application server, which processes the contained code. Our current design connects the application server to a database that stores user inquiries. Evaluations of the project are based on this information. Database queries create record sets that are returned to the application server to complete the page. The final result is in pure HTML format, which the application server passes back to the web server. The page is then sent to the requesting browser. Figure 3 illustrates this process.

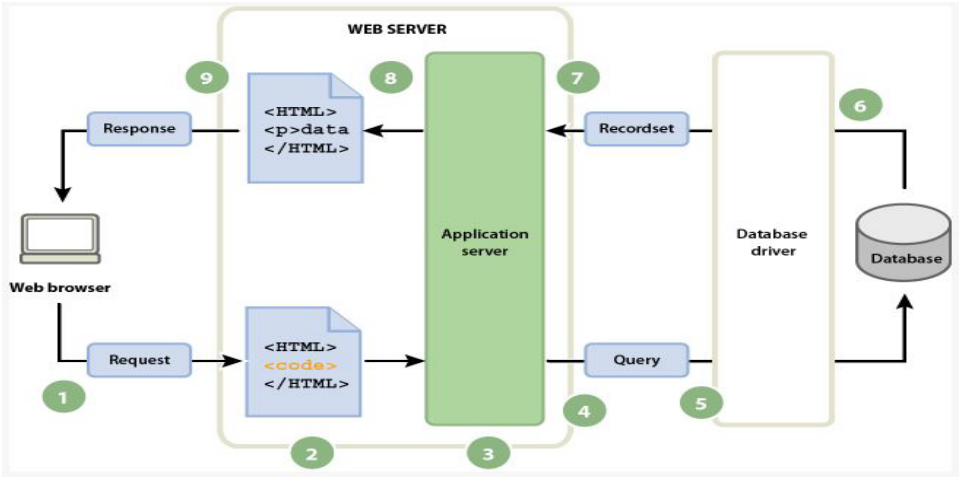


Figure 3. Processing dynamic web pages.

1. Web browser requests a dynamic page.
2. Web server locates the page and passes it to the application server.
3. Application server scans the page for instructions.
4. Application server sends the query to the database driver.
5. Driver executes the query against the database.
6. Recordset is returned to the driver.
7. Driver passes the recordset to the application server.
8. Application server inserts data in page and then passes the page to the web server.
9. Web server sends the finished page to the requesting browser.

5. Evaluation

This section evaluates the described Cetus techniques and capabilities. Section 5.1 measures the impact of each of the techniques of Section 2 on the performance of the benchmark applications. The performance of the subscripted subscript analysis algorithm of Section 3 is measured by hand application of the algorithm to key loops from two real scientific applications in Section 5.2. Section 5.3 quantifies the importance of the iCetus tool of Section 4 via a user survey.

5.1. Performance Impact of Individual Cetus Techniques

We discuss the impact of individual optimization techniques on the overall application performance. We performed our experiments on a set of 7 benchmarks from the NAS Parallel benchmark suite (NPB) v3.3 [5] and all of the 30 benchmarks from the PolyBench benchmark suite v4.2 [6]. For each benchmark, we report the reduction in performance of the Cetus parallel code over the serial execution, after disabling a technique, as a measure of the technique's impact.

5.1.1. Experimental Setup

The NAS Parallel Benchmarks were derived from computational fluid dynamics (CFD) codes [24]. They were designed to compare the performance of parallel computers and are widely recognized as a standard indicator of computer performance. The benchmark suite consists of five kernels (IS, EP, MG, FT, and CG); three pseudo applications (BT, SP, and LU); and two benchmarks for unstructured computation (UA and DC). We used serial versions of the benchmarks written in C [25] for our experiments and measured the performance of the applications for input Class B, having a problem size that is neither too small nor too big. We evaluated the codes EP, MG, CG, BT, SP, LU, and DC, which present opportunities for the Cetus techniques.

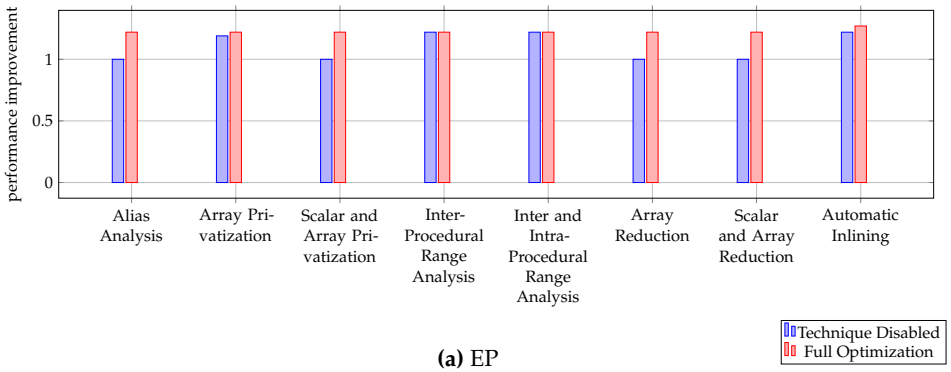
PolyBench v4.2 [6,26] is a benchmark suite of 30 numerical computations extracted from operations in various application domains (19 linear algebra computations, 3 image-processing applications, 6 physics simulations, and 2 data-mining applications). We were able to compile and run the Cetus parallel code for all of the benchmarks from this suite. We used slightly modified versions of the source code for some of these benchmarks in order to accommodate language features supported by Cetus [26].

The execution times for each of the benchmarks were recorded on a compute node with a 20 core Intel Xeon Gold 6230 processors in a dual socket configuration, with a processor base frequency of 2.1 GHz, 27.5 MB cache and we used up to 1 GB of DDR4 memory. We compiled the application codes using GCC v4.8.5 with the -O3 optimization flag enabled on CentOS v7.4.1708. We report the median of three application runs. We used one thread per core.

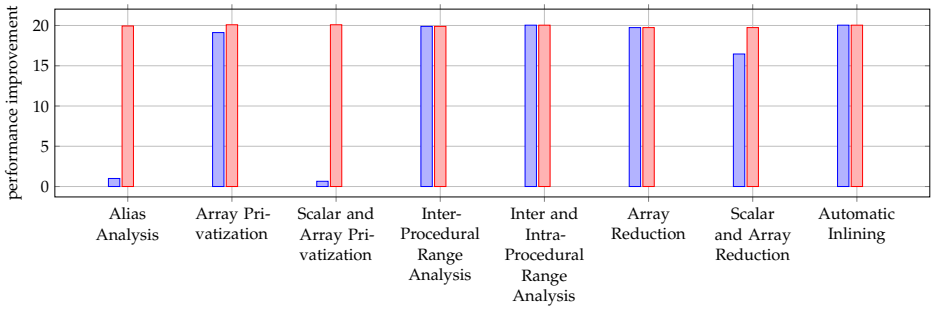
5.1.2. Results

We discuss the impact of disabling individual optimization techniques on the overall performance of the applications. We also study the effect of interactions among optimization techniques on performance. The performance of the application code, compiled using Cetus under full optimization (all techniques turned ON) serves as the baseline; then we turn off one technique at a time and measure the impact on performance. Figures 4–6 show the performance results for each evaluated benchmark from the NAS Parallel Benchmarks and PolyBench Benchmark suites. We report the performance results for 2 mm, 3 mm, Doitgen and Gramschmidt benchmarks from the PolyBench suite for the various techniques. These results are representative of the results obtained for other benchmarks from this suite. We make the following observations:

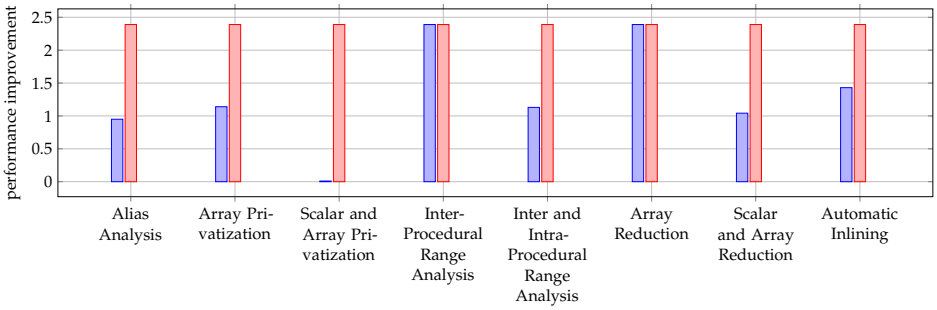
1. Scalar and array privatization is the most important technique, affecting the performance of five out of the seven benchmark codes tested from the NPB suite. For CG, SP and MG, the parallel performance drops below the performance of the serial code if privatization is disabled, as it leads to parallelization of the inner loops in computationally intensive loop nests. Disabling privatization affects the performance of every benchmark from the PolyBench suite, but never drops below that of the serial code.
2. Reduction parallelization affects the performance of four applications in the NPB suite, whereas the induction variable substitution has little to no impact. We attribute the latter to the fact that the NAS and PolyBench benchmarks were already prepared for parallel execution, with most induction variables being substituted [2].
3. Disabling range analysis (both inter and intra-procedural analysis) substantially deteriorates the performance of MG and has a slight impact on the performance of SP and BT from the NPB suite. In the PolyBench suite, disabling range analysis reduces the performance of the Cetus parallel code in Doitgen as shown in Figure 6. The reason for this result is that symbolic range analysis leads to the privatization of arrays, which in turn parallelized the computationally intensive loops in these benchmarks.
4. Alias analysis affects almost all of the benchmarks. Cetus assumes conservative all-to-all aliases in the presence of complex pointer declarations for arguments within function calls. We found this to be the case in benchmarks, such as CG and DC, from the NAS suite and almost every benchmark from the PolyBench suite. We set the option to assume that no aliases exist, which is correct for these benchmarks.
5. Disabling inlining leads to a reduction in the performance of EP by about 5% as shown in Figure 4a and about 42% for SP from 2.45–1.42 times as shown in Figure 4c. In our experiments, performance of automatic inlining was comparable to the performance of selective inlining (inlining inside selected functions). We did not evaluate the performance of LU and MG, as automatic inlining led to excessive code growth.
6. The locality enhancement technique, loop interchange, does not show any impact on the performance of the NAS benchmarks, whereas in the PolyBench suite, disabling the loop interchange leads to a reduction in performance of codes 2 mm and 3 mm, as shown in Figure 6a,b, respectively. For these benchmarks, the loop interchange could determine the best order of loops in the computationally intensive loop nests for improving locality and cache reuse.



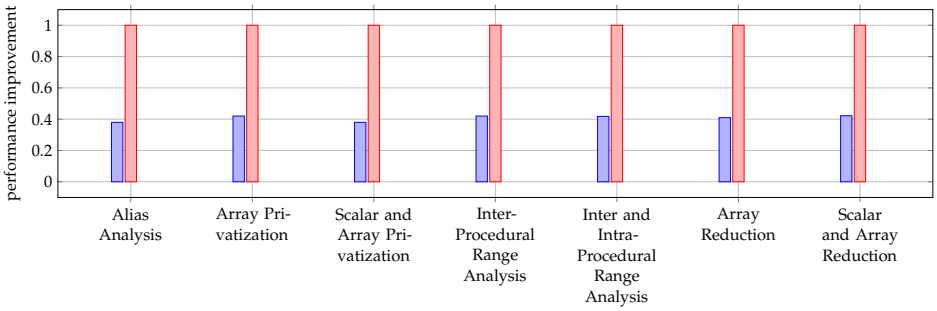
(a) EP



(b) CG



(c) SP



(d) DC

Figure 4. (**a**) Performance results for EP, (**b**) Performance results for CG, (**c**) Performance results for SP and (**d**) Performance results for DC benchmarks from the NAS Parallel benchmark suite v3.3 for input class B. The selected base case is the best tuned version (all on). One technique at a time is turned off, and the impact on performance is plotted (Lower bar means a higher impact of the technique).

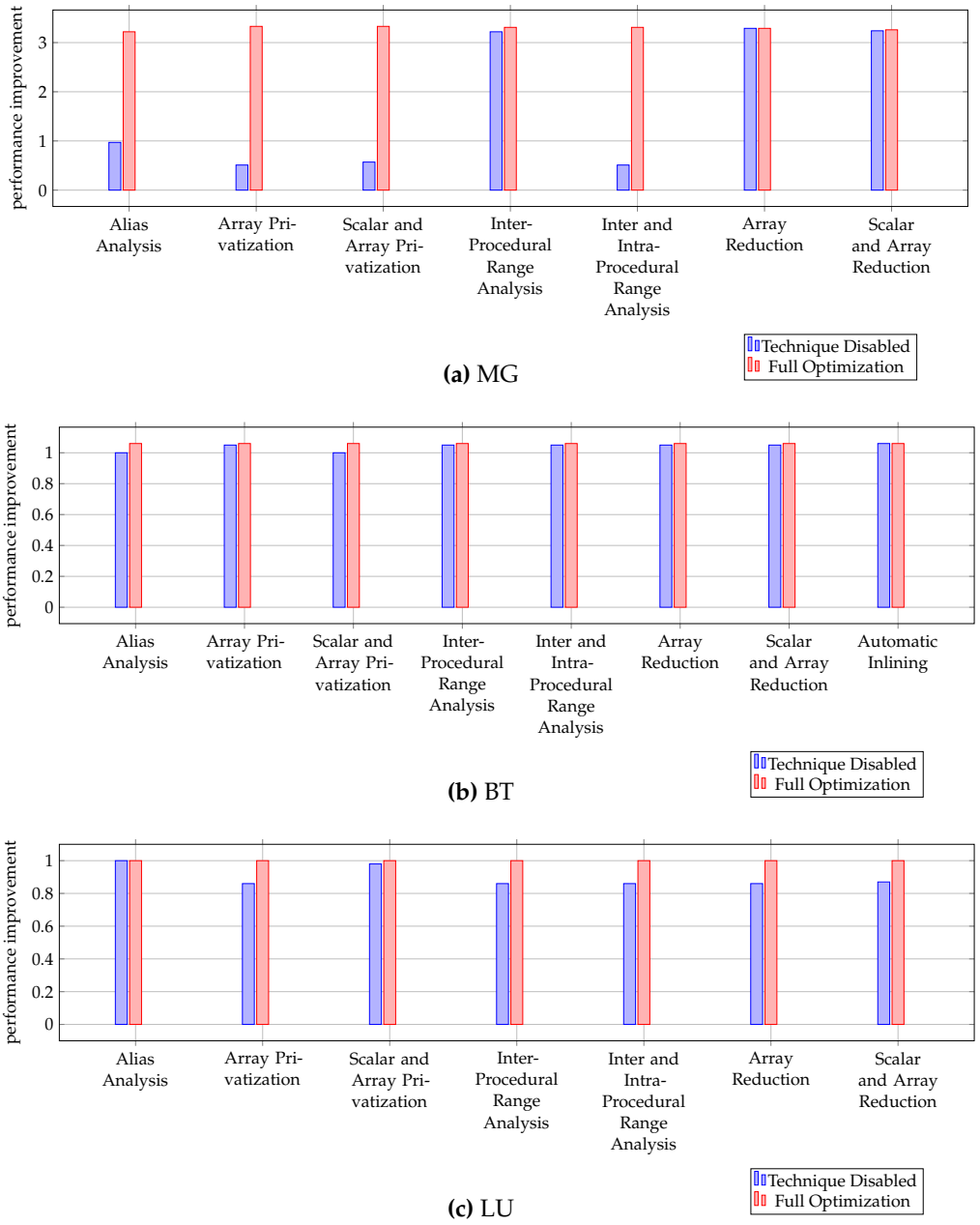


Figure 5. (**a**) Performance results MG, (**b**) Performance results for BT and (**c**) Performance results for LU benchmarks from the NAS Parallel benchmark suite v3.3 for input class B. The selected base case is the best tuned version (all on). One technique at a time is turned off, and the impact on performance is plotted (Lower bar means a higher impact of the technique).

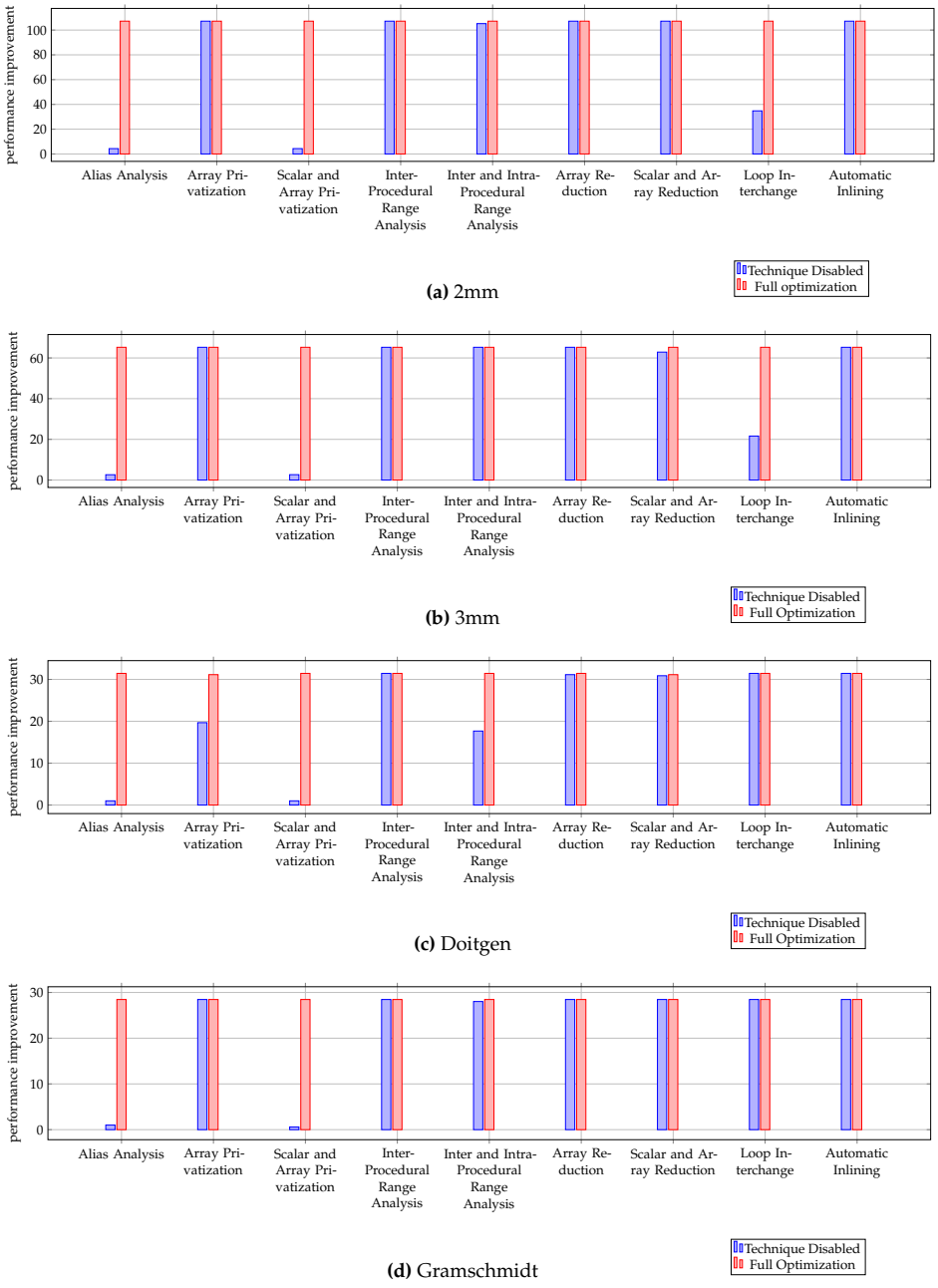


Figure 6. (**a**) Performance results for 2mm, (**b**) Performance results for 3mm, (**c**) Performance results for Doitgen and (**d**) Performance results for Gramschmidt benchmarks from the PolyBench benchmark suite v4.2. The selected base case is the best tuned version (all on). One technique at a time is turned off and the impact on performance is plotted (lower bar means a higher impact of the technique).

5.2. Evaluation of Subscripted Subscript Analysis

5.2.1. Experimental Setup

We applied the array analysis techniques presented in Section 3 to the numerical supernodal sparse Cholesky factorization and the symmetric sparse matrix scaling codes from the CHOLMOD package of the latest SuiteSparse benchmark suite v5.4.0 [20] by hand. The loop shown in Figure 7 is one of two loops exhibiting subscripted subscript patterns in the Cholesky factorization code. The code also spends time calling BLAS [27] and LAPACK [28] routines. The loop shown in Figure 8 is integral to the computational part of the sparse matrix scaling code. This application spends significant time performing I/O operations. We report the performance of the actual computation.

```
#pragma omp parallel for private (p, pf, i, k, q, fjk)
for(k=k1; k<k2; k++)
{
  for(pf=Fp[k]; pf<Fp[k+1]; pf++)
  {
     fjk[0] = Fx[2*pf];
     fjk[1] = Fx[2*pf+1];
     for(p=Ap[Fi[pf]]; p<Ap[Fi[pf]+1]; p++)
     {
        i = Ai[p];
        if(i>=k && Map[i]>=0)
        {
           q = (Map[i]+psx+(k-k1)*nsrow);
           Lx[2*q] += Ax[2*p]*fjk[0]-Ax[2*p+1]*fjk[1];
           Lx[2*q+1] += Ax[2*p+1]*fjk[0]-Ax[2*p]*fjk[1];
        }
     }
  }
}
```

Figure 7. Loop to parallelize in the subroutine *cholmod_super_numeric* from the supernodal module in CHOLMOD from the SuiteSparse benchmark suite [20]. The outermost *k*-loop can be parallelized if array *Map* has values in the range $[0 : nsrow - 1]$.

We used 10 non-symmetric sparse matrices from the University of Florida Sparse Matrix collection [29] as inputs for our experiments. The number of non-zero elements in these matrices ranges between 7.3×10^{-6}% and 1.4%. Table 2 shows the breakdown of the serial execution time of the Cholesky factorization application. Matrices which satisfy the dimensional constraints described in the application code were chosen as inputs. As can be observed in Table 2, 71.75–100% of the overall application execution time is spent in the parallel subscripted subscript loop, and the BLAS and LAPACK routines. The remainder of the execution time is spent in another loop exhibiting subscripted subscript patterns, but is not yet parallelizable using our technique. We recorded the execution times on the compute node mentioned in Section 5.1.1 and we used up to 128 GB of DDR4 memory. We also used the same execution environment as mentioned in Section 5.1.1 to compile and run the application codes. We report the mean of 10 application runs.

Table 2. Serial execution time of the supernodal sparse Cholesky factorization application, showing overall time and time of the parallelizable parts.

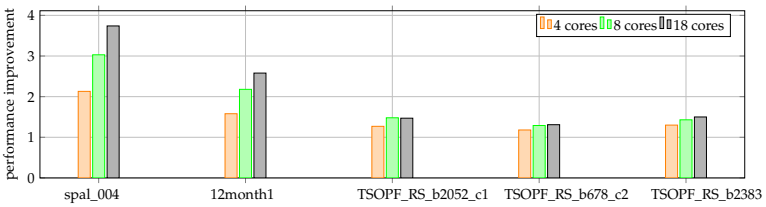
Input Matrix	Serial Execution Time of the Application	Serial Execution Time of the Subscripted Subscript Loop Parallelizable Using Aggregation (time/%)	Serial Execution Time of the BLAS and LAPACK Routines (time/%)
spal_004	20.26 s	10.1 s/49.85%	10.16 s/50.15%
12month1	28.64 s	11.22 s/39.17%	17.32 s/60.47%
TSOPF_RS_b2052_c1	96.72 s	22.2 s/22.95%	47.85 s/49.47%
TSOPF_RS_b678_c2	287.04 s	45.99 s/16.02%	159.97 s/55.73%
TSOPF_RS_b2383	372.91 s	98.83 s/26.5%	182.57 s/48.95%

```
#pragma omp parallel for private(j, k)
for(j=0; j<ncol; j++)
{
    for(k=Ap[j]; k<Ap[j+1]; k++)
    {
        Ax[k]=s[j]*Ax[k]*s[Ai[k]];
    }
}
```

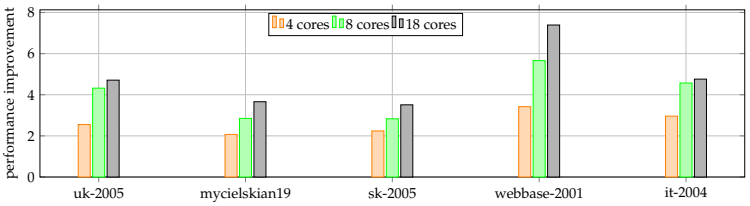
Figure 8. Example loop to parallelize. This loop performs symmetric scaling of a sparse matrix A by a scaling factor s, where s is a diagonal vector. The outer loop can be parallelized if array Ap is monotonically increasing.

5.2.2. Results

Figure 9a shows the performance results for the supernodal Cholesky factorization code and Figure 9b shows the results for the sparse matrix scaling application code. Performance improvement is defined as the execution time without versus with the key loops of Figures 7 and 8 parallel, the latter being enabled by our technique. The performance of the parallel codes on 4, 8 and 18 cores are shown in the aforementioned figures. Our technique improves the performance of the supernodal Cholesky factorization code by as much as 383% and by about 739% for the computational part of the sparse matrix scaling code.



(a) Performance improvement obtained for the Cholesky factorization application.



(b) Improvement in performance of the computational part of the Sparse Matrix Scaling code.

Figure 9. Improvement in the performance of the parallel (**a**) Cholesky factorization and (**b**) Sparse Matrix Scaling applications after applying subscripted subscript analysis.

5.3. Evaluation of iCetus

In order to evaluate the preliminary results of the iCetus project and prioritize what features should be added to the next version of the tool, we presented the tool to over 20 users. Our goal for the iCetus tool is to cater to all user classes, and hence we deliberately chose users with different skill levels. By doing so, the features that make this tool useful for beginners, as well as advanced users, were identified.

Users exhibited varying levels of familiarity with parallelization techniques (38.1% were beginners, 47.6% intermediate, and 14.3% advanced), with OpenMP (66.7% were unfamiliar, and 33.3% were knowledgeable), and with the Cetus compiler (61.9% were unfamiliar, and 38.1% had used it before).

We presented the current iCetus features and also features that we consider implementing in the next version of the tool. Users rated these features on a scale of 1 (unimportant) to 5 (very important). Section 5.3.1 quantifies the importance of the current features of the iCetus tool, while Section 5.3.2 evaluates the importance of proposed features for the next version of the tool.

5.3.1. Importance and Usefulness of Existing iCetus Features

Figure 10, presents the survey results of the importance of the existing iCetus features. The user scores for all questions are above 4, indicating the importance and usefulness of all implemented features.

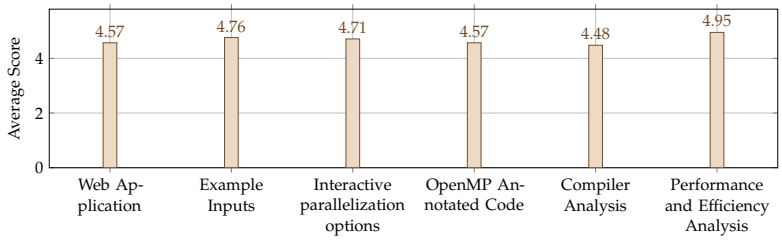


Figure 10. User feedback on existing iCetus features.

- *Web application:* This question asked about the usefulness of iCetus being available as a web application. Having the tool implemented as a web application eliminates the need for download, install, and updates. All processing is done on the server side; hence, it also benefits clients with limited computational power. The high score of 4.57 indicates strong agreement with these advantages.
- *Example inputs:* iCetus offers many example input programs that the user can choose from, illustrating key concepts of parallel programming and transformations, as well as the tool functionalities. This feature was especially important to novice users. Users scored this feature 4.76 out of 5.
- *Interactive parallelization options:* Users can choose parallelization options in a menu-driven way. This feature enables skilled users to take detailed control of the applied analyses and transformation techniques while providing reasonable defaults for beginners. This feature was deemed very important by all users, obtaining a score of 4.71 out of 5.
- *OpenMP annotated code:* Building on the Cetus source-to-source restructurer, iCetus shows the results of its transformations in the form of OpenMP-annotated source code. Users scored this feature 4.57 out of 5. They also offered the following comments to explain the relevance of this capability: OpenMP-annotated source code makes it easy to understand the transformations applied to a code. OpenMP portability provides for a good abstraction of possible underlying machines, eliminating the need for understanding many architectural details. Similarly, reasonable performance portability is appreciated. In addition, users valued the incremental parallelization process supported by this feature.
- *Compiler analysis:* This key feature enables users to understand the applied compiler passes and inspect specific categories of the program analysis results. In this way, users can query the compiler's reasoning, drilling down into the specifics of why certain program optimizations can or cannot be applied, and determining possible manual program changes to improve performance. Users scored this feature 4.48.
- *Performance and efficiency analysis:* With the highest score of 4.95, users judged the availability of the run-time information, such as performance and efficiency, as most important. This result is consistent with the fact that the lack of run-time information can be viewed as the Achilles heel of static, batch-oriented automatic parallelization.

Among the aforementioned features, *web applications* and *example inputs* were rated highly by beginner users, whereas advanced users found the *OpenMP Annotated Code* feature to be of the utmost importance. *Interactive parallelization options*, *compiler analysis* and *performance and efficiency analysis* were rated highly by all users.

5.3.2. Importance and Usefulness of Proposed iCetus Features

In preparation for the next version of the tool, we sought user feedback on the proposed features. Figure 11 reports the obtained scores.

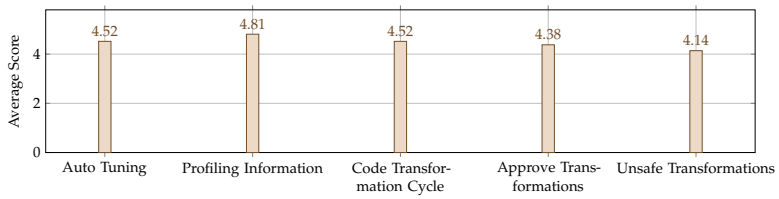


Figure 11. User feedback on proposed features for the next version of iCetus.

- *Auto-tuning:* Having an *auto-tuning* capability that determines the best combination of compiler options obtained a score of 4.52. Some users wanted the tool to find the combination that leads to the best performance but wanted some control over the techniques being tuned. Having such control is essential, as auto-tuning can be a highly time-consuming process. Another reason given was that auto-tuning could help users to learn and understand program parallelization, how it applies in different use cases, and what performance can be expected.
- *Profiling information:* Providing loop-by-loop profiling information for the serial and parallel codes and loop speedups and efficiencies are important aids in the optimization process, as evidenced by the score of 4.81. The feature helps users focus attention on relevant code sections and understand performance bottlenecks.
- *Code transformation cycle:* Being able to modify the input code and submit it for another round of compilation is essential in an interactive optimization scenario. Applying such modifications in the presence of the available analyses information goes substantially beyond the features offered by a standard program editor. The user score for this feature was 4.52.
- *Approve transformations:* Giving the user the ability to approve or reject transformations suggested by the parallelizer provides fine control over the code optimization process, especially for judging the profitability of a transformation. The score for this feature was 4.38.
- *Unsafe transformations:* A score of 4.14 indicates that users value the ability to choose from potentially applicable transformations, even if they are unsafe. Some users requested that this option be made available only to advanced programmers, as program correctness is no longer guaranteed.

For beginners, *auto-tuning* was very important. Advanced users scored highly such features as *code transformation cycle*, *approving transformations* and reporting *unsafe transformations*. Meanwhile, all users rated *profiling information* highly.

Since all proposed features scored above 4, they will all be implemented in the next version of the tool. Nevertheless, the scores are slightly lower than those of the implemented capabilities. We attribute this in part to the fact that it is easier to understand and judge the benefit of a tool's functionality when one can experiment with it. We expect the scores to increase further once the proposed features are implemented.

As part of the user interviews, one of the questions asked which additional features would be helpful to users during the optimization process. Overall, a majority of beginner developers expressed a desire for an automated optimization process that would help them improve their code performance to the greatest possible extent, along with providing helpful information about the code transformations applied during this process. In contrast, advanced developers requested features that would enable them to customize the optimization process and allow fine-grained control over the transformations applied to the code.

6. Related Work

6.1. Evaluation of Optimization Techniques in Cetus

Several contributions have compared the performance of automatic parallelizers (including Cetus) on a set of benchmark applications from the NAS and PolyBench benchmark suites. Harel et al. [30] analyzed the performances and inspected the capabilities of three

automatic parallelizers—AutoPar [31], Par4all [32] and Cetus [1]—on the NAS Parallel benchmarks. Cetus outperformed the other parallelizers in five out of the seven programs evaluated, with minimal user intervention. Similar experiments were performed by Mosseri et al. [33] on a set of five benchmarks from the PolyBench benchmark [6] suite. The Cetus parallel code for each of the benchmarks could achieve substantial speedup over the serial versions, improving the performance by as much as 754%, on average. Our work builds on the work by Bae et al. [2], who evaluated the performance of Cetus on an earlier version of the NAS Parallel benchmarks v2.4. The Cetus parallel code could improve upon the performance of the serial code, in all but three benchmarks—FT, IS and MG. However, in our experiments, the performance of the Cetus parallel code for benchmarks, such as BT and LU, is comparable to that of the serial code, whereas in MG, the Cetus parallel code achieves substantial speedup over the serial version. We attribute this discrepancy to the structural and algorithmic changes to the source codes in the newer versions of the NAS benchmarks, allowing Cetus to extract more parallelism in certain code sections, while degrading the performance in others.

Blume et al. [34] studied the effect of disabling automatic parallelization and program restructuring techniques on the performance of Perfect benchmarks. The transformations examined included recurrence replacement, induction variable substitution, scalar expansion, forward substitution, reduction recognition, loop interchange and strip-mining. Scalar expansion proved to be the most effective technique, improving the performance of 4 out of the 12 benchmarks tested, followed by reduction recognition. Bae et al. [2] also measured the contributions of individual optimization techniques to the overall performance of the NAS Parallel benchmarks. They found that scalar and array privatization, reduction parallelization, symbolic analysis and inlining had the most impact, whereas locality enhancement techniques, such as loop interchange, did not show significant effects. In our experiments, we found that these techniques are still some of the most important optimization techniques in improving the performance of present-day applications.

6.2. Subscripted Subscript Analysis

McKinley [35] described the importance of monotonicity for analyzing subscript arrays. Run-time analysis techniques for detecting monotonicity were presented by Gutierrez et al. [36], primarily in applications containing irregular reductions. Spezialetti and Gupta [37] presented compile-time techniques for detecting monotonic statements in loops. Their techniques can only detect monotonicity for scalar variables and are inadequate for detecting monotonic subscript arrays. Lin and Padua [38–40] presented a compile-time technique to analyze the content of index arrays and automatically parallelize loops. Their techniques made use of interprocedural query propagation to detect various array properties in a demand-driven manner. Their technique can detect certain properties in specific types of loops. For example, their technique can determine an injective subscript array only in loops, wherein the value of the loop index variable is assigned to the subscript array (index gathering loops). In addition, properties, such as the closed-form distance, are detected using pattern matching. Their technique is incapable of detecting index array properties in loops with recurrence relationships presented in Section 3. By contrast, our technique is capable of analyzing various classes of loop patterns that define and modify subscript arrays, using symbolic range aggregation and manipulation. In doing so, the technique derives subscript array properties that are sufficient for eventual parallelization.

A technique that can produce precise information for the forms of loops that compile-time techniques usually attempt to parallelize is the aggregation of information gathered in the loop body across the iteration space. This method was applied by Tu and Padua in array privatization [15] to analyze array sections that are defined and used. We use a similar method, extended to capture the effect of certain recurrence relationships, which allow us to gather such array properties, as monotonicity.

6.3. Interactive Cetus (iCetus)

There are a variety of tools developed over time for parallelizing sequential codes with differing degrees of user involvement.

ParTool [41] is a feedback-directed parallelizer, built over the ROSE compiler infrastructure [17].The tool automatically parallelizes the serial code by inserting OpenMP Pragmas into the code. It performs data-dependence analysis provided by ROSE to ascertain whether a loop nest is safe to parallelize. If not, the dependencies that prevent parallelization are provided as feedback to the user. Command-line flags are the means by which ParTool offers its functionalities. In contrast, iCetus has a web-based design, facilitating easy user interaction.

The ParaScope Editor (PED) [42] is an interactive parallel programming tool developed at Rice University that supports scientific Fortran programmers. PED displays data-dependence information and offers a variety of source-to-source transformations for the user to choose from. Data-dependence information was perceived as being too low level by users, and they require assistance with program transformations. Furthermore, PED does not integrate dynamic performance data.

HTGviz [43] is an interactive parallelization environment implemented on top of the Parafrase-2 parallelizing compiler [44]. It provides various views to the user, such as the task graph view, the serial code view, the directive view to insert OpenMP tags, and the parallel code view. Interaction between the compiler and user happens via the hierarchical task graph (HTG) program representation, where task parallelism is represented by precedence relations (arcs) among task nodes. There is no support for measuring the parallelization benefits or displaying potential parallelism. In contrast, iCetus shows the performance gain for the code after optimization.

Parceive [45] is an interactive tool that operates on user applications in binary form. It supports the parallelization of applications written in C, C++, and C# by dynamic instrumentation of binaries and providing a visualization environment to detect parallelism at several levels and not just the loops and instructions. The visualization environment offers three different views: performance view, calling context tree (CCT) view, and source view. While the performance view is an interactive representation of a programs profiling and trace data, the CCT view displays a calling context tree consisting of call nodes, loop nodes, and memory nodes. The source view shows the source code of the instrumented application. In contrast, the focus of iCetus is on understanding the transformed, OpenMP-annotated code.

iCetus distinguishes itself from these previous efforts in three main ways: (i) Building on one of the most advanced parallelizers, the Cetus compiler, the tool allows the user to inspect in detail the results of different compiler analyses, such as data-dependence analysis, variable range analysis, and private variable analysis, in an easy to understand format. (ii) The tool provides the user with dynamic analysis information of the program, such as the speedup gained from a transformation, enabling the user to judge when further optimizations may be beneficial or have a diminishing return. (iii) The tool supports the user in all phases of the program optimization process by providing feedback while enabling the user to edit the input code and re-run the optimizer.

7. Conclusions

We presented an overview of the Cetus source-to-source compiler infrastructure. We evaluated the various analysis and transformation techniques in Cetus using the latest version of the NAS and PolyBench benchmark suites. We also described and evaluated the recent key advances of Cetus: subscripted subscript analysis and interactive parallelization.

Cetus could achieve significant performance improvement in 50% of the applications from the NAS Parallel benchmark suite and about 75% of the applications from the PolyBench suite. Previous studies also found success in 50% of science/engineering applications. Through the evaluation of individual optimization techniques, we found that techniques that were important in previous generations of compilers are also among the key

optimizations today. Even though both benchmark codes and architectures have evolved significantly, existing autoparallelization techniques are still successful in optimizing the application codes.

We also presented a novel compile-time analysis method for subscripted subscripts, which can symbolically analyze the content of subscript arrays to successfully parallelize an important class of programs exhibiting sparse matrix patterns. We applied this technique by hand to the Supernodal Cholesky factorization and sparse matrix scaling application codes. Our technique yielded a performance improvement of 383% for the Supernodal Cholesky factorization application code and 739% for the sparse matrix scaling code, compared to the best alternative.

State-of-the-art parallelizing compilers are batch-oriented tools, limited to static program analyses and transformations. iCetus is a new web application that involves the user in the code transformation process by providing feedback and enabling the user to edit the input serial code and the parallel code. It provides static and dynamic analysis information to guide this process. Users of varying skill sets found the implemented as well as the proposed features in iCetus to be both interesting and useful. The next release of the tool will incorporate more features in support of interactivity as well as such capabilities as a loop-level profiler and an auto-tuner.

Author Contributions: Conceptualization, A.B. and R.E.; methodology, R.E.; validation, A.B., M.R.R. and P.B.; formal analysis, A.B. and P.B.; investigation, M.R.R.; data curation, M.R.R. and P.B; writing—original draft preparation, A.B., P.B. and R.E.; writing—review and editing, A.B. and P.B.; supervision, R.E.; project administration, R.E.; funding acquisition, R.E. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the University of Delaware and by the National Science Foundation under awards 2112606, 2125703, 1931339, 1919839 and 1833846.

Data Availability Statement: Source codes to reproduce all the results described in this paper can be found at: (https://github.com/akshay9594/Polybench-4.2, Polybench-4.2) (accessed on 1 December 2021); (https://www.nas.nasa.gov/software/npb.html, NPB-3.3) (accessed on 1 November 2021).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dave, C.; Bae, H.; Min, S.-J.; Lee, S.; Eigenmann, R.; Midkiff, S. Cetus: A source-to-source compiler infrastructure for multicores. *IEEE Comput.* **2009**, *42*, 36–42. [CrossRef]
2. Bae, H.; Mustafa, D.; Lee, J.; Lin, H.; Dave, C.; Eigenmann, R.; Midkiff, S. The cetus source-to-source compiler infrastructure: Overview and evaluation. *Int. J. Parallel Program.* **2013**, *41*, 753–767. [CrossRef]
3. Mustafa, D.; Eigenmann, R. Performance analysis and tuning of automatically parallelized OpenMP applications. In Proceedings of the International Workshop on OpenMP, Chicago, IL, USA, 13–15 June 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 151–164
4. Bhosale, A.; Eigenmann, R. On the automatic parallelization of subscripted subscript patterns using array property analysis. In Proceedings of the ACM International Conference on Supercomputing, New York, NY, USA, 14–17 June 2021; pp. 392–403.
5. Bailey, D.; Barszcz, E.; Barton, J.; Browning, D.; Carter, R.; Dagum, L.; Fatoohi, R.; Frederickson, P.; Lasinski, T.; Schreiber, R.; et al. The NAS Parallel Benchmarks. *Int. J. Supercomput. Appl.* **1991**, *5*, 63–73. [CrossRef]
6. Yuki, T.; Pouchet, L.N. PolyBenchC-4.2.1. Available online: https://github.com/MatthiasJReisinger/PolyBenchC-4.2.1/blob/master/polybench.pdf (accessed on 5 January 2021).
7. Lee, S.; Min, S.; Eigenmann, R. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *ACM Sigplan Not.* **2009**, *44*, 101–110. [CrossRef]
8. Basumallik, A.; Eigenmann, R. Towards automatic translation of OpenMP to MPI. In Proceedings of the 19th annual international conference on Supercomputing, Cambridge, MA, USA, 20–22 June 2005; pp. 189–198.
9. Johnson, T.; Lee, S.; Fei, L.; Basumallik, A.; Upadhyaya, G.; Eigenmann, R.; Midkiff, S.P. *Experiences in Using Cetus for Source-to-Source Transformations*; Springer: Berlin/Heidelberg, Germany, 2004; pp. 1–14.
10. William, B.; Rudolf, E. The Range Test: A Dependence Test for Symbolic, Non-linear Expressions. In Proceedings of the ACM/IEEE Conference on Supercomputing, Washington, DC, USA, 14–18 November 1994; pp. 528–537.
11. Wolfe, M.; Banerjee, U. Data dependence and its application to parallel processing. *Int. J. Parallel Program.* **1987**, *16*, 137–178. [CrossRef]

12. Emami, M.; Ghiya, R.; Hendren, L.J. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Acm Sigplan Not.* **1994**, *29*, 242–256. [CrossRef]
13. William, B.; Rudolf, E. Symbolic Range Propagation. In Proceedings of the 9th International Symposium on Parallel Processing, Santa Barabara, CA, USA, 25–28 April 1995; pp. 357–363.
14. Dagum, L.; Menon, R. OpenMP: An industry standard API for shared-memory programming. *IEEE Comput. Sci. Eng.* **1998**, *5*, 46–55. [CrossRef]
15. Peng, T.; David, P. Automatic Array Privatization. In Proceedings of the Lecture Notes in Computer Science: Languages and Compilers for Parallel Computing: 6th International Workshop, Portland, OR, USA, 12–14 August 1993; Volume 768, pp. 500–521.
16. Kennedy, K.; McKinley, K.S. Optimizing for parallelism and data locality. In Proceedings of the 6th international conference on Supercomputing, Washington, DC, USA, 19–24 July 1992; pp. 323–334.
17. Quinlan, D.; Liao, C.L. Quinlan, D.; Liao, C. The ROSE source-to-source compiler infrastructure. In Proceedings of the Cetus Users and Compiler Infrastructure Workshop, in Conjunction with PACT, Galveston Island, TX, USA, 10 October 2011; Springer: Berlin/Heidelberg, Germany, 2011; Volume 1.
18. Automatic Parallelization with Intel Compilers. Available online: https://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers (accessed on 21 October 2019).
19. Utpal, B.; Rudolf, E.; Alexandru, N.; David, P. Automatic Program Parallelization. *Proc. IEEE* **1993**, *81*, 211–243.
20. Davis, T.A. *Direct Methods for Sparse Linear Systems*; Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2006.
21. Henning, J.L. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comput. Archit. News* **2006**, *34*, 1–17. [CrossRef]
22. Heroux, M.A.; Doerfler, D.W.; Crozier, P.S.; Willenbring, J.M.; Edwards, H.C.; Williams, A.; Rajan, M.; Keiter, E.R.; Thornquist, H.K.; Numrich, R.W. *Improving Performance via Mini-Applications*; Technical Report; Sandia National Laboratories: Albuquerque, NM, USA; Livermore, CA, USA, 2009.
23. Bhosale, A.; Eigenmann, R. Compile-time parallelization of subscripted subscript patterns. In Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), New Orleans, LA, USA, 28 July 2020; pp. 317–325.
24. Jin, H.; Frumkin, M.; Yan, J. *The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance*; NASA Ames Research Center: Silicon Valley, CA, USA, 1999.
25. NAS Parallel Benchmarks C Version. Available online: http://aces.snu.ac.kr/software/snu-npb/ (accessed on 3 January 2019).
26. Modified Version of PolyBench-4.2. Available online: https://github.com/akshay9594/Polybench-4.2 (accessed on 23 December 2021).
27. Blackford, L.S.; Petitet, A.; Pozo, R.; Remington, K.; Whaley, R.C.; Demmel, J.; Dongarra, J.; Duff, I.; Hammarling, S.; Henry, G.; et al. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Softw.* **2009**, *42*, 135–151.
28. Anderson, E.; Bai, Z.; Bischof, C.; Blackford, L.; Demmel, J.; Dongarra, J.; Du Croz, J.; Greenbaum, A.; Hammarling, S.; McKenney, A.; et al. *TLAPACK Users' Guide*; SIAM: Philadelphia, PA, USA, **1999**.
29. Davis, T.; Hu, Y. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw. (TOMS)* **2011**, *38*, 1–25. [CrossRef]
30. Harel, R.; Mosseri, I.; Levin, H.; Alon, L.; Rusanovsky, M.; Oren, G. Source-to-source parallelization compilers for scientific shared-memory multi-core and accelerated multiprocessing: Analysis, pitfalls, enhancement and potential. *Int. J. Parallel Program.* **2020**, *48*, 1–31. [CrossRef]
31. AutoPar. Available online: https://github.com/rose-compiler/rose/wiki/ROSE-based-tools#autopar (accessed on 29 August 2021).
32. Amini, M.; Creusillet, B.; Even, S.; Keryell, R.; Goubier, O.; Guelton, S.; McMahon, J.O.; Pasquier, F.; Péan, G.; Villalon, P.; et al. Par4all: From Convex Array Regions to Heterogeneous Computing. In Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques, Paris, France, 23–25 January 2012.
33. Mosseri, I.; Alon, L.; Harel, R.; Oren, G. *ComPar: Optimized Multi-Compiler for Automatic OpenMP S2S Parallelization. International Workshop on OpenMP*; Springer: Berlin/Heidelberg, Germany, 2020; pp. 247–262.
34. Blume, W.; Eigenmann, R. Performance analysis of parallelizing compilers on the Perfect BenchmarksTM Programs. *IEEE Trans. Parallel Distrib. Syst.* **1992**, *3*, 643–656. [CrossRef]
35. McKinley, K. *Dependence Analysis of Arrays Subscripted by Index Arrays*; Technical Report CRPC-TR91187; Rice Univ.: Houston, TX, USA, July 1991.
36. Gutiérrez, E.; Asenjo, R.; Plata, O.; Zapata, E.L.L. Automatic parallelization of irregular applications. In *Parallel Computing*; Elsevier: Amsterdam, The Netherlands, 2000; Volume 26, pp. 1709–1738.
37. Spezialetti, M.; Gupta, R. Loop monotonic statements.*IEEE Trans. Softw. Eng.* **1995**, *21*, 497–505. [CrossRef]
38. Lin, Y.; Padua, D. Compiler Analysis of Irregular Memory Accesses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*; ACM: New York, NY, USA, 1 May 2000; pp. 157–168.
39. Lin, Y.; Padua, D. Demand-Driven Interprocedural Array Property Analysis. In Proceedings of the International Conference on Compiler Construction, London, UK, 5–13 April 2014; Springer: Berlin/Heidelberg, Germany, 2000; pp. 202–218.
40. Lin, Y.; Padua, D. *Analysis of Irregular Single-Indexed Array Accesses and Its Applications in Compiler Optimizations. International Workshop on Languages and Compilers for Parallel Computing*; Springer: Berlin/Heidelberg, Germany, 1999; pp. 303–317.
41. Varun, M.; Sanjeev, K.; Aggarwal, O.T.; Pen-Chung, Y.; Binyu, Z. ParTool: A Feedback-Directed Parallelizer. In *Advanced Parallel Processing Technologies*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 157–171.

42. Balasundaram, V.; Kennedy, K.; Kremer, U.; McKinley, K.; Subhlok, J. The Parascope editor: An interactive parallel programming tool. In Proceedings of the 1989 ACM/IEEE Conference on Supercomputing, Ser. Supercomputing 89, Reno, NV, USA, 12–17 November 1989; ACM: New York, NY, USA, 1989; pp. 540–550.
43. Giordano, M.; Furnari, M.M. HTGviz: A graphic tool for the synthesis of automatic and user-driven program parallelization in the compilation process. In Proceedings of the Second International Symposium on High Performance Computing, Ser. ISHPC '99, Kyoto, Japan, 26–28 May 1999; Springer: London, UK, 1999; pp. 312–319.
44. Polychronopoulos, C.; Girkar, M.; Haghighat, M.; Lee, C.; Leung, B.; Schouten, D. PARAFRASE-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. *Int. J. High Speed Comput.* **1989**, *1*, 45–72. [CrossRef]
45. Wilhelm, A.; Savu, V.; Amadasun, E.; Gerndt, M.; Schuele, T. A Visualization Framework for Parallelization. In Proceedings of the 2016 IEEE Working Conference on Software Visualization (VISSOFT), Raleigh, NC, USA, 3–4 October 2016; pp. 81–85.