Int J Parallel Prog (2013) 41:753–767
DOI 10.1007/s10766-012-0211-z

The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation

Hansang Bae · Dheya Mustafa · Jae-Woo Lee · Aurangzeb · Hao Lin · Chirag Dave · Rudolf Eigenmann · Samuel P. Midkiff

Received: 24 January 2012 / Accepted: 23 July 2012 / Published online: 10 August 2012
© Springer Science+Business Media, LLC 2012

Abstract This paper provides an overview and an evaluation of the Cetus source-to-source compiler infrastructure. The original goal of the Cetus project was to create an easy-to-use compiler for research in automatic parallelization of C programs. In meantime, Cetus has been used for many additional program transformation tasks. It serves as a compiler infrastructure for many projects in the US and internationally. Recently, Cetus has been supported by the National Science Foundation to build a community resource. The compiler has gone through several iterations of benchmark studies and implementations of those techniques that could improve the parallel

H. Bae (✉) · D. Mustafa · J.-W. Lee · Aurangzeb · H. Lin · R. Eigenmann · S. P. Midkiff
Purdue University, West Lafayette, IN 47907, USA
e-mail: baeh@purdue.edu

D. Mustafa
e-mail: dmustaf@purdue.edu

J.-W. Lee
e-mail: jaewoolee@purdue.edu

Aurangzeb
e-mail: orangzeb@purdue.edu

H. Lin
e-mail: lin116@purdue.edu

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

S. P. Midkiff
e-mail: smidkiff@purdue.edu

C. Dave
Qualcomm, San Diego, CA 92121, USA
e-mail: cdave@qualcomm.com

performance of these programs. These efforts have resulted in a system that favorably compares with state-of-the-art parallelizers, such as Intel's ICC. A key limitation of advanced optimizing compilers is their lack of runtime information, such as the program input data. We will discuss and evaluate several techniques that support dynamic optimization decisions. Finally, as there is an extensive body of proposed compiler analyses and transformations for parallelization, the question of the importance of the techniques arises. This paper evaluates the impact of the individual Cetus techniques on overall program performance.

Keywords Automatic parallelization · Compiler infrastructure · Source-to-source translation · Performance

1 Introduction

Cetus is an infrastructure for research on compiler optimizations for multicores, with an emphasis on automatic parallelization. We have created a compiler infrastructure that supports source-to-source transformations, is user-oriented, easy to use, and provides the most important parallelization passes as well as the underlying enabling techniques.

This paper provides an overview of Cetus. A more in-depth description is found in [13]. We will put emphasis on the description on new Cetus capabilities; in particular, we will discuss the important issue of making optimization decisions at runtime.

Cetus is already in use by a number of research groups in the US and worldwide [2,3,5,11,17,21,27,33,34]. In our ongoing work, we are applying the infrastructure for creating translators that convert shared-memory programs written in OpenMP into other models, such as message-passing and CUDA (for Graphics Processing Units) [20].

This paper also pursues the question of the importance of the individual optimization techniques in a parallelizer. Previous studies of this question have discussed the effectiveness of automatic parallelization and restructuring techniques on the Perfect Benchmarks [7,14]. The techniques analyzed included *reduction substitution, recurrence substitution, induction variable elimination, scalar expansion, forward substitution, stripmining,* and *loop interchange*. The studies found that 50 % of the programs showed a respectable improvement from autoparallelization 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 [19,26,29] and the number of loops that can be parallelized [9].

Building on the methodologies applied in these previous studies, the present paper comprehensively evaluates modern autoparallelizers and their underlying techniques for today's multicores. We present overall program performance results for the NAS Benchmarks when parallelized by Cetus, by Intel's ICC compiler, and by hand. We evaluate the effectiveness of dynamic optimization decision support and measure the importance of the individual analysis and transformation techniques in Cetus. A key idea of our methodology is the use of a *customizable empirical tuning system* [24],

which is part of our parallelizer. Used for dynamic optimization decision support, this system can navigate through a large search space of optimization variants and identify the best. Used for our evaluation study, the same system supports the comprehensive exploration of compiler techniques that can be applied to individual loops in our program suite. Furthermore, by exploring combinations of techniques in many variants, this methods can identify interactions between techniques.

2 Components of Cetus

Cetus consists of a C parser, an internal representation, and a set of analysis and transformation passes with emphasis on automatic parallelization. In this section we overview the internal representation and the passes currently available in the Cetus compiler.

2.1 Internal Representation

Cetus' internal program representation (IR) is implemented in the form of a Java class hierarchy. A high-level representation provides a syntactic view of the source program to the pass writer, making it easy to understand, access and transform the input program. For example, the *Program* class type represents the entire program that may consist of multiple source files. Each source file is represented as a *TranslationUnit*. Other base IR object types are *Statement, Declaration* and *Expression*. Specific source constructs are represented in the IR by classes that are derived from these base classes, e.g., *ExpressionStatement* represents a *Statement* that contains an *Expression*, and an *AssignmentExpression* represents an *Expression* that assigns the value of the right-hand side to the left-hand side. Figure 1 illustrates how the input program is represented in Cetus. There is complete data abstraction, and pass writers only manipulate the IR through access functions. Important features of the IR include:

- **Traversable objects:** All Cetus IR objects are derived from a base class *Traversable*. This class provides the functionality to iterate over lists of objects in a generic way.
- **Iterators:** Breadth-first, depth-first and flat iterators are built into the functionality to provide easy traversal and search over the program IR.
- **Symbol table:** Cetus' symbol table interface provides information about identifiers and data types. Its implementation makes direct use of the information stored in declaration statements in the IR. There is no separate and redundant symbol table storage.
- **Annotations:** Comments, pragmas, directives, and other auxiliary information about IR objects can be stored in *Annotation* objects. An annotation may be associated with a statement (e.g., an OpenMP directive belonging to a *for* statement) or may stand independently (e.g., a comment line).

```
0 /* file: a.c */
1 int temp;
2 int main(void)
3 {
4    int i, j, c, a[100];
5    c = 2;
6    for (i = 0; i < 100; i++) {
7        a[i] = c*a[i] + a[i];
8    }
9 }
```

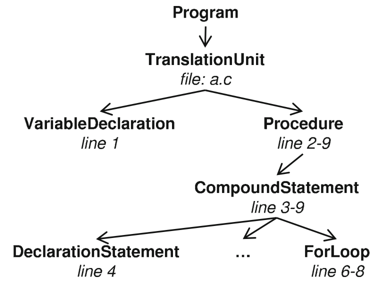

Fig. 1 An input program and its internal representation in Cetus. (Full internal representations down to expressions are omitted)

Table 1 Major passes implemented in Cetus

General passes	Parallelization passes
Symbolic analysis	Induction variable recognition/substitution
Points-to/alias analysis	Reduction recognition/transformation
Function inlining	Scalar/array privatization
USE/DEF chain analysis	Data dependence analysis
Miscellaneous	Loop parallelizer

2.2 Analysis and Transformation Passes

The main capability of the Cetus compiler is automatic parallelization, and many of its passes were developed to support this functionality. Advanced program analysis and transformation techniques are required to achieve good automatic parallelization. These techniques range from general analysis, such as alias analysis, to parallelization-specific techniques, such as privatization. We overview the passes implementing these techniques in two categories, general passes and parallelization passes, as summarized in Table 1. General passes support automatic parallelization as enabling techniques.

2.2.1 Symbolic Analysis

Like its predecessor Polaris [6,22], a key feature of Cetus is its ability to analyze the represented program in symbolic terms. This ability is implemented in terms of a powerful expression manipulator and range analysis, which propagates value ranges of variables through the program. The ability to manipulate symbolic expressions is essential when designing analysis and transformation algorithms that deal with real programs. For example, a data dependence test can easily extract the necessary information if array subscripts are normalized with respect to the relevant loop indices. Cetus supports such expression manipulations with tools that simplify and normalize symbolic expressions. Figure 2a shows examples of these capabilities.

Range analysis provides an environment in which the values of two symbolic expressions can be compared at any program point. This environment consists of the range propagation pass, which collects value ranges of integer-typed variables,

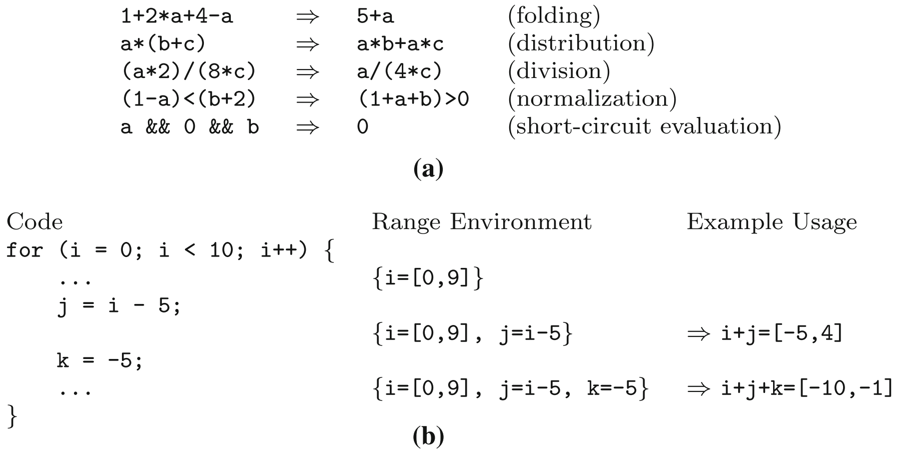

Fig. 2 Symbolic expression simplification and normalization provided in Cetus (**a**) and example usages of symbolic range analysis (**b**)

and the manipulation interfaces. The latter enables value comparisons or manipulations, such as constant propagation/substitution and bounding values of expressions. Figure 2b shows a code section, its range environment, and an example usage that bounds expressions. Range analysis and its manipulation techniques are widely used in Cetus passes. They enable the passes to obtain and reason about compile-time information available in the program.

2.2.2 Points-to and Alias Analysis

Pointer variables in the C language complicate all aspects of compiler analysis and transformations; alias analysis is a necessary prerequisite for other passes' correct performance. The alias analysis pass implemented in Cetus was built on top of points-to analysis [16], with some variations from the original algorithm: The analysis pass in Cetus does not provide context-sensitive results, and heap-allocated objects are handled conservatively. We identified that disproving alias relationships among arrays is most important and developed a simple but effective analysis. A full description of our points-to analysis pass is provided in [12].

Alias analysis is performed in two steps. First, the points-to analysis collects points-to relations at each program point, interprocedurally. The resulting points-to relations contain mappings from pointer variables to the dereferenced memory locations. Second, an alias set for each variable is created from the points-to relations. A variable is alias-free if the returned set is empty. Figure 3 illustrates these steps for the given code example.

2.2.3 Function Inlining

Cetus provides function inlining in place of complete interprocedural analysis and transformations. Aggressive function inlining can decrease readability. We also found

Input Code	Points-to/Alias Analysis Result
`double a[100];`	
`double *b = &a[50];`	⇒ points-to relation: (b,a,P)
`for (i = 0; i < 50; i++) {`	⇒ alias set for a: {a,b}
`    b[i] = ...`	⇒ alias set for b: {b,a}
`    ... = a[i+20];`	
`}`	

Fig. 3 Example of alias analysis. The analysis result shows the points-to relation and the alias sets available immediately before the loop. `(b,a,P)` means `b` *probably* points to the memory location named `a`

that the parallel coverage tends to be less than when using selective inlining. Cetus provides flexible support for inlining. This support can be utilized by the users of Cetus, by Cetus transformation passes, and by the tuning framework. While Cetus allows the clients to perform aggressive inlining, it also provides a rich set of command line options as well as an API for selective inlining. Inlining can be performed inside selected functions and code blocks. Similarly, selected functions can be marked as candidates of inlining at their call sites. Cetus also provides pragma annotations for selective inlining.

2.2.4 USE/DEF Chain Analysis

The use-def chain computation is a flow-sensitive and context-insensitive inter-procedural analysis, based on the program summary graph [10]. At a high level, the algorithm proceeds as follows: The pass first executes reaching-definition and upward-exposed use analysis on each procedure to generate the program summary information. The initial analysis is done on the following code segments: between the procedure entry and the first call site, between call sites, and between the last call site and the end of the procedure. Second, use and def lists are placed at procedure entry, exit, and call sites. Third, use and def information is propagated along edges in the program summary graph; the edges associate formal and actual procedure parameters. Fourth, reaching definition analysis is executed again using the computed program summary information. Finally, the use-def and def-use chains are computed using the reaching definition analysis result and the upward-exposed use information. Cetus' alias information is used during the use-def chain computation.

2.2.5 Induction Variable Recognition and Substitution

Induction variable recognition and substitution is one of the techniques that remove data dependences on scalar variables. An induction statement has a recurrence form, `iv = iv + expr`. This statement needs to be replaced by a form that does not induce a data dependence. If the right-hand side in the statement can be expressed as a closed-form expression that does not contain `iv`, the dependence will be removed. Cetus detects and substitutes induction variables where `expr` is either loop-invariant or another induction variable. The pass visits every statement in a loop nest and symbolically computes at each statement the increments of induction variables since the entry to the loop nest. It then adds the increments to every use of the induction

Input Code
```
k = 0;
for (i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        k += i;
        ...
    }
    a[k] = ...;
}
```
⇒

Transformed Code
```
k = 0;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        ...
    }
    a[(i*n+i*i*n)/2] = ...;
}
```

Fig. 4 Induction variable recognition and substitution. Cetus can handle *generalized induction variables*, where the increment is another induction variable. In this case, k is incremented by the loop variable i

variables and removes the induction statements. Figure 4 shows the original code with an induction statement and the transformed code without the induction statement.

2.2.6 Reduction Recognition and Transformation

Reduction operations are used in many computational applications. They commonly take the form `rv = rv + expr`. Recognizing such operations is key to successfully auto-parallelizing many loops. A data dependence analyzer will report a dependence on a reduction operation unless it is marked as a reduction operation. Cetus' reduction variable analyzer detects additive reduction variables that satisfy the following criteria:

- First, the loop contains one or several assignment expressions of the form `rv = rv + expr`, where `rv` is either a scalar variable or an array access, and `expr` is typically a real-valued, loop-variant expression.
- Second, `rv` appears nowhere else in the loop.

The result of this recognition pass is used in the loop-parallelization pass which ignores the data dependences carried by the reduction statements. In addition, a reduction transformation pass is invoked to convert parallel loops containing recognized reductions into parallel forms: Scalar variables in reduction statements are simply marked within an OpenMP clause while reduction operations on array variables are converted into a form with local summation and synchronized global update. Figure 5 shows an example.

2.2.7 Scalar and Array Privatization

Identifying private variables in a loop is an important step an automatic parallelizer has to perform. A private variable is a variable that is used as a temporary variable in a loop; it is written first and used later in the loop. Array sections are used as temporary locations for private array variables. Private variables do not need to be exposed to the other threads at runtime, so the data dependence analyzer can safely assume these variables do not have dependences. We implemented a simple but effective array privatizer in Cetus, which can handle array sections containing symbolic terms; it is a simplified version based on the technique described in [30]. The array privatizer traverses a loop nest from the innermost to the outermost loop while collecting *defined* (written), *used*, and *upward-exposed* (used but not defined since the loop entry) array sections

Input Code		Transformed Code
		`#pragma omp parallel private(qqreduce)`
		`{`
		`qqreduce[:] = 0;`
		`#pragma omp for ... reduction(+: sx, sy)`
`for (k = 1; k <= np; k++) {`		`for (k = 1; k <= np; k++) {`
`...`		`...`
`l = ...`		`qqreduce[l] += 1.0;`
`qq[l] += 1.0;`	⇒	`sx = sx + t3;`
`sx = sx + t3;`		`sy = sy + t4;`
`sy = sy + t4;`		`...`
`...`		`}`
`}`		`#pragma omp critical`
		`qq[:] += qqreduce[:];`
		`}`

Fig. 5 Input code and the resulting code after reduction recognition and transformation. `qqreduce` is initialized as a private copy for each thread to store the local sums ([:] notation indicates "entire array"). Scalar reductions are marked as such with an OpenMP clause; array reductions are transformed explicitly

or scalar variables. Next, it identifies private variables by checking if there are no upward-exposed uses for the variables. To improve the accuracy, the privatizer makes use of Cetus' symbolic range analysis. For example, when computing upward-exposed uses, the intersection of two must-defined sections $[1 : m] \cap [1 : n]$ is $[1 : min(m, n)]$; knowledge of the symbolic relationship $n \leq m$ allows the result to be simplified to $[1 : n]$.

2.2.8 Data Dependence Analysis

Data dependence analysis is a *memory disambiguation* technique that tries to identify data references accessing the same memory location during program execution and characterize dependences between those references. Array data dependence analysis involves the process of analyzing array subscripts in order to disprove that two computations access the same elements of an array. In a loop, these subscripts are usually functions of the loop index variables. Data dependence tests try to find integer solutions to systems of equations defined under loop and direction vector constraints to analyze the dependences between array accesses.

Cetus implements an array data dependence analyzer. An information-collection wrapper interfaces with the IR to collect array access-related and loop-related information. It currently handles all canonical loops of the form `for (i=lb;i<ub;i+=inc)`. Advanced symbolic range analysis (Sect. 2.2.1) is used to simplify loop-related information and array subscripts in order to obtain simple affine expressions that can be evaluated for dependence. The wrapper feeds into a data dependence test framework that currently uses the Banerjee-Wolfe inequalities to return direction vector information for the dependences [1,32]. Cetus also contains the range test [8] as an alternative dependence test.

All dependences identified within a loop nest are appended to the program data dependence graph, which is attached to the program's IR. This information is then available to all Cetus analysis and transformation passes through appropriate interface routines.

2.2.9 Loop Parallelizer and Code Generator

The loop parallelization pass analyzes the results of parallelization-enabling passes and marks loops that do not carry any data dependences. Data dependences on scalar variables and array variables are analyzed differently by this pass. For scalar variables, the information produced by the reduction and privatization passes is used to eliminate dependences; for array variables, the data dependence graph is searched to determine dependences. A loop is marked as parallel if no scalar variable carries dependences and all dependence arcs in the graph show non-loop-carried dependences with respect to the loop.

The final step of automatic parallelization is code generation for a specific target language. Cetus generates OpenMP program from the input program and the analyzed result. This final step is straightforward, because the internal annotations inserted by the passes are similar to OpenMP directives.

2.2.10 Dynamic Optimization Decision Support

A key challenge for optimizing compilers is to decide where and when to apply their techniques, given limited knowledge about the program's input data and runtime environment. In many programs or program sections, eager parallelization will lead to performance degradation instead of speedup. This is especially true for small, inner loops; it also holds for transformation techniques that apply significant code changes, such as reduction parallelization, tiling, and subroutine inlining. For subroutine inlining, recent work [18] reports how to reduce negative effects of inlining with respect to interprocedural parallelization.

Profitability Tests: Cetus includes several methods to deal with this issue. By default, the compiler uses a simple performance model, eliminating small parallel loops that are likely to cause overheads. The model estimates a loop's workload as the product of the number of statements and loop iterations. Loops with workloads that are less than a threshold do not get parallelized. If the workload expression cannot be evaluated at compile-time, the technique uses an OpenMP IF construct, which decides at runtime on parallel or serial execution. We evaluate this heuristic, which we refer to as *model-based profitability test*.

Cetus also supports a profitability test that is based on profiling. The basic idea is to use profiled runtime of a sequential execution and a parallel execution of a parallel loop and to decide whether the loop should be executed in parallel after the profiling. A parallel loop is executed in three different phases: *grace phase* where the loop is executed in parallel, *profile phase* where the loop is profiled several times both in serial and in parallel, and *stable phase* where the loop is executed either serially or in parallel, based on the profile results. Users can vary the behavior of each phase through Cetus options. One limitation of this method is its sensitivity to variations in the loop execution time over the course of the program execution.

Empirical Tuning: Improved runtime decision making in Cetus comes through an automatic tuning capability, which 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 optimization 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 the production data set.

Cetus uses a methods called window-based empirical tuning [23]. A problem with most approaches that try to find the best combination of all possible optimizations for all sections of a program is that tuning times can be very long. This is because optimization techniques tend to interact and thus all combinations of techniques on all loops would need to be tried for finding the best combination. Cetus' tuner only considers the interactions of techniques within *windows*, which are code sections in close proximity and optimization techniques that are likely to interact. Specifics of this method are described in [24]. The user can 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.

3 Performance Evaluation

We present two sections of measurements: overall performance and the performance of individual techniques. Overall performance measurements compare speedups over the sequential versions of the Cetus-parallelized programs versus hand parallel and versus automatically tuned programs; we also measure the importance of dynamic optimization decision support. For evaluating the impact of individual techniques, we started from the fully optimized programs; then we disabled one compiler technique at a time, and measured the reduced performance. Before presenting these results, we briefly describe the used benchmarks and the experiment setup.

3.1 Benchmarks Characteristics

The NAS Parallel Benchmarks (NPB) are a set of programs designed to help evaluate the performance of parallel supercomputers. The benchmarks are derived from Computational Fluid Dynamics (CFD) applications. They consist of five kernels and three pseudo-applications. We used two classes of problems in the evaluation—Class W and Class A. Class W was used as a training data set for the window-based tuning system, while class A served as a production data set. The classes of problems in NPB differ mainly in the sizes of principle arrays, which generally affect the number of iterations of contained loops [4,31]. As a real OpenMP application, the NPB suite is ideal for benchmarking parallelizing compilers. Another advantage of the suite is its availability in efficient hand implementations of C code [28]. We used this suite as the input to our automatic parallelizer after turning off the OpenMP runtime library calls and ignoring the OpenMP directives.

3.2 Experiment Setup

We conducted experiments using a single-user x86-64 machine with two 2.5 GHz Quad-Core AMD 2380 processors and a 32 GB memory, running Red Hat Enterprise

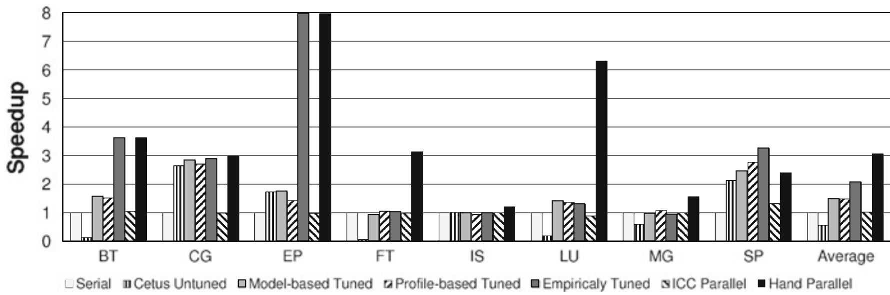

Fig. 6 Overall Performance of the parallelizing compilers on NPB suite using A dataset. Cetus Untuned shows performance for compiler-parallelized code using Cetus. Model-based Tuned represents the speedup for the automatically parallelized programs using model-based profitability test available in Cetus. Profile-based Tuned represents the speedup for the automatically parallelized programs using profile-based profitability test available in Cetus. Empirically Tuned shows speedup for tuned programs using the window-based tuning system. Hand Parallel shows speedup for original NPB programs. ICC Parallel shows speedup for parallelized code using "-parallel" ICC flag; other versions were compiled with the "-openmp" flag to enable only the OpenMP translation

Linux. We used the Intel ICC compiler version 11.1 with "-O3" optimization, both as a back-end code generator for Cetus and as an autoparallelizer that we compare Cetus to in the experimental results.

3.3 Overall Evaluation of Tuned Autoparallelization

We measure the speedup of automatically parallelized programs by Cetus and ICC and compare it to the hand-parallelized programs as well as the tuned programs using window-based tuning, model-based as well as profile-based profitability testing. Figure 6 shows the speedups of the NPB programs using dataset A.

We produced the serial codes of the Benchmarks using the backend compiler (ICC) without the OpenMP parallel option. *Hand Parallel* refers to the original parallel benchmarks. *Cetus Untuned* is the automatically parallelized code without tuning. *Empirically Tuned* presents transformed programs that are improved using the window-based tuning system. *Model-Based* uses Cetus' model-based profitability test. *Profile-Based* uses Cetus' profile-based profitability test. *ICC Parallel* shows speedup using the ICC autoparallelization option ("-parallel" flag).

Despite a conservative profitability test, ICC could not guarantee non-degrading performance. Cetus with its model-based profitability test outperforms ICC in all programs that show noticeable speedup. Window-based tuning adds substantially to this speedup, matching or exceeding the hand-parallelized performance in two cases. In CG and SP, the parallel coverage exceeds 90 % and the results show a speedup on both Cetus Untuned and Cetus Tuned versions; SP Tuned outperforms the hand parallel version. One important reason is that the hand parallel version parallelizes loops at the second level, while the Cetus tuner selects the outer loop to be parallel. Another reason is that Cetus Untuned parallelizes profitable small loops that were considered inefficient by the programmer of the parallel code.

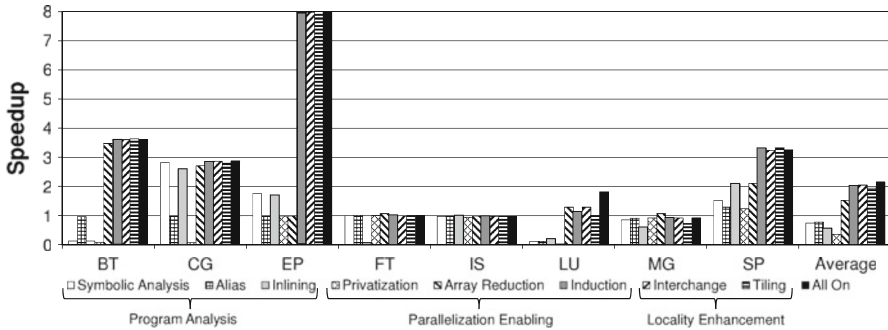

Fig. 7 Performance contribution of individual optimization techniques on NPB benchmarks using dataset A. The selected base case is the best tuned version (all on). One technique at a time is turned off for the versions labeled with the specified techniques; all other techniques remain fixed

While investigating the effect of inlining on parallel coverage, we found that the performance of BT and EP is significantly improved by inlining as we will show later. IS' low parallel coverage explains the lack of parallel speedup.

In FT, LU, and MG, the parallel coverage is around 50 %. However, performance is low since most parallel regions are inner loops and the tuner serializes them to avoid parallel loop overhead.

3.4 Effectiveness of Individual Optimization Techniques

In this section, we discuss the contributions of individual optimization techniques to the overall performance. Also, we study the effect of interactions among optimization techniques on performance. We select the best tuned version as a base case (All-Tuned), then we turn off one technique at a time and measure the execution time. Figure 7 shows the speedup results for this experiment. We gain the following insights from the figure.

– Among the parallelism-enabling techniques, scalar and array privatization is the most important, affecting five programs significantly. Reduction parallelization has a pronounced effect in two programs, while induction variable substitution does not show a clear impact. These results are consistent with the study of an earlier generation of parallelizers [7], which found the simpler versions of these techniques (scalar privatization and scalar reduction recognition) to also be highly effective. Other work [15] has pointed out the importance of induction variable substitution. One reason for this discrepancy is that the NPB programs are already parallel, with most induction variables already being replaced.
– Among the analysis techniques, symbolic analysis is most important, followed by inlining. It affects almost all programs in a very significant manner. Inlining is also highly effective in many programs. We have mentioned earlier the importance of tuning the inlining transformation. If applied eagerly, inlining may degrade the performance of individual code sections. This is a significant finding, as many compilers offer subroutine inlining only as a global option. Alias Analysis appears

to affect most of the benchmarks; the compiler assumes very conservative all-to-all aliases when the technique is disabled.

– The locality enhancement techniques do not show a significant effect in these measurements. As discussed further in [24], these techniques are partly *substituting*; disabling them individually does not make a significant difference.

4 Conclusions

We have provided an overview of Cetus, a source-to-source compiler infrastructure for C programs. Using the NAS Parallel Benchmarks, we have evaluated the compiler in various dimensions.

Overall, the compiler is able to gain significant speedup in four of the eight programs. Our results are also significant, as they produce similar findings to previous generations of parallelizers on previous parallel computer architectures. Previous studies also found success in approximately 50 % of science/engineering applications. Even though the underlying architectures have changed dramatically, today's autoparallelization techniques for these architectures are equally successful.

Cetus improves over the parallelizer of Intel's ICC compiler by 166 % on average. We also compared with hand-parallelized programs. In only two benchmarks, the hand-optimized version significantly outperforms the Cetus' results. In one case, the automatically parallelized program performs the best.

One aspect of parallelizers has become more critical, however. Good decisions about the application of optimization techniques need runtime information. We have presented several techniques that help with dynamic decision support. Cetus includes an automatic tuning capability, which improves by 156 % over eager compile-time parallelization, and by 71 % over a compiler-inserted profitability test, on average.

The evaluation of individual parallelization techniques showed that advanced versions of those techniques found to be important in previous generations of compilers are also among the most important optimizations in Cetus.

The development of Cetus has primarily focused on automatic parallelization of C programs; the experiments in this paper investigated the performance of Cetus with regard to parallel optimization. As Cetus represents a general source-to-source compiler infrastructure, its suitability for other compilation tasks is being evaluated in ongoing work. Among the current evaluation projects are also the study of the sensitivity to diverse environment and architecture parameters, the importance of which has been demonstrated [25].

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

References

1. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufman, San Francisco (2002)

2. Asenjo, R., Castillo, R., Corbera, F., Navarro, A., Tineo, A., Zapata, E.: Parallelizing irregular C codes assisted by interprocedural shape analysis. In: 22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08) (2008)
3. Baek, W., Minh, C.C., Trautmann, M., Kozyrakis, C., Olukotun, K.: The opentm transactional application programming interface. In: PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques, pp. 376–387. IEEE Computer Society, Washington, DC, USA (2007). doi:10.1109/PACT.2007.74
4. Barszcz, E., Barton, J., Dagum, L., Frederickson, P., Lasinski, T., Schreiber, R., Venkatakrishnan, V., Weeratunga, S., Bailey, D., Bailey, D., Browning, D., Browning, D., Carter, R., Carter, R., Fineberg, S., Fineberg, S., Simon, H., Simon, H.: The NAS parallel benchmarks. Int. J. Supercomput. Appl. Technical report (1991)
5. Basumallik, A., Eigenmann, R.: Optimizing irregular shared-memory applications for distributed-memory systems. In: PPoPP '06: Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 119–128. ACM, New York, NY, USA (2006). doi:10.1145/1122971.1122990
6. Blume, W., 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. IEEE Computer **29**(12), 78–82 (1996)
7. Blume, W., Eigenmann, R.: Performance analysis of parallelizing compilers on the perfect benchmarks programs. IEEE Trans. Parallel Distrib. Syst. **3**(1), 643–656 (1992)
8. Blume, W., Eigenmann, R.: The range test: a dependence test for symbolic, non-linear expressions. In: Proceedings of Supercomputing '94, Washington, DC, pp. 528–537 (1994)
9. Callahan, D., Dongarra, J., Levine D.: Vectorizing compilers: a test suite and results. In: Proceedings of the 1988 ACE/IEEE Conference on Supercomputing, Orlando, FL, USA, pp. 98–105. IEEE Computer Society Press, Los Alamitos, CA (1988)
10. Callahan, D.: The program summary graph and flow-sensitive interprocedual data flow analysis. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language design and Implementation, PLDI '88, pp. 47–56. ACM, New York, NY, USA (1988). doi:10.1145/53990.53995
11. Christen, M., Schenk, O., Burkhart, H.: PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2011 (2011)
12. Dave, C.: Parallelization and performance-tuning: automating two essential techniques in the multicore era. Master's thesis, Purdue University (2010)
13. 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)
14. Eigenmann, R., Blume, W.: An effectiveness study of parallelizing compiler techniques. In: Proceedings of the International Conference on Parallel Processing, vol. 2, pp. 17–25 (1991)
15. Eigenmann, R., Hoeflinger, J., Padua, D.: On the automatic parallelization of the perfect benchmarks. IEEE Trans. Parallel Distrib. Syst. **9**(1), 5–23 (1998)
16. Emami, M., Ghiya, R., Hendren, L.J.: Context-sensitive interprocedural points-to analysis in the presence of function pointers. In: Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation, PLDI '94, pp. 242–256. ACM, New York, NY, USA (1994). doi:10.1145/178243.178264
17. Fei, L., Midkiff, S.P.: Artemis: practical runtime monitoring of applications for execution anomalies. In: PLDI '06: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 84–95. ACM, New York, NY, USA (2006). doi:10.1145/1133981.1133992
18. Guo, J., Stiles, M., Yi, Q., Psarris, K.: Enhancing the role of inlining in effective interprocedural parallelization. In: Parallel Processing (ICPP), 2011 International Conference on, pp. 265–274 (2011). doi:10.1109/ICPP.2011.68
19. 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)
20. Lee, S., Min, S.J., Eigenmann, R.: OpenMP to GPGPU: a compiler framework for automatic translation and optimization. In: Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'09), ACM Press (2009)
21. Liu, Y., Zhang, E.Z., Shen, X.: A cross-input adaptive framework for GPU program optimizations. In: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing,

pp. 1–10. IEEE Computer Society, Washington, DC, USA (2009) doi:10.1109/IPDPS.2009.5160988. http://portal.acm.org/citation.cfm?id=1586640.1587597
22. Min, S.J., Kim, S.W., Voss, M., Lee, S.I., Eigenmann, R.: Portable compilers for OpenMP. In: OpenMP Shared-Memory Parallel Programming, Lecture Notes in Computer Science #2104, pp. 11–19. Springer, Heidelberg (2001)
23. Mustafa, D., Eigenmann, R.: Portable section-level tuning of compiler parallelized applications. In: Proceedings of the 2012 ACM/IEEE Conference on Supercomputing. IEEE Press (2012)
24. Mustafa, D., Eigenmann, R.: Window-based empirical tuning of parallelized applications. Technical report, Purdue University, ParaMount Research Group (2011)
25. Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.: The effect of omitted-variable bias on the evaluation of compiler optimizations. Computer **43**(9), 62–67 (2010). doi:10.1109/MC.2010.214
26. 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)
27. Papakonstantinou, A., Gururaj, K., Stratton, J.A., Chen, D., Cong, J., Hwu, W.M.W.: High-performance CUDA kernel execution on FPGAs. In: Proceedings of the 23rd International Conference on Supercomputing, ICS '09, pp. 515–516. ACM, New York, NY, USA (2009). doi:10.1145/1542275.1542357
28. Satoh, S.: NAS Parallel Benchmarks 2.3 OpenMP C version [Online]. Available: http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp (2000)
29. Shen, Z., Li, Z., Yew, P.: An empirical study of Fortran programs for parallelizing compilers. IEEE Trans. Parallel Distrib. Syst. **1**(3), 356–364 (1990)
30. Tu, P., Padua, D.: Automatic array privatization. In: Banerjee, U., Gelernter, D., Nicolau, A., Padua D. (eds.) Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 768, pp. 500–521, Portland (12–14 August 1993)
31. der Wijngaart, R.F.V.: NAS parallel benchmarks version 2.4. Technical report, Computer Sciences Corporation, NASA Advanced Supercomputing (NAS) Division (2002)
32. Wolfe, M.: Optimizing Supercompilers for Supercomputers. MIT Press, Cambridge (1989)
33. Yang, Y., Xiang, P., Kong, J., Zhou, H.: A GPGPU compiler for memory optimization and parallelism management. In: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pp. 86–97. ACM, New York, NY, USA (2010). doi:10.1145/1806596.1806606
34. Yang, Y., Xiang, P., Kong, J., Zhou, H.: An optimizing compiler for GPGPU programs with input-data sharing. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, pp. 343–344. ACM, New York, NY, USA (2010). doi:10.1145/1693453.1693505