CETUS: A SOURCE-TO-SOURCE COMPILER INFRASTRUCTURE FOR MULTICORES

Reuse Rights and Reprint Permissions:
Educational or personal use of this material is permitted without fee, provided such use: 1) is not made for profit; 2) includes this notice and a full citation to the original work on the first page of the copy; and 3) does not imply IEEE endorsement of any third-party products or services. Authors and their companies are permitted to post their IEEE-copyrighted material on their own Web servers without permission, provided that the IEEE copyright notice and a full citation to the original work appear on the first screen of the posted copy.

Permission to reprint/republish this material for commercial, advertising, or promotional purposes or for creating new collective works for resale or redistribution must be obtained from IEEE by writing to the IEEE Intellectual Property Rights Office, 445 Hoes Lane, Piscataway, NJ 08854-4141 or pubs-permissions@ieee.org. Copyright © 2009 IEEE. All rights reserved.

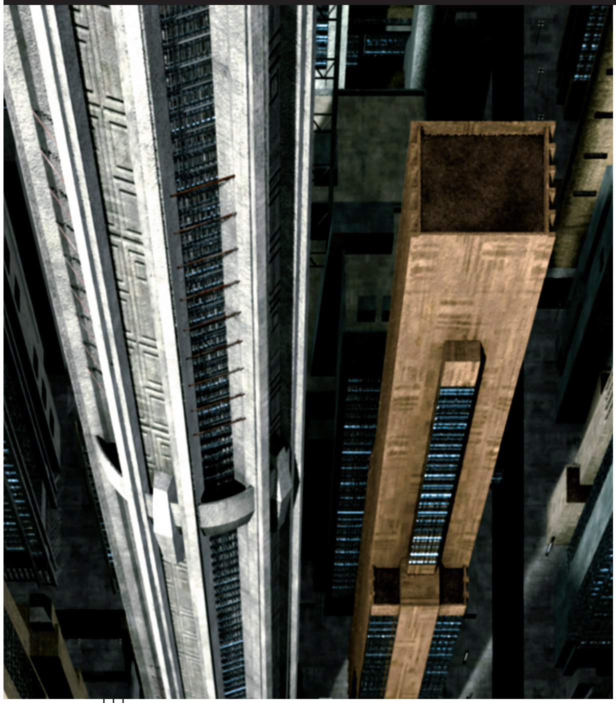
Abstracting and Library Use:
Abstracting is permitted with credit to the source. Libraries are permitted to photocopy for private use of patrons, provided the per-copy fee indicated in the code at the bottom of the first page is paid through the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923.

Citation:
Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, Samuel Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer*, vol. 42, no. 12, pp. 36-42, Dec. 2009, doi:10.1109/MC.2009.385

For more information, please visit:
http://cetus.ecn.purdue.edu

Contact us at:
cetus@ecn.purdue.edu

The Cetus Team



CETUS: A SOURCE-TO-SOURCE COMPILER INFRASTRUCTURE FOR MULTICORES

Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff,
Purdue University

The Cetus tool provides an infrastructure for research on multicore compiler optimizations that emphasizes automatic parallelization. The compiler infrastructure, which targets C programs, supports source-to-source transformations, is user-oriented and easy to handle, and provides the most important parallelization passes as well as the underlying enabling techniques.

With the advent of multicore architectures, automatic parallelization has, for several reasons, re-emerged as an important tool technology. While classical parallel machines served a relatively small user community, multicores aim to capture a mass market, which demands user-oriented, high-productivity programming tools. Further, multicores are replacing complex superscalar processors, the parallelism of which was unquestionably exploited by the compiler and underlying architecture.

This same model is desirable for the new generation of CPUs: Automatic parallelization had its successes on shared-address-space architectures exhibiting small numbers of processors, which is the very structure of today's multicores.

This state-of-the-art snapshot of automatic parallelization for multicores uses the Cetus tool. Cetus is an infrastructure for research on multicore compiler optimizations, with an emphasis on automatic parallelization. We have created a compiler infrastructure that supports source-to-source transformations, is user-oriented and easy to handle, and provides the most important parallelization passes as well as the underlying enabling techniques. The infrastructure project follows Polaris,[1,2] which was arguably the most advanced research infrastructure for optimizing compilers on parallel machines. While Polaris translated Fortran, Cetus targets C programs. Cetus arose initially from the work of several enthusiastic graduate students, who continued what they began in a class project. Recently, we have obtained funding from the US National Science Foundation to evolve the project into a community resource.

In our work, we have measured both Cetus and Cetus-parallelized program characteristics. These results show a high-quality parallelization infrastructure equally powerful but easier to use than other choices. Cetus can be compared to Intel's ICC compiler and the COINS research compiler. Initially we had also considered infrastructures such as SUIF (suif.stanford.edu), Open64 (www.open64.net), Rose (rosecompiler.org), the Gnu C compiler, Pluto (pluto-compiler.sourceforge.net), and the Portland Group (PGI) C Compiler. However, parallelization results for these tools were either unavailable or lacked explanation.

Published by the IEEE Computer Society

0018-9162/09/$26.00 © 2009 IEEE

```
class Instrumenter
{
    ...
    public void instrumentLoops(Program p)
    {
       DepthFirstIterator iter = new DepthFirstIterator(p);
       int loop_number = 0;
       while ( iter.hasNext() ) {
          Object obj = iter.next();
          if ( obj instanceof ForLoop )
            insertTimingCalls((ForLoop)obj, loop_number++);
       }
    }

    private void insertTimingCalls(ForLoop loop, int number)
    {
       FunctionCall tic, toc;
       tic = new FunctionCall(new Identifier("cetus_tic"));
       toc = new FunctionCall(new Identifier("cetus_toc"));
       tic.addArgument(new IntegerLiteral(number));
       toc.addArgument(new IntegerLiteral(number));
       CompoundStatement parent = (CompoundStatement)loop.getParent();
       parent.addStatementBefore(loop, new ExpressionStatement(tic));
       parent.addStatementAfter(loop, new ExpressionStatement(toc));

    }
   ...
}
```

Figure 1. Implementation example for basic loop instrumentation. The instrumenter inserts timing calls around each `for` loop in the given input program.

Developing a dependable community support system and reaching out to Cetus users is one of our goals. Cetus maintains a community portal at cetus.ecn.purdue.edu, where the compiler can be downloaded under an artistic license. The portal offers a utility for submitting bug reports and feature requests, and a Cetus users' mailing list to discuss ideas, new functionality, research topics, and user concerns. The Cetus portal further provides documentation for installing and running the compiler and for writing new analysis and transformation passes using the internal program representation (IR) interface.

Several US and worldwide research groups already use Cetus.[3-5] In our ongoing work, we apply the infrastructure for creating translators that convert shared-memory programs written in OpenMP to other models, such as message-passing[6] and CUDA (for graphics processing units).[7]

CETUS ORGANIZATION AND INTERNAL REPRESENTATION

Cetus's 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 Translation Unit. Other base IR object types are Statement, Declaration, and Expression. Specific source constructs in the IR are represented by classes derived from these base classes. For example, 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. There is complete data abstraction, and pass writers only manipulate the IR through access functions. Important features of the IR include the following:

- *Traversable objects*. All Cetus IR objects are derived from a base class "Traversable." This class provides the functionality to iterate over lists of objects generically.
- *Iterators*. BreadthFirst, DepthFirst, and Flat iterators are built into the functionality to provide easy traversal and search over the program IR.
- *Symbol table*. Cetus's symbol table functionality 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 types of auxiliary information about IR objects can be stored in annotation objects, which take the form of declarations. An annotation can be associated with a statement (such as information about an OpenMP directive belonging to a `for` statement) or could stand independently (such as a comment line).
- *Printing*. The printing functions have been extended to allow for flexible rendering of the IR classes.

CETUS ANALYSES AND TRANSFORMATIONS

Automatically instrumenting source programs is a commonly used compiler capability. Figure 1 shows a basic loop instrumenter that inserts timing calls around each `for` loop in the given input program. Here we assume that the function call `cetus_tic(id)` stores the current time (`get time of day`, for example) with the given integer tag, and `cetus_toc(id)` computes and stores the time difference since the last call of `cetus_tic(id)`.

At the end of an application run, basic runtime statistics on the instrumented code sections can be generated from this information. The Cetus code in Figure 1 assigns a

```
int foo(void)
{
  int i;
  double t, s, a[100];
  for ( i = 0; i < 50; ++i )
  {
    t = a[i];
    a[i+50] = t + (a[i] + a[i+50])/2.0;
    s = s + 2*a[i];
  }
  return 0;
}
```
(a)

```
int foo()
{
  int i;
  double t, s, a[100];
  cetus_tic(0);
  for (i = 0; i < 50; ++i)
  {
    t = a[i];
    a[(i+50)] = (t + ((a[i] + a [(i+50)])/2.0));
    s = (s + (2*a[i]));
  }
  cetus_toc(0);
  return 0;
}
```
(b)

Figure 2. (a) Input source code and (b) the output of loop instrumentation; the instrumenter inserted the `cetus_tic(0)` and `cetus_toc(0)`, with an integer tag "0" assigned to the loop.

unique integer tag for each loop and inserts the two timing calls around the loops. The `instrumentLoop()` method drives this transformation by iterating over the entire program in depth-first order, by checking the type of the current IR object and invoking the actual insertion method `insertTimingCalls()`. The `insertTimingCalls()` method constructs the two function calls described, accesses the compound statement enclosing the current loop, and then inserts the two function calls.

Figure 2b shows the output obtained from Cetus using the loop instrumenter for the loop in the input program shown in Figure 2a. The instrumenter inserts the begin-timing function call `cetus_tic(loopid)` before the start of the loop, and an end-timing function call `cetus_toc(loopid)` after the loop.

In the case of multiple loops, each loop at every nest level would be instrumented with profiling calls with a unique loopid for each loop. The Cetus implementation described in Figure 1 requires only about 25 lines of Java code. The abstractions of the Cetus IR hierarchy for searching, creating, manipulating, and removing IR objects make this possible. The user can easily traverse the IR using Cetus-provided iterators, including the functionality to search objects of specific types such as Loops, FunctionCalls, BinaryExpressions, and others. IR objects provide a rich interface for the user to think in terms of source-level constructs such as "adding arguments" to a FunctionCall or "adding an additional Statement" to an existing CompoundStatement in the IR.

Analysis passes

Advanced program analysis capabilities are essential to Cetus; they will grow both through our own efforts and the Cetus user community's. Here we describe some basic analyses.

Symbolic manipulation. Like its predecessor, Polaris,[1,2] Cetus provides a key feature in its ability to analyze the represented program in symbolic terms. 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 3 shows these tools' capabilities.

Array section analysis. Array sections describe the set of array elements accessed by program statements. Compiler passes equipped with array section analysis are more accurate than others that use a name-based approach. Cetus's array-section analysis pass performs a may-use/may-def analysis of array variables for a given input program by computing the value ranges of the array subscripts. These array sections are merged together if the analysis is applied to code sections with multiple statements. The following code example shows the result of array section analysis, expressed as a pragma annotation, for the given loop:

```
c = 2;
N = 100;
#pragma cetus USE(A[0:100][0:100])
      DEF(B[1:99][1:99])
for (i=1; i<N; i++) {
  for (j=1; j<N; j++) {
    B[c*i -i][j] = (A[i-1][j]+A[i+1]
        [j]+A[i][j-1]+A[i][j+1])/4;
  }
}
```

Data dependence analysis. This analysis provides a memory disambiguation technique that seeks to identify data references that access the same memory location during program execution and that characterizes dependencies between those references. Array data-dependence analysis involves the process of analyzing array subscripts 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 dependencies 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)`. We use advanced symbolic range analysis to simplify loop-related information and array subscripts 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 dependencies.[8,9] Other tests are under development.

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

Range analysis. This technique computes integer variables' value ranges at each program point and returns a map from each statement to a set of value ranges valid before each statement. This repository of ranges, together with utility functions, provides other passes with knowledge about symbolic terms, including the bounds of variables and expressions, and determines through symbolic comparison if one expression is semantically greater than another. For example, our range analysis framework can conclude that `v1 + v2 ≥ v3` at a program point that has a set of value ranges `{v1 = [0, 10], v2 = v1 -5, v3 = -5}`, since the value range of the expression `v1 + v2 is [-5, 5]`. We use this range analysis framework in many applications—including array section analysis, array privatization, induction variable substitution, and data-dependence analysis—to improve their accuracy.

Parallelizing transformation passes

The basic parallelizing transformation techniques Cetus currently implements are privatization, reduction variable recognition, and induction variable substitution. These are the techniques found to be most important for automatically parallelizing compilers.[10,11] In ongoing work, we are developing techniques that can enhance these transformations further, including interprocedural analysis and advanced alias analysis.

Privatization. Identifying private variables in a loop is an important step automatic parallelizers must perform. A private variable serves as a temporary variable in a loop, one that is written first and used later in the loop. Array sections provide temporary locations for private array variables. These variables do not need to be exposed to the other threads at runtime, so the data-dependence analyzer can safely assume these variables do not have dependencies.

We implemented a simple but effective array privatizer in Cetus, which can handle array sections containing symbolic terms. 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 or scalar variables.

Next, the array privatizer identifies private variables by checking if there are no upward-exposed uses for the variables. The privatizer's accuracy improves when using a symbolic analysis technique such as range analysis to perform array section operations. For example, the privatizer always seeks to compute large must-defined sections to minimize upward-exposed uses, and the intersection of the two must-defined sections, `[1 : m] ∩ [1 : n]`, results in `[1 : n]` rather than `[1 : min(m, n)]` if the expression comparison tool can decide `n ≤ m`.

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

- 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; and
- `rv` appears nowhere else in the loop.

Induction variable substitution. The third parallelization transformation technique is induction variable recognition and substitution. An induction statement has a form, `iv = iv + expr`, similar to a reduction statement, and must be replaced by another form that does not induce data dependence. If the right-hand side in the preceding form can be expressed as a closed-form expression that does

```
1 + 2*a + 4 - a       ⇒  5 + a            (folding)
a*(b + c)             ⇒  a*b + a*c        (distribution)
 (a*2) / (8*c)        ⇒  a / (4*c)        (division)
 (1 - a) < (b + 2)    ⇒  (1 + a + b) > 0  (normalization)
a && 0 && b           ⇒  0                (short-circuit evaluation)
```

Figure 3. Cetus symbolic expression tools. The ability to manipulate symbolic expressions is essential when designing analysis and transformation algorithms that deal with real programs.

Table 1. Statistics on loop parallelization with Cetus running on HotSpot VM and 2.33 GHz Xeon.

Benchmarks	Lines	Memory usage (Mbytes)	Runtime (secs)	Throughput (lines/sec)
BT	3,766	142	65.13	58
CG	985	103	4.52	218
EP	326	80	1.96	166
FT	1,319	108	7.12	185
IS	766	78	1.77	433
LU	3,666	144	31.87	115
MG	1,366	99	8.80	155
SP	3,110	145	22.34	139
equake	1,590	105	6.98	228
art	1,977	92	5.06	391
ammp	13,501	196	51.85	260

not contain iv, the dependence in the preceding statement will be removed.

Cetus has an induction variable recognition and substitution pass that can detect and substitute variables such as `iv` when `expr` is either loop-invariant or another induction variable. This pass visits every statement in a loop nest and symbolically computes the increments of induction variables at each statement following entry to the loop nest. It then adds the increments to every use of the induction variables while removing the induction statements.

We also use symbolic analysis to avoid possibly unsafe transformation because of symbolic loop bounds. For example, the following transformation is not safe, because the increment of the variable `k` after the inner loop is not `2*n + 2*i*n` when `n < 0`:

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

EVALUATION

We take two approaches to evaluating Cetus: presenting characteristics of Cetus itself, and discussing automatic parallelization's state of the art while comparing Cetus with two other parallelizers.

Cetus characteristics

Cetus provides the preceding analyses and transformations, as well as a highly programmable IR, with the goal of improving usability, productivity, and extensibility for the community of Cetus users. The compiler successfully translates and validates 13 out of 14 SPEC CPU2006 benchmarks (456.hmmer currently does not validate because of an unsupported usage of both the K&R C and ANSI C formats for function declarations). Automatic parallelization, as implemented using the transformations previously described, successfully generates OpenMP parallel code for all C programs in the SPEC OMP 2001 and NAS Parallel Benchmark suites.

Although Cetus represents a high-quality parallelizer, its IR implementation consists of only 17,000 lines of Java code, while the parallelization passes, including analyses and transformations, consist of only about 5,000 lines of Java code. Cetus, in total, consists of about 45,000 lines of Java code, including the IR and the parallelization framework. These metrics underline the advantage of Cetus as a compact and easy-to-use infrastructure. It can handle real applications and conveniently facilitates the creation of new analysis and transformation passes. Table 1 describes the performance of Cetus as an autoparallelization framework.

Runtime consists mainly of parallelization time, of which the dependence analyzer consumes a major portion. BT performs relatively poorly in terms of throughput because of a significant number of loops that are both large and deeply nested. These properties add complexity for the dependence analyzer. IS achieves high throughput because of many singly nested loops that undergo dependence testing. Cetus's performance remains relatively consistent for the remaining benchmarks.

Although Cetus requires a longer compilation time than the industrial compiler, this time is offset by—and partly the result of—the programmability Cetus offers. Cetus's memory usage is driven primarily by the complexity of loops analyzed for dependence testing, in terms of their nesting levels and the total number of array accesses they contain. While the memory footprint is well within the resource limits of current computer systems, it could be improved through a more efficient implementation of dependence information storage, especially with regard to the data-dependence graph.

Automatic parallelization

Cetus enables automatic parallelization by using data dependence analysis with the Banerjee-Wolfe inequalities, array and scalar privatization, reduction-variable recognition, and induction-variable substitution. Several compilers implement automatic parallelization using different analysis techniques, including a variety of dependence analyzers and loop-transformation techniques such as loop distribution and loop interchange.[8,9] We sought to compare Cetus's performance with compilers that provide enough parallelization information to merit a fair comparison. The Intel C Compiler (icc) provides source-level information related to parallelism detection, which let us gather comparison data. COINS (www.coins-project.org/international) is an infrastructure similar to Cetus; coded in Java, it provides source-to-source translation along with automatic parallelization.

Comparing the number of parallelized loops between Cetus, icc, and COINS, as shown in Figure 4, reveals that Cetus performs close to or better than icc on 7 of 11 benchmarks, and better than COINS on 10 benchmarks. In the five benchmarks where Cetus performs poorly, the deficit in number of parallel loops ranges from 10 to 40 percent.

In LU's case, Cetus generates fewer parallel loops because it exploits more outer-level parallelism, while icc parallelizes inner loops without parallelizing these outer loops, thus increasing the number of parallel loops but decreasing parallel-loop granularity. Effectively, compared to icc Cetus is closer to hand-parallelized code. In the case of ammp, Cetus performs poorly, mainly because of the function calls within the loops and the absence of interprocedural analyses within our current framework. These results emphasize the Cetus framework's scope as we achieve close to state-of-the-art parallelization using four parallelizing transformations that span a significantly small number of code lines compared to other compiler infrastructures.[10]

The automatically parallelized loops with Cetus actually cover a substantial part of the sequential execution time as shown in Figure 5. This metric translates to the theoretical speedup of an application on an ideal parallel machine without parallel execution overhead. Cetus detects important parallel loops or their inner-loop parallelism in CG, IS, SP, and art, but fails to parallelize such loops in EP, equake, and ammp. While the results in Figure 5 do not show direct speedups obtained on actual parallel machines, they provide insights into the compilers' abilities to achieve automatic parallelization. Important architectural features that determine the ultimate multicore performance include the memory hierarchy and the ability to provide fast synchronization.

In ongoing work, we are implementing state-of-the-art locality enhancement techniques such as tiling, as well as techniques to move compile-time decisions into runtime. In addition, we expect the infrastructure to grow through community contributions such as techniques dealing with memory access and synchronization cost. Cetus provides the community with a good starting point for research into these and other important issues.

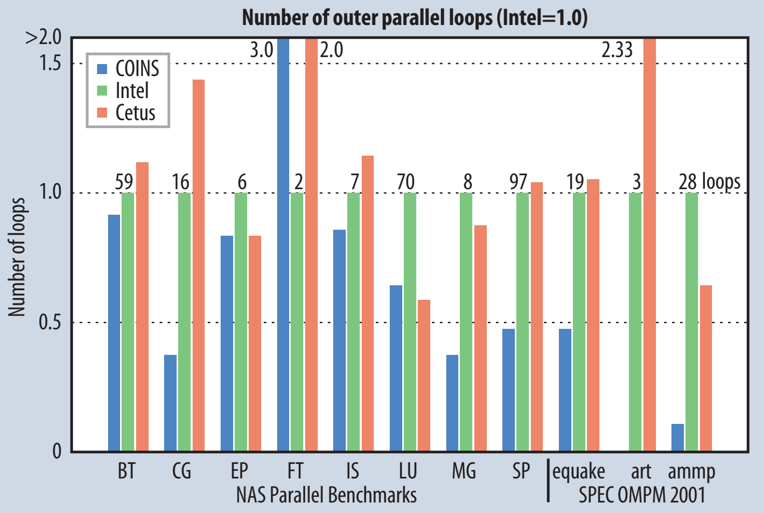


Figure 4. Compiler comparison. Cetus is as powerful as other automatic parallelizers in terms of detection of parallel loops.

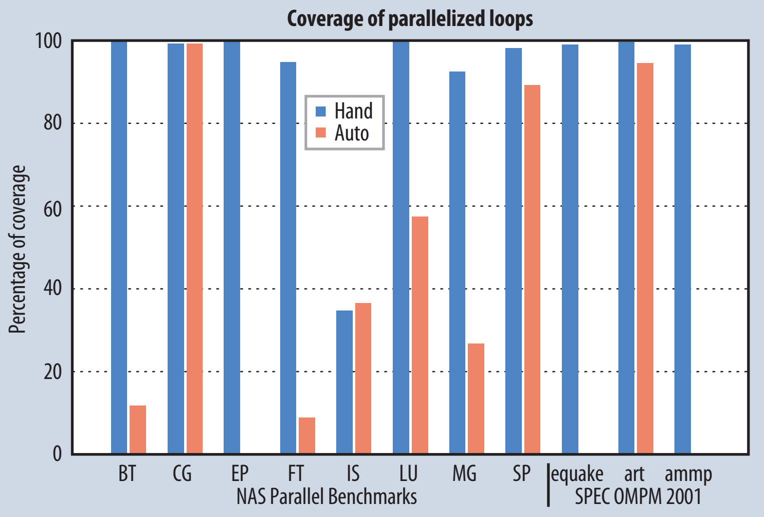


Figure 5. Parallel coverage. While these results do not show the direct speedups obtained on actual parallel machines, they provide insights into the compilers' abilities to achieve automatic parallelization.

To increase our user community, we are exploring the Cetus organization and IR interface; the analyses, transformations, and results of the infrastructure itself; and the characteristics of Cetus-parallelized programs. Cetus is being used and extended in ways beyond those described here. Notably, in ongoing work we are adding passes for improved alias and data-dependence analysis as well as additional parallelizing transformations. Other projects are extending Cetus to related languages, such as C++ and Java, as well as dialects such as C for GPUs.

Cetus has grown from a simple, student-designed source-to-source translator into a robust system supported by the National Science Foundation as a community infrastructure. Cetus is a high-quality, easy-to-use tool ready for the user community. Via the Community Portal at cetus.ecn.purdue.edu, we can respond to user requests and incorporate community-developed modules. Through these mechanisms, we expect Cetus to become a research infrastructure widely applicable to source-level optimizations and transformations for both multicore and large-scale parallel programs. C

Acknowledgment

This work was supported, in part, by the National Science Foundation under grants Nos. 0751153-CNS, 0707931-CNS, 0833115-CCF, and 0916817-CCF.

References

1. W. Blume et al., "Parallel Programming with Polaris," *Computer*, Dec. 1996, pp. 78-82.
2. S.-J. Min et al., "Portable Compilers for OpenMP," *OpenMP Shared-Memory Parallel Programming*, LNCS 2104, Springer Verlag, 2001, pp. 11-19.
3. L. Fei and S.P. Midkiff, "Artemis: Practical Runtime Monitoring of Applications for Execution Anomalies," *Proc. 2006 ACM SIGPLAN Conf. Programming Language Design and Implementation* (PLDI 06), ACM Press, 2006, pp. 84-95.
4. W. Baek et al., "The Open Transactional Application Programming Interface," *Proc. 16th Int'l Conf. Parallel Architecture and Compilation Techniques* (PACT 07), IEEE CS Press, 2007, pp. 376-387.
5. R. Asenjo et al., "Parallelizing Irregular C Codes Assisted by Interprocedural Shape Analysis," *Proc. 22nd IEEE Int'l Parallel and Distributed Processing Symp.* (IPDPS 08), IEEE Press, 2008, pp. 1-12.
6. A. Basumallik and R. Eigenmann, "Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems," *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming* (PPoPP 06), ACM Press, 2006, pp. 119-128.
7. S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," *Proc. ACM Symp. Principles and Practice of Parallel Programming* (PPoPP 09), ACM Press, Feb. 2009, pp. 101-110.
8. R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures*, Morgan Kaufmann, 2002.
9. M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989.
10. R. Eigenmann, J. Hoeflinger, and D. Padua, "On the Automatic Parallelization of the Perfect Benchmarks," *IEEE Trans. Parallel Distributed Systems*, vol. 9, no. 1, 1998, pp. 5-23.
11. H. Bae et al., "Automatic Parallelization with Cetus," tech. report ECE-HPCLab-08202, Purdue Univ., School of Electrical and Computer Engineering, High-Performance Computing Laboratory, 2008.

Chirag Dave *is a PhD student in the School of Electrical and Computer Engineering at Purdue University. His research interests include parallel programming, automatic performance tuning, and optimizing compilers. Dave received a BEng in communications and electronic engineering from the University of Leicester, UK. Contact him at cdave@purdue.edu.*

Hansang Bae *is a PhD student in the School of Electrical and Computer Engineering at Purdue University. His research interests include optimizing compilers, program analysis, and high-performance computing. Bae received an MS in electrical and computer engineering from Purdue University. Contact him at baeh@purdue.edu.*

Seung-Jai Min *formerly at Purdue University, where he received a PhD in electrical and computer engineering, is a postdoctoral researcher in the Future Technologies Group at Lawrence Berkeley National Laboratory. His research interests include optimizing compilers, parallel programming, and high-performance computing. Min received a PhD in electrical and computer engineering from Purdue University. Contact him at sjmin@lbl.gov.*

Seyong Lee *is a PhD student in the School of Electrical and Computer Engineering at Purdue University. His research interests include parallel programming and performance optimization in heterogeneous computing environments, program analysis, and optimizing compilers. Lee received an MS in electrical and computer engineering from Purdue University. Contact him at lee222@purdue.edu.*

Rudolf Eigenmann *is a professor in the School of Electrical and Computer Engineering at Purdue University. His research interests include optimizing compilers, programming models for parallel computing, and cyberinfrastructures. Eigenmann received a PhD in electrical engineering/computer science from ETH Zurich, Switzerland. Contact him at eigenman@purdue.edu.*

Samuel Midkiff *is a professor in the School of Electrical and Computer Engineering at Purdue University. His research interests include abstractions for parallelism, debugging of parallel programs, and memory models. Midkiff received a PhD in computer science from the University of Illinois at Urbana-Champaign. Contact him at smidkiff@purdue.edu.*