Cetus: A Source-to-Source Compiler Infrastructure for Multicores

Hansang Bae, Leonardo Bachega, Chirag Dave, Sang-Ik Lee, Seyong Lee, Seung-Jai Min, Rudolf Eigenmann and Samuel Midkiff

Purdue University[⋆]

Abstract. We describe the Cetus compiler infrastructure and its use in a number of transformation tasks for multicore architectures. The original intent of Cetus was to serve as a parallelizing compiler. In addition, the infrastructure has been used to build translators for programs written in the OpenMP directive language to be compiled onto multicore architectures. They include a direct OpenMP translator for current multicores, an OpenMP to MPI translator for many-cores exhibiting disjoint address spaces, and a translator for OpenMP onto GPU architectures. We are also building autotuning capabilities into Cetus, which can defer compile-time optimization decisions to runtime. This feature is especially important for heterogeneous multicore architectures. We will describe the organization of the Cetus infrastructure and present preliminary results of several application projects.

1 Introduction

Cetus is a source-to-source restructuring compiler infrastructure for C programs, and is a follow-on project to the Polaris Fortran translator [1, 2]. The driving motivation was the need for a source-level compiler infrastructure that facilitates advanced optimizations for parallel programs written in C. Cetus has already been used for a number of applications, several of which we will describe.

Cetus had originally been created in a Purdue advanced compiler class project and has evolved into a fairly robust infrastructure. At first, the manpower behind the Cetus development was all "volunteer work" by several dedicated graduate students. Recently, the project has obtained funding from the U.S. National Science Foundation to become a community resource. This grant allows us to improve the robustness of Cetus, respond to user requests, and add new features.

Cetus is already in use by a number of research groups in the U.S. and worldwide [3–6]. Increasing the user community is one goal of this paper. To this end, this paper describes the Cetus Community Portal in Section 2, the Cetus internal organization in Section 3, current analysis and transformation capabilities in Sections 4 and 5, respectively, and applications projects in Section 6.

[⋆] This work was supported, in part, by the National Science Foundation under grants No. 0429535-CCF, 0650016-CNS, CNS-0751153, and CNS-0707931.

Cetus is being used and extended in ways beyond those described in this paper. Notably, in ongoing work we are adding passes for improved alias and data-dependence analysis as well as additional parallelizing transformations. Other projects extend Cetus to related languages, such as C++ and Java, and dialects, such as C for GPUs.

2 Cetus Community Portal

Developing a dependable community support system and reaching out to Cetus users is one of our primary goals. As we continue to build more functionality, it is crucial to be able to cater to the needs of our users and influence our development with their feedback. The Community Portal at http://cetus.ecn.purdue.edu serves that purpose. The Cetus infrastructure is available for download at this portal.

2.1 Documentation

The website provides easy access to documentation in order to assist with the process of installing and running the Cetus compiler. This documentation includes the Cetus Compiler manual, which provides sections on the Cetus architecture as well as important information to help pass writers with incorporating new analysis and transformation passes using the existing Cetus API. Sections on existing analysis and transformation passes are under development. The Cetus API is made available in Javadoc format via the website as well.

2.2 Download

The latest version can be downloaded at the same website. Cetus downloads are maintained under the Artistic License, details of which can be found on the website.

2.3 Community Support

We have now incorporated a strong user feedback system within the Cetus website to additionally help us with answering questions from users and dealing with the issues they encounter as they continue to use our framework to meet their needs.

Bugzilla: The website supports Bugzilla v3.0.1. This is a utility that allows users to submit bugs reports. Users can also inform the Cetus team about issues they have come across while using the compiler, to see the progress related to bugs submitted and to receive feedback when the bugs have been fixed. This allows us to interact with our customer base while developing the infrastructure with the right needs in mind and to involve users in the development process of Cetus.

Mailing List: A Cetus-users mailing list can now be used to discuss ideas, new functionality and research related to Cetus that would benefit the entire research community as well as the users of Cetus. This can be considered as a robust means of information exchange that would help incorporate new ideas into Cetus.

3 Cetus Organization and Internal Representation

3.1 Cetus Class Hierarchy

Cetus' internal program representation (IR) is implemented in the form of a Java class hierarchy. There is complete data abstraction, with pass writers only manipulating the IR through access functions. An early version of this class hierarchy was described in [7]. We briefly mention a few changes with respect to this version.

– *Symbol table:* Cetus' symbol table functionality provides information about identifiers and data types. Its implementation makes direct use of the information stored in declaration statements stored in the IR. There is no separate and redundant symbol table storage.
– *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.
– *Annotations:* Comments, pragmas, directives, and other types of auxiliary information about IR objects can be stored in *annotation objects*. They take the form of declarations. An annotation may be associated with a statement (e.g., info about an OpenMP directive belonging to a `for` statement) or may stand independently (e.g., a comment line).
– *Printing:* The printing functions have been extended to allow for flexible rendering of the IR classes.

3.2 Symbolic Manipulation

Like its predecessor Polaris [1, 2], a key feature of Cetus is the ability to reason about the represented program in symbolic terms. For example, compiler analyses and optimizations at the source level often require the expressions in the program to be in a simplified form. A specific example is data dependence analysis that collects the coefficients of affine subscript expressions, which are passed to the underlying data dependence test package. Cetus has functionalities that ease the manipulation of expressions for pass writers. These utilities simplify and normalize a given expression. The following examples show the features of the simplification utility. In addition, Cetus provides basic arithmetical operations (add, multiply, subtract, and divide) between symbolic expressions.

While most current Cetus passes make use of these capabilities, they are key for the Symbolic Range Analysis Pass, described in Section 4.4.

`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)

Fig. 1. Symbolic simplification/normalization examples.

4 Program Analysis Capabilities

Advanced program analysis capabilities are essential to Cetus; they will grow through our ongoing efforts and by incorporating passes developed by the Cetus user community. Here we are describing some basic analyses, including a data flow framework, array section analysis, symbolic range analysis, and data dependence analysis.

4.1 Data Flow Analysis

Table 1. Equations used in Data Flow Analysis

	Forward Flow	Backward Flow
Any Path	$Out(b) = Gen(b) \cup (In(b) - Killed(b))$ $In(b) = \cup\ Out(i)$ where $i \in Pred(b)$	$In(b) = Gen(b) \cup (Out(b) - Killed(b))$ $Out(b) = \cup\ In(i)$ where $i \in Succ(b)$
All Path	$Out(b) = Gen(b) \cup (In(b) - Killed(b))$ $In(b) = \cap\ Out(i)$ where $i \in Pred(b)$	$In(b) = Gen(b) \cup (Out(b) - Killed(b))$ $Out(b) = \cap\ In(i)$ where $i \in Succ(b)$

Cetus provides a template for implementing data flow analysis. A pass writer only needs to specify the kind of data flow analysis (forward/backward and all-path/any-path) and provide two methods, which compute the `Gen` set and `Killed` set depending on a specific data flow problem.

Figure 2 shows how reaching definition analysis is implemented in Cetus, using a worklist algorithm. In the figure, both `is_forward` and `is_allpath` are set to *true* because reaching definition problem is a forward allpath problem. The first node is added in the worklist `work` and the while-loop is executed. Inside the while-loop, a node is chosen from the sorted control flow graph to visit. Based on the value of the `is_forward` and `is_allpath` variables, `computeDataFlowEquation` will solve one of the four dataflow equations described in Table 1. `computeDataFlowEquation()` computes the `In` and `Out` sets for that given node. `Out` set is monitored to determine if any node has to be added to the worklist. When the worklist `work` is empty, the loop terminates.

4.2 Array Section Analysis

Array sections describe the set of array elements that are accessed by program statements. Array section analysis enables the Cetus passes to deal with sub-

```
public class reachingDefinition extends DataFlowPass
{
   public ControlFlowGraph cfg;
   public reachingDefinition(Traversable root) {
      super(root);
      cfg = new ControlFlowGraph(root);
   }
   public void solveRechingDefinition() {
      Traversable cur_node;
      boolean is_forward = true;    /* true: forward, false: backward */
      boolean is_allpath = true;    /* true: allpath, false: anypath */
      Hashset<Traversable> In ;
      Hashset<Traversable> Out ;

      /* sort cfg in reverse post-order for forward, post-order for backward */
      cfg.sort(is_forward);
      work.enqueue(cfg.getFirst());
      while(!work.isEmpty()) {
         cur_node = work.dequeue();
         computeDataFlowEquation(cur_node, In, Out, is_forward, is_allpath);
         if( Out has changed) {
            work.enqueue(cur_node.getSuccessors());
         }
      }
   }
   public Hashset getGenSet(Traversable t) { /* Pass-writer-defined Gen function */ }
   public Hashset getKilledSet(Traversable t) { /* Pass-writer-defined Killed function */ }
}
```

Fig. 2. An example reaching definition dataflow algorithm implemented in Cetus

arrays. This capability increases the accuracy of analysis passes and enhances the efficiency of the transformation passes, compared to a name-based array analysis.

Cetus' array section analysis pass performs a conservative `use/def` analysis of array variables for a given *Traversable* input program. Before array section analysis is applied to a loop, all array subscript expressions are simplified. Array section analysis then collects and merges the obtained range information of array accesses to find the `may-use` and `may-def` set of array variables, where array variables are expressed in terms of array sections. Figure 3 shows the result of array section analysis for an example loop.

```
c = 2;
N = 100;
#pragma cetus useset(A[0:100][0:100]) defset(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;
  }
}
```

Fig. 3. Array section analysis example

4.3 Data Dependence Analysis

The Cetus data dependence analysis framework gathers dependence information for array accesses within loop nests and creates a data dependence graph. The framework comprises a Data Dependence Test (DDT) Driver that acts as a wrapper around conventional array subscript dependence tests, such as the Banerjee test and the GCD test.

Information Collection and Storage The driver iterates over all the loops in the High-level Intermediate Representation (HIR) and identifies eligible loops. Currently, we define eligible loops as perfect loop nests and loops in the form `for(i=0;i<n;i++)`, where `n` is a known integer value. Loop information (loop bounds, loop step and enclosing loops) and array access-related information (array references, enclosing loops and parent statements) is collected in *hashmap* data structures and forwarded as input to the dependence test interface described below.

Running The Dependence Tests The data dependence test tries to disprove dependence between a pair of array accesses and, if unable to do so, it returns a dependence vector representing the directions of dependence in each dimension of the iteration space spanned by the enclosing loop nest. Our implementation of the dependence test is based on the general algorithm given in Section 3.6 of [8]. The algorithm encompasses the testing of array accesses with multiple subscripts. Affine subscripts are first split into *partitions*, followed by a sequence of independent subscript tests in each of the partitions. The result of each subscript test is a dependence vector associated with the subscript pair. At the end, the dependence vectors from all independent subscript tests are merged together in a single dependence vector. Currently, Cetus includes two subscript tests: the Banerjee test, as described in [8, 9] and the GCD test. Our data dependence framework permits easy expansion to other subscript tests, such as the Omega test [10] and the Range test [11].

DD Graph Interface Our implementation of the driver builds an edge-based data dependence graph for every loop nest based on the information returned by the dependence tests. The Graph module is designed to provide interface functions to easily query dependence information such as source, sink, and direction vectors pertaining to specific array accesses. The interface is currently under development, and we expect it to form a high-level interface that compiler pass writers would eventually use while incorporating dependence analysis into transformation and analysis passes.

4.4 Range Analysis

The goal of Range Analysis is to collect, at each program statement, a map from integer-typed scalar variables to their symbolic value ranges, represented by a

symbolic lower bound and an upper bound. In other words, a symbolic value range expresses the relationship between the variables that appear in the range. We use a similar approach as in Symbolic Range Propagation [12], with necessary adjustment for the C language, to compute the set of value ranges before each statement. The set of value ranges at each statement can be used in several ways. Pass writers can directly query the symbolic bound of a variable or can compare two symbolic expressions using the constraints given by the set of value ranges.
The high-level algorithm does fix-point iteration in two phases when propagating the value ranges throughout the program. The first phase applies widening operations at nodes that have incoming back edges to guarantee the termination of the algorithm. The second phase compensates the loss of information due to the widening operations by applying narrowing operation to the node on which widening has occurred. During the fix-point iteration, the value ranges are merged at nodes that have multiple predecessors, and outgoing value ranges are computed by symbolically executing the statement. Two typical types of program semantics that cause such changes of value ranges are constraints from conditional expressions and assignments to variables.

5 Basic Parallelizing Transformation Passes

We briefly describe the algorithms used in the current Cetus implementation for the basic three parallelizing transformation techniques: privatization, reduction variable recognition, and induction variable substitution. These are the techniques found most important in automatic parallelizing compilers [13]. The parallelization transformation algorithms described in this section can be further enhanced by the advanced analysis techniques, including inter-procedural analysis and alias analysis, which we will implement as future work.

5.1 Privatization

The high-level algorithm in Figure 4 describes the process of detecting privatizable variables, both scalars and array sections, in a loop. The set operations that appear in the algorithm are performed on the array sections if the variable is an array. We use the power of symbolic analysis techniques in Cetus to make the symbolic section operation possible. For example, $[1 : m] \cap [1 : n]$ results in $[1 : n]$ if the expression comparison tool with the value range set can decide $n \leq m$.

The algorithm traverses a loop nest from the innermost to the outermost loop. At each level, it first collects *definitions* (write references) and *uses* (read references) in the loop body. Uses that are covered by prior definitions create privatizable variables (or array sections) for the current loop. The other uses are *upward exposed*, creating read references seen in the outer loop. Second, the algorithm aggregates all these array sections over the loop iteration space, creating the array sections that are private, written and upward exposed for the

```
procedure Privatization($L$)
  Input : Loop $L$
  Output : $DEF[L], UEU[L], PRI[L]$
  // $DEF$: Defined set, $USE$: Used set
  // $UEU$: Upward-exposed USE, $PRI$: Private variables
  // $KILL$: Killed set due to modified variables in the section representation
  foreach direct inner loop $l$ in $L$
    $(DEF[l], USE[l])$ = Privatization($l$)
  // Create a CFG of the loop body with collapsed inner loops
  $G(N, E)$ = BuildCFG($L$)
  // Compute must defined set $DEF$ for each node
  Iteratively solve data flow equation $DEF$ for node $n \in N$.
    $DEF_{in}[n] = \bigcap\{DEF_{out}[p], p \in$ Predecessors of $n\}$
    $DEF_{out}[n] = (DEF_{in}[n] - KILL[n]) \cup DEF[n]$
  $DEF[L] = DEF_{out}[L_{exit}]$    // $L_{exit}$ is the exit node of $L$
  $UEU[L] = \{\}$
  // Collect defined scalars and arrays with loop-invariant sections
  $PRI[L]$ = CollectCandidates($L$)
  foreach node $n \in N$
    $UEU[L] = UEU[L] \cup (USE[n] - DEF_{in}[n])$
  foreach variable $v \in UEU[L]$
    $PRI[L] = PRI[L] - \{v\}$
  // Aggregate with respect to the iteration space
  $DEF[L]$ = AggregateDEF($DEF[L]$)
  $UEU[L]$ = AggregateUSE($UEU[L]$)
  return $(DEF[L], UEU[L])$
end
```

Fig. 4. Privatization Algorithm. This is a simplified version of the original algorithm in [14]

entire loop. The aggregation for the written sections (*DEF*) computes the must-defined regular sections of the arrays over the entire loop iterations while the aggregation of upward-exposed sections (*UEU*) requires only the conservative value ranges of the sections (may-used sections). This algorithm is a slightly simpler version of the one described in [14].

5.2 Reduction Variable Recognition

We implemented the algorithm described in [15]. Essentially, for additive reductions, the algorithm recognizes statements of the form `x = x + expr`, where `expr` is typically a real-valued, loop-variant expression. The reduction variable `x` can be a scalar or an array expression. One or several reduction statements may appear in a loop, however `x` must not appear in any non-reduction statement. The algorithm is shown in Section 5.

```
// An expression "sum" must satisfy two criteria to be a sum reduction
// - criteria 1: the loop contains one or several assignment statements of the form
//               sum=sum+expr
// - criteria 2: sum does not appear anywhere else in the loop
procedure Reduction($L$)
  Input : Loop $L$
  Side-effect : it creates reduction annotations into the IR

  // $lhse$ and $rhse$ are left-hand and right-hand side expression of $stmt$, respectively
  // findREF finds all references (name-only analysis) in a given expression or statement
  // "-" is a symbolic manipulation operator
  // $expr1.contains(expr2)$ returns true if $expr2$ exists in $expr1$

  $REDUCTION = \{\}$, $REF = \{\}$
  foreach $stmt$ in $L$
    $criterion1$ = false
    if ($stmt$ is Loop)
      call Reduction(stmt) // recursive call
    else if ($stmt$ is AssignmentStatement)
      $expr$ = $rhse$-$lhse$
      if ( !($expr.contains(lhse.basename)$) )
        $criterion1$ = true
        $REDUCTION = REDUCTION \cup lhse$
        $REF = REF \cup$ findREF($expr$)
    if (!$criterion1$)
        $REF = REF \cup$ findREF($stmt$)

  // Scalar reductions are scalar expressions (i.e., scalar variables) or arrays with
  // loop-invariant subscripts. Arrays with loop-variant subscripts are array reductions
  foreach $expr$ in $REDUCTION$
    if ( !(REF.contains($expr.basename$)) )
      if ($expr$ is ArrayAccess AND $expr.subscript$ is loop-invariant)
        CreateAnnotation(sum-reduction, ARRAY, $expr$)
      else
        CreateAnnotation(sum-reduction, SCALAR, $expr$)
end
```

Fig. 5. The above pseudo code describes an algorithm to recognize both scalar and array additive reduction variables in a loop.

5.3 Induction Variable Substitution

Figure 6 shows the induction variable recognition and substitution algorithm being developed in Cetus. The algorithm is applied to each loop nest, where it traverses three types of statements: (I) induction statements of the form `iv = iv + exp`, where `exp` is either loop-invariant or another induction variable, (U) statements using the induction variable `iv`, and (L) inner loops using `iv` or containing induction statements for `iv`. The recognition of induction statements is

```
procedure Main
  $inc$ = FindIncrement($L_0$)    // $L_0$ is the outermost loop of the nest
  Replace($L_0, iv$)
  InsertStatement($iv = inc$)    // Omit this statement if iv is not live-out
end

procedure FindIncrement($L$)
  // Find the increments incurred by $iv$ relative to the beginning of loop $L$
  // - $inc\_after[s]$ is the increment from beginning of loop body to statement $s$
  // - $inc\_into\_loop[L]$ is the increment from beginning of $L$ to beginning of $j^{th}$ iteration
  // - the subroutine returns the total increment added by $L$
  $inc = 0$
  foreach $si$ of type $I, L$
    if $si$ is type of $L$
      $inc$ += FindIncrement($si$)
    else
      $inc$ += $exp$
    $inc\_after[si] = inc$
  $inc\_into\_loop[L] = \sum_1^{j-1} inc$ // $inc$ may depend on $j$
  return $\sum_1^{ub} inc$
end

procedure Replace($L, initialval$)
  // Substitute $v$ with the closed-form expression
  foreach $si$ of type $I, L, U$
    if $si$ is type of $L$
      Replace($si, val$)
    if $si$ is type of $L, I$
      $val = initialval + inc\_into\_loop[L] + inc\_after[si]$
    if $si$ is type of $U$
      Substitute($si, expr, iv, val$)
end
```

Fig. 6. Induction Variable Substitution Algorithm. The algorithm handles generalized induction variables, as per [15]

similar to the recognition of reduction statements in Section 5.2. The algorithms handle generalized induction variables, where the increment *exp* can either be a loop-invariant expression or a linear expression that depends on another induction variable (with no cyclic relationships being allowed). For the latter case (i.e., *coupled induction variables*), the algorithm is successively applied to all induction variables, beginning with the one with no such dependences.

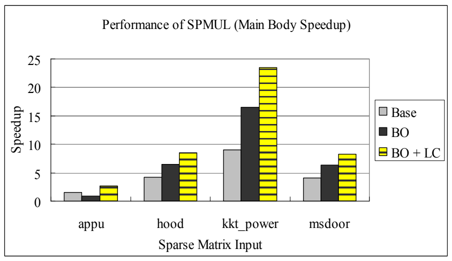
6 Application Passes

6.1 OpenMP-to-GPU: Automatic translation and compiler-driven optimizations

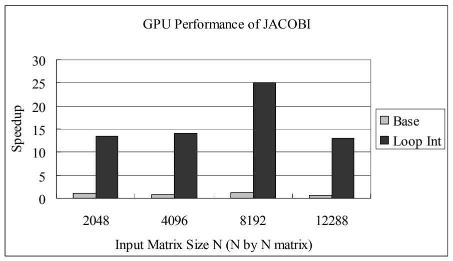
Hardware accelerators, such as GPUs, have emerged as powerful parallel platforms for high-performance computing. While a GPU provides an inexpensive, highly parallel system to application developers, its programming complexity poses a significant challenge for developers. Even though the CUDA (Compute Unified Device Architecture) programming model [16], recently introduced by NVIDIA, offers a more user-friendly programming model General-Purpose computation on GPUs (GPGPU), programming GPGPUs is still complex and error-prone. In this project, we have developed an automatic OpenMP to CUDA GPU translator and optimization techniques. OpenMP is a well-established programming model for shared memory parallel computers. Due to the similarity between the OpenMP and CUDA programming models, we were able to convert the OpenMP parallelism, especially the loop-level parallelism, into the forms that best express parallelism in the CUDA programming model. However, there are architectural differences between traditional shared-memory machines, served by OpenMP, and the stream architecture adopted by most GPGPUs, which may lead to the differences in optimization strategies. To address these issues, we have implemented several transformation techniques to optimize the performance of translated CUDA programs. Preliminary results in Figure 7 show that simple compiler transformation techniques, such as *loop interchange* and *loop coalescing*, can boost the performance of translated GPU programs. In case of *SPMUL* kernel translation (Figure 7 (a)), the base translation without any optimizations (*Base*) gives reasonable speedups, even though *SPMUL* is an irregular application. However, after applying *basic optimizations* (*BO*), such as exploiting registers or shared memory for frequently-accessed global data, and *loop coalescing* transformation (*LC*), the performance dramatically increases. In the *JACOBI* case, naive translation (*Base*) degrades performance severely. However, simple *loop interchange* transformation can cure this problem (*Loop Int*).

6.2 ATune: Compiler-Driven Adaptive Execution

The goal of this project is to develop compilers and runtime systems for dynamically adaptive applications. In previous work, we have created a system for dynamic adaptation of computational applications by tuning compiler options [17]. The current study investigates new methods to enable dynamic, adaptive optimization and tuning in diverse architectures, with emphasis on distributed irregular applications. For irregular applications, such as sparse matrix-vector (SpMV) multiplication kernels, static performance optimizations are difficult because memory access patterns may be known only at runtime. On distributed parallel applications, load balancing and communication cost reduction are two key issues. To address these issues, we have implemented a compiler-driven adaptive load-mapping and communication-algorithm-selection system. Our tuning



(a) SPMUL



(b) JACOBI

Fig. 7. Performance comparison between base translated version and optimized version. The results show that even simple compiler translation techniques can boost the performance of translated GPU programs.

system targets MPI-based distributed irregular applications, where the Cetus compiler inserts various algorithmic alternatives for given MPI collective communication calls, and generates necessary codes for adaptive iteration-to-process mapping. Actual tuning is conducted at runtime with the help of a Cetus-generated tuning driver.

Figure 8 shows the performance improvements of our adaptive runtime tuning system. We applied our techniques to distributed SpMV multiplication kernels (*SPMUL*). Experiments on 26 real sparse matrices in the UF Sparse Matrix Collection [18] show that our adaptive tuning system (*Tuned*) reduces execution times up to 66.7% (33.3% on average) on 16 nodes. More detailed information can be found in [19].

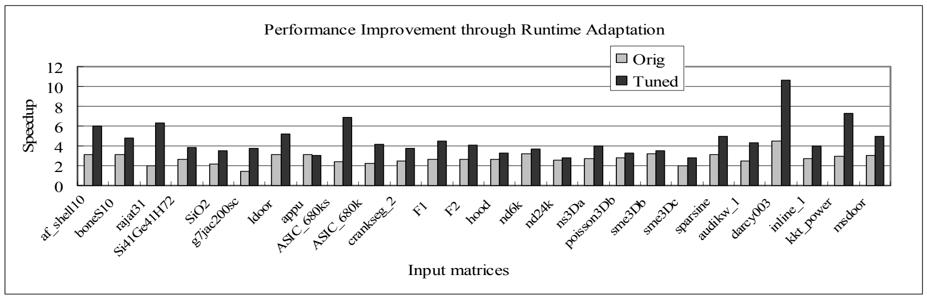


Fig. 8. Performance Improvement Through Adaptive Runtime Tuning. The bars show the speedups of the base parallel version (*Orig*) and adaptively tuned version (*Tuned*) on 16 nodes.

7 Conclusions

Cetus has grown from a simple, student-designed source-to-source translator into a robust systems that is now supported by the National Science Foundation as a community infrastructure. We have presented the current status in terms of the existing internal organization, analysis and transformation passes, and several applications. Cetus is ready for use by the user community. Via the Compunity Portal at http://cetus.ecn.purdue.edu we are able to respond to user requests and incorporate community-developed modules. Through these mechanisms, we expect Cetus to become a research infrastructure that is widely applicable to source-level optimizations and transformations for both multicore and large-scale parallel programs.

References

1. W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu, "Parallel programming with Polaris," *IEEE Computer*, vol. 29, no. 12, pp. 78–82, Dec. 1996.
2. Seung-Jai Min, Seon Wook Kim, Michael Voss, Sang-Ik Lee, and Rudolf Eigenmann, "Portable compilers for openmp," in *OpenMP Shared-Memory Parallel Programming*, Springer Verlag, Heidelberg, Germany, July 2001, Lecture Notes in Computer Science #2104, pp. 11–19.
3. Long Fei and Samuel P. Midkiff, "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*, New York, NY, USA, 2006, pp. 84–95, ACM.
4. Ayon Basumallik and Rudolf Eigenmann, "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*, New York, NY, USA, 2006, pp. 119–128, ACM.
5. Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun, "The opentm transactional application programming interface," in *PACT '07: Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, Washington, DC, USA, 2007, pp. 376–387, IEEE Computer Society.
6. R. Asenjo, R. Castillo, F. Corbera, A. Navarro, A. Tineo, and E.L. Zapata, "Parallelizing irregular c codes assisted by interprocedural shape analysis," in *22nd IEEE International Parallel and Distributed Processing Symposium (IPDPS'08)*, 2008.
7. Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff, "Experiences in using Cetus for source-to-source transformations," in *Proc. of the Workshop on Languages and Compilers for Parallel Computing (LCPC'04)*. Sept. 2004, pp. 1–14, Springer Verlag, Lecture Notes in Computer Science.
8. Randy Allen and Ken Kennedy, *Optimizing compilers for modern architectures*, Morgan Kaufman Publishers, San Francisco, CA, 2002.
9. Michael Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, Cambridge, MA, 1989.

10. William Pugh, “The Omega test: a fast and practical integer programming algorithm for dependence analysis,” in *Proceeding Supercomputing ’91.* 1991, pp. 4–13, IEEE Comput. Soc. Press.
11. William Blume and Rudolf Eigenmann, “The Range Test: A Dependence Test for Symbolic, Non-linear Expressions,” *Proceedings of Supercomputing ’94, Washington D.C.*, pp. 528–537, Nov. 1994.
12. William Blume and Rudolf Eigenmann, “Demand-Driven, Symbolic Range Propagation,” *Lecture Notes in Computer Science, 1033: Languages and Compilers for Parallel Computing*, pp. 141–160, 1996.
13. Rudolf Eigenmann, Jay Hoeflinger, and David Padua, “On the Automatic Parallelization of the Perfect Benchmarks,” *IEEE Trans. Parallel Distributed Syst.*, vol. 9, no. 1, pp. 5–23, Jan. 1998.
14. Peng Tu and David Padua, “Automatic Array Privatization,” in *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science.*, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua, Eds., August 12-14, 1993, vol. 768, pp. 500–521.
15. Bill Pottenger and Rudolf Eigenmann, “Idiom Recognition in the Polaris Parallelizing Compiler,” *Proceedings of the 9th ACM International Conference on Supercomputing*, 95.
16. “NVIDIA CUDA [online]. available: http://developer.nvidia.com/object/cuda.html,” .
17. Zhelong Pan and Rudolf Eigenmann, “Fast, automatic, procedure-level performance tuning,” in *Proc. of Parallel architectures and Compilation Techniques*, 2006, pp. 173–181.
18. T. Davis, “University of Florida Sparse Matrix Collection [online]. available: http://www.cise.ufl.edu/research/sparse/matrices/,” .
19. Seyong Lee and Rudolf Eigenmann, “Adaptive runtime tuning of parallel sparse matrix-vector multiplication on distributed memory systems,” in *ACM International Conference on Supercomputing (ICS08)*, June 2008, pp. 195–204.