Experiences in Using Cetus for Source-to-Source Transformations[⋆]

Troy A. Johnson, Sang-Ik Lee, Long Fei, Ayon Basumallik, Gautam Upadhyaya, Rudolf Eigenmann, and Samuel P. Midkiff

School of Electrical & Computer Engineering
Purdue University, West Lafayette IN 47906, USA,
{troyj,sangik,lfei,basumall,gupadhya,eigenman,smidkiff}@ecn.purdue.edu,
http://www.ece.purdue.edu/ParaMount

Abstract. Cetus is a compiler infrastructure for the source-to-source transformation of programs. Since its creation nearly three years ago, it has grown to over 12,000 lines of Java code, been made available publically on the web, and become a basis for several research projects. We discuss our experience using Cetus for a selection of these research projects. The focus of this paper is not the projects themselves, but rather how Cetus made these projects possible, how the needs of these projects influenced the development of Cetus, and the solutions we applied to problems we encountered with the infrastructure. We believe the research community can benefit from such a discussion, as shown by the strong interest in the mini-workshop on compiler research infrastructures where some of this information was first presented.

1 Introduction

Parallelizing compiler technology is most mature for the Fortran 77 language [1, 3, 12, 13, 16]. The simplicity of the language without pointers or user-defined types makes it easy to analyze and to develop many advanced compiler passes. By contrast, parallelization technology for modern languages, such as Java, C++, or even C, is still in its infancy. When trying to engage in such research, we were faced with a serious challenge. We were unable to find a parallelizing compiler infrastructure that supported interprocedural analysis, exhibited state-of-the-art software engineering techniques to help shorten development time, and allowed us to compile large, realistic applications. We feel these properties are of paramount importance because they enable a compiler writer to develop "production strength" passes. Production strength passes, in turn, can work in the context of the most up-to-date compiler technology and lead to compiler research that can be evaluated with full suites of realistic applications. The lack of such thorough evaluations in many current research papers has been observed and criticized by many. The availability of an easy-to-use compiler infrastructure

⋆ This material is based upon work supported in part by the National Science Foundation under Grant No. 9703180, 9975275, 9986020, and 9974976.

would help improve this situation significantly. Hence, continuous research and development in this area are among the most important tasks of the compiler community.

Cetus was created with those needs in mind. It supports analyses and transformations at the source level; other infrastructures are more appropriate for instruction-level compiler research. Cetus is composed of over 10,000 lines of Java code that implements the Cetus intermediate representation (IR), over 1,500 lines of code that implements source transformations, a C parser using Antlr, and standalone C and C++ Bison parsers that have yet to be integrated completely into Cetus. The Cetus IR is the product of three graduate students working part-time over two years. Several others have contributed analysis and transformation passes, as well as used Cetus for their own research projects. We discuss these projects in this paper from the perspective of how Cetus made these projects possible, how the needs of these projects influenced the development of Cetus, and the solutions we applied to problems we encountered with the infrastructure. We believe the research community can benefit from such a discussion, as shown by the strong interest in the mini-workshop on compiler research infrastructures where some of this information was first presented.

Section 2 briefly covers the Cetus IR. In Section 3, we cover basic analysis, transformation, and instrumentation passes. Section 4 contains five case studies of more complex passes. Section 5 discusses the effects of user-feedback on the project. Finally, Section 6 concludes.

2 Cetus Intermediate Representation

For the design of the IR we chose an abstract representation, implemented in the form of a class hierarchy and accessed through the class member functions. We consider a strong separation between the implementation and the interface to be very important. In this way, a change to the implementation may be done while maintaining the API for its users. It also permits passes to be written before the IR implementation is ready. These concepts had already proved their value in the implementation of the Polaris infrastructure [2], which served as an important example for the Cetus design. Polaris was rewritten three to four times over its lifetime while keeping the interface, and hence all compilation passes, nearly unmodified [5]. Cetus has a similar design, shown in Figure 1, where the high-level interface insulates the pass writer from changes in the base.

Our design goal was a simple IR class hierarchy easily understood by users. It should also be easy to maintain, while being rich enough to enable future extension without major modification. The basic building blocks of a program are the *translation units*, which represent the content of a source file, and *procedures*, which represent individual functions. Procedures include a list of simple or compound statements, representing the program control flow in a hierarchical way. That is, compound statements, such as *IF*-constructs and *FOR*-loops include inner (simple or compound) statements, representing *then* and *else* blocks

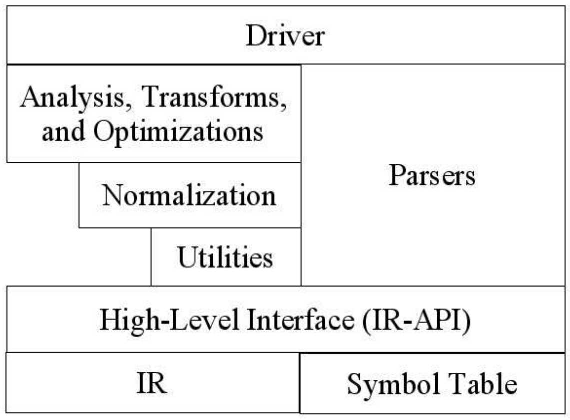
Driver
Analysis, Transforms, and Optimizations
Normalization
Utilities
Parsers
High-Level Interface (IR-API)
IR
Symbol Table

Fig. 1. Cetus components and interfaces: Components of Cetus only call methods of the components beneath them. The driver interprets command-line arguments and initiates the appropriate parser for the input language, which in turn uses the high-level interface to build the IR. The driver then initiates analysis and transformation passes. Normalization passes and utilities are provided to perform complex operations that are useful to multiple passes. The interface functions are kept lean and generally provide only a single way of performing IR manipulations.

or loop bodies, respectively. *Expressions* represent the operations being done on variables, including the assignments to variables.

Cetus' IR contrasts with the Polaris Fortran translator's IR in that it uses a hierarchical statement structure. The Cetus IR directly reflects the block structure of a program. Polaris lists the statements of each procedure in a flat way, with a reference to the outer statement being the only way for determining the block structure. There are also important differences in the representation of expressions, which further reflects the differences between C and Fortran. The Polaris IR includes assignment statements, whereas Cetus represents assignments in the form of expressions. This corresponds to the C language's feature to include assignment side effects in any expression.

The IR is structured such that the original source program can be reproduced, but this is where source-to-source translators face an intrinsic dilemma. Keeping the IR and output similar to the input will make it easy for the user to recognize the transformations applied by the compiler. On the other hand, keeping the IR language-independent leads to a simpler compiler architecture, but may make it impossible to reproduce the original source code as output. In Cetus, the concept of statements and expressions are closely related to the syntax of the C language, facilitating easy source-to-source translation. The correspondence between syntax and IR is shown in Figure 2. However, the drawback is increased complexity for pass writers (since they must think in terms of C syntax) and limited extensibility of Cetus to additional languages. That problem is mitigated by the provision of several interfaces that represent generic control constructs. Generic passes can be written using the abstract interface, while more language-specific

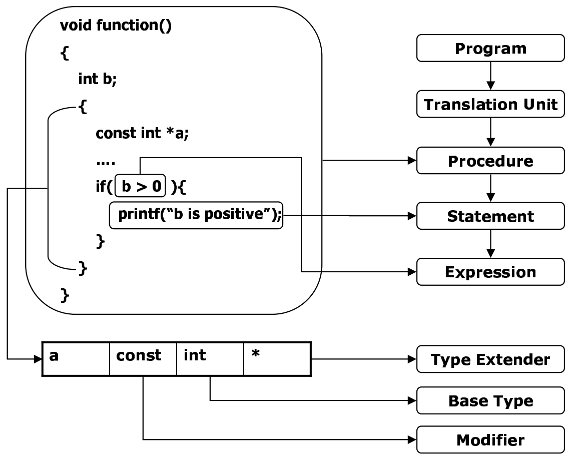

Fig. 2. A program fragment and its IR in Cetus. IR relationships are similar to the program structure and a symbol table is associated with each block scope.

passes can use the derived classes. For example, the classes that represent for-loops and while-loops both implement a loop interface. A pass that manipulates loops may be written using the generic loop interface if the exact type of loop is not important.

The high-level interface, or IR-API, is the interface presented to compiler writers. In general the IR-API is kept minimal and free of redundant functionality, so as to make it easy to learn about its basic operation and easy to debug. IR-API calls expect the IR to be in a consistent state upon entry and ensure the state is consistent upon their return. Cetus also provides a utility package, that offers convenience to pass writers. The utility package provides additional functions, where needed by more than a single compiler pass. Obviously, this criterion will depend on the passes that will be written in the future. Hence, the utilities will evolve, while we expect the base to remain stable. The utility functions operate using only the IR-API.

2.1 Navigating the IR

Traversing the IR is a fundamental operation that will be used by every compiler pass. Therefore, it is important that traversals be easy to perform and require little code. Cetus provides an abstract IRIterator class that implements the standard Java Iterator interface. The classes BreadthFirstIterator, DepthFirstIterator, and FlatIterator are all derived from IRIterator. The constructor for each of these classes accepts as its only parameter a Traversable object which defines the root of the traversal. Traversable is an interface that ensures any implementing class can act as a node of a tree by providing methods to access its parent and children. A design alternative here was to have every class provide a getIterator method instead of passing the root object to an iterator constructor, but that required adding an implementation of getIterator to every class, and was rejected.[1] The DepthFirstIterator visits statements and expressions sequentially in program order without regard to block scope. The BreadthFirstIterator visits all children of an object before visiting the children's children; i.e., block scope is respected with outer objects visited first. The FlatIterator does not visit the root of the traversal and instead visits the root's children sequentially without visiting the children's children.

In addition to providing a `next` method, as all Iterators must, an IRIterator provides `next(Class)`, `next(Set)`, and `nextExcept(Set)` to allow the caller to specify that only objects of a certain class, or that belong or do not belong to a particular set of classes, are of interest. When these methods were first introduced, we were able to rewrite older Cetus passes using considerably fewer lines of code. Figure 3 shows the usefulness of these methods.

```
/* Look for loops in a procedure. Assumes proc is a Procedure object. */

BreadthFirstIterator iter = new BreadthFirstIterator(proc);
try {
  while (true)
  {
    Loop loop = (Loop)iter.next(Loop.class);
    // Do something with the loop
  }
} catch (NoSuchElementException e) {
}
```

Fig. 3. Using iterators to find loops within a procedure. Outer loops are discovered first.

2.2 Type System and Symbol Table

Modern programming languages provide rich type systems. In order to keep the Cetus type system flexible, we divided the elements of a type into three concepts: base types, extenders, and modifiers. A complete type is described by a combination of these three elements. Base types include built-in primitive types, which have a predefined meaning in programming languages, and user-defined types. User-defined types are new types introduced into the program by providing the layout of the structure and the semantics. These include typedef, struct, union, and enum types in C. Base types are often combined with type extenders. Examples of type extenders are arrays, pointers, and functions. The last concept is modifiers which express an attribute of a type, such as const and volatile in C. They can decorate any part of the type definition. Types are understood by decoding the description one element at a time, which is a

[1] The decision was primarily due to Java's lack of multiple inheritance, since in most cases inheritance had already been used.

sequential job in nature. We use a list structure to hold type information so that types can be understood easily by looking at the elements in the list one at a time.

Another important concept is a symbol, which represents the declaration of a variable in the program. Symbols are not part of the IR tree and reside in symbol tables. Our concept of a symbol table is a mapping from a variable name to its point of declaration, which is located in a certain scope and has all of the type information. As a result, scope always must be considered when dealing with symbols. In C, a block structure defines a scope. Therefore, structs in C are also scopes and their members are represented as local symbols within that scope. A compiler may use one large symbol table with hashing to locate symbols [4]. In Cetus, since source transformations can move, add, or remove scopes, we use distributed symbol tables where each scope has a separate physical symbol table. The logical symbol table for a scope includes its physical symbol table and the physical symbol tables of the enclosing scopes, with inner declarations hiding outer declarations. There are certain drawbacks to this approach, namely the need to search through the full hierarchy of symbol tables to reach a global symbol [6], but we find it to be convenient. For example, all the declarations in a scope can be manipulated as a group simply by manipulating that scope's symbol table. It is especially convenient in allowing Cetus to support object-oriented languages, where classes and namespaces may introduce numerous scopes whose relationships can be expressed through the symbol table hierarchy.

3 Capabilities for Writing Passes

Cetus has a number of features that are useful to pass writers. Classes that support program analysis, normalization, and modification are discussed below.

3.1 Analysis

Call Graph Cetus provides a CallGraph class that creates a call graph from a Program object. The call graph maps callers to callees as well as callees to callers. A pass can query the call graph to determine if a procedure is a leaf of the call graph or if a procedure is recursive.

Control-Flow Graph Cetus provides a ControlFlowGraph class, which creates a control-flow graph from a Program object. The graph represents the structure of each procedure in terms of its basic blocks connected by control-flow edges.

3.2 Normalization

Single Return Compiler passes often become simpler if they can assume that each procedure has a single exit point. A procedure with multiple return statements complicates such actions as inserting code that should execute just prior to a procedure returning to its caller. To eliminate this problem, the single-return pass creates a new variable to hold the return value, identifies all return statements, and replaces the return statements with an assignment to the variable followed immediately by a goto to the end of the procedure. Then, the procedure is appended with a single return statement that returns the value of the variable.

Single Call Instrumentation passes sometimes need to place code before and after procedure calls, but most languages allow multiple calls per statement. Thus, the pass writer has two choices: find a way within the language to insert their code into an already complex statement, or separate all the calls so there is one per statement. The first option often is not possible, and when it is, such as by using the comma operator in C, it results in obfuscated code. Therefore, Cetus provides a single-call pass to separate statements with multiple calls, including "unwrapping" nested calls. New variables are introduced to hold return values and the calls are placed in the appropriate order. It is worth noting that the C language standard leaves undefined the order of execution of multiple calls in a statement at the same nesting level. Arbitrarily ordering such calls left to right did not appear to affect any programs that we tested.

Single Declarator Languages allow multiple symbols to be introduced by a single declaration. An example is `int x, y, z;` in C. If a pass needs to copy, move, or remove the declaration of a single variable, then it must be careful not to change the rest of the declaration. Cetus provides a single-declarator pass that would separate the example above into three declarations, allowing passes to work on individual declarations.

3.3 Modifying the Program

Annotation System The Cetus IR provides an Annotation class that is derived from the general Declaration class. Annotations can appear wherever declarations can appear, allowing them to appear both inside and outside of procedures. They can be used to insert comments, pragmas, raw text, or a hash map to act as a database that facilitates the exchange of information between passes.

Inserting New Code All of the statement and expression classes in the Cetus IR have constructors that can be used to create new IR. These constructors are used by the parser to create the IR for the program. Therefore, pass writers are able to insert new IR in exactly the same way as the parser creates the IR for the original code. The constructors and other methods of the IR classes check their arguments to ensure that the IR remains consistent at all times. For example, an exception is thrown if a duplicate symbol is about to be created, or if an attempt is made to place the same IR object in two parts of the IR tree.

4 Case Studies

Here we present five case studies in which Cetus was used to accomplish more complex analyses and transformations. Each case study was written by the person who used Cetus for that purpose, so this section represents the experiences and opinions of five different people.

4.1 Extraction of Loops into Subroutines

A number of loop transformations are more easily applied if the loop is available as a separate subroutine. The micro-tasking implementation described below in Section 4.2 is one such example. Separating a loop from a procedure and moving it to its own procedure faces several issues. There will be values used by the loop that are defined above the loop and must be passed by value into the new procedure. There will be values used below the loop that are defined within the loop and must be passed by reference (or by pointer) into the new procedure. Cetus has basic use-def analysis to support both of these.

A Cetus utility method to search and replace variables by name is very useful to this pass. For example, if a variable `p` is passed to the new procedure via a pointer (i.e., it was not a pointer in the original code) then all occurrences of `p`, must be replaced with `*p` in the new procedure. The search and replace method must know to skip replacing names that are structure members, because, for example, `x.*p` is not legal C code.

4.2 Translation of OpenMP Applications

One of the early experiences in using Cetus was the creation of an OpenMP translator pass. OpenMP is currently one of the most popular paradigms for programming shared-memory parallel applications [7]. Unlike programming with MPI (message-passing interface), where one inserts communication library calls, OpenMP programmers use directives that have semantics understood by the compiler.

Compiler functionality for translating OpenMP falls into two broad categories. The first category deals with the translation of the OpenMP work-sharing constructs into a micro-tasking form, requiring the extraction of the work-sharing code (such as the bodies of parallel loops) into separate micro-tasking subroutines. It also requires inserting corresponding function calls and synchronization. Cetus provides functionality that is sufficient for these transformations. The second category deals with the translation of the OpenMP data clauses, which requires support for accessing and modifying symbol table entries. Cetus provides several ways in which the pass writer can access the symbol table to add and delete symbols or change their scope. There are currently two different OpenMP translators which have been implemented using Cetus. Both of them use the same OpenMP front end. One translator generates code for shared-memory systems using the POSIX threads API. The other translator targets software distributed shared memory systems and was developed as part of a project to extend OpenMP to cluster systems [11]. Although the entire OpenMP 2.0 specification is not supported yet, the translators are powerful enough to handle benchmarks such as art and equake , two of the larger applications from the SPEC OMPM2001 suite.

Cetus is also being used in an ongoing project to translate OpenMP applications directly to MPI. The project is also a source-to-source translation, but it makes use of a wider range of compiler techniques and Cetus functionality. One major component of this transformation is the interprocedural analysis of array accesses. A regular array section analysis pass was implemented in Cetus to summarize array accesses within loops using Regular Section Descriptors [8]. The flow graph described in Section 3, along with the regular section analysis pass, was used to implement an array dataflow pass. The array dataflow pass is then used to resolve producer-consumer relationships for shared data, and to insert MPI calls to satisfy these relationships. Three aspects of Cetus greatly facilitated the development of these passes. First, Cetus provides a rich set of iterators for program traversal. Second, Cetus provides functions for conveniently accessing the symbol tables visible within specific scopes, as well as their parent scopes. Finally, Cetus provides a convenient interface for the insertion of function calls at the source level in a program. These aspects of Cetus allowed us to conveniently create the requisite dataflow passes and insert the MPI calls.

4.3 Pointer Alias Analysis

Pointer analysis generally has two different variations. Points-to analysis is a program analysis pass that determines the set of memory objects that a pointer could point to at a given place in the program. Similarly, alias analysis is a program analysis pass that determines if two pointers can point to the same memory object at a given place in the program. These two analyses are related in the sense that alias analysis could be done by doing a points-to analysis first and then applying set intersection operations.

The goal was to write a context-sensitive and flow-sensitive points-to analysis. The pass was written using an earlier version of Cetus [10] and updated to use newer features. To implement a points-to analysis pass, the underlying compiler has to support several basic features. First, pointer variables should be easy to identify. This requires adequate symbol table operations. Second, our points-to analysis is an iterative analysis that traverses the entire program, finding pointer related expressions and evaluating them until reaching a fixed-point. While the earlier Cetus version provided limited flexibility for traversing the program IR, the new functionality mentioned in Section 2 greatly simplified this task.

Writing a flow-sensitive analysis pass requires a control-flow graph, as described in Section 3.1. Additionally, program normalization functionality, as per Section 3.2, helped reduce the complexity of the pass substantially. It resulted in more regular expressions that needed less special-case handling. Similarly, normalizing each statement to have a single function call simplified the interprocedural analysis pass. We also normalized each statement to have a single assignment. Flow sensitive analysis requires keeping track of changes to points-to sets at every program point. Saving the entire points-to set per statement requires excessive memory, so an incremental way of recording the points-to set change is needed. We implemented a method that records the change to the points-to set at each program location only. When the pass needs to look at the entire points-to set, all reaching definitions are looked up. We implemented this functionality by representing the program in SSA form. The availability of a control-flow graph was useful for creating the SSA representation.

4.4 Software Debugging

Cetus is a useful tool for source-level instrumentation. The high-level IR keeps all of the information available from source code. The hierarchical-structured IR and iterators make it easy to traverse the IR tree. Each object in the IR tree has its direct corresponding element in the source code. Each `Statement` object keeps the line number in the source file of the statement it represents. With all this information, the user can write an instrumentation pass that does the transformation analogous to the transformation the user would directly apply on the source code. That is, the gap between the abstract instrumentation pass in Cetus and the concrete instrumentation that user expects to apply to the source code is small. The small gap makes it easy for the user to design an instrumentation pass with Cetus.

However, there are still limitations. First, creating an IR object (e.g. `Symbol`, `Expression`, `Statement`) is not easy. For instance, if the user wishes to add a `Statement` object into the IR tree, he/she has to create the whole subtree (with the `Statement` object as its root). Because the user typically thinks in C, there is a big gap between the concept of inserting instrumentation into the source code and creating an IR subtree in the existing IR. It is impractical for the instrumentation pass writer to build the IR subtree corresponding to the instrumentation he/she wishes to insert if the instrumentation is complex or if the pass writer has insufficient knowledge about Cetus parsing and IR implementation. It would be useful to have an on-demand utility that can properly parse the instrumentation the user wishes to add (expressed in `C`) and return a legal IR subtree that fits into the context. For instance, if the user wishes to add a `printf` statement which displays the value of a local variable, the utility should translate (`printf("value = %d", local_symbol);`) into an IR subtree and return it to the user. The utility should also make proper changes to the symbol table and make proper reference to the local symbol used in the instrumentation. Such a utility has not been implemented in Cetus because it requires maintaining multiple parsers (e.g., one for the entire C language, one only for statements, and one only for expressions), or a parser generator that supports multiple starting productions.

Second, the requirement to keep the IR tree consistent, as per Section 2, makes it less flexible to instrument the IR tree. It is a design specification that every operation on the IR tree should result in a legal IR tree, i.e. in a series of operations, the IR tree should be consistent after every operation. This requirement trades the flexibility in manipulating the IR tree for correctness and robustness of the IR. However, it is commonplace that instrumentation is done out of order. For example, if the instrumentation uses temporary data structures, and an analysis pass is performed after the instrumentation pass to determine how many of the data structures can be reused to avoid excessive waste of memory, it is not determined what temporary data structures really needed to be declared until we finish the analysis pass. In this scenario, it is desirable to add the uses of the temporary data structures in the instrumentation pass before we add the declarations of those temporary data structures that are really needed after the analysis pass. It is sometimes infeasible or not desirable for modularization reasons to interleave the redundancy analysis with the instrumentation pass.

In order to avoid the two limitations discussed above, we develop a two-phased instrumentation utility. The first phase is a Cetus instrumentation pass that traverses the IR tree, performs analysis, and logs the instrumentation operations needed to be performed (in any order) in an instrumentation profile. The second phase is an instrumentation program, which reads in the instrumentation profile generated by the instrumentation pass, rearranges the instrumentation operations into a proper ordering, and performs the instrumentation operations on the source files. Different instrumentation tasks need to have different Cetus instrumentation passes, while the second phase is shared. This instrumentation utility is used in our past research on the AccMon project [17], where instrumentation is added to turn on runtime monitoring on memory locations that need to be monitored.

4.5 A Java Byte-Code Translator

We used Cetus to construct a bytecode-to-bytecode translator with the purpose of experimenting with optimization passes for the Java programming language. With this infrastructure, we plan on performing quasi-static [14] optimizations at the bytecode level – these are ahead-of-time optimizations in which assumptions about other classes are checked at runtime (by an intelligent classloader [9]) to verify the correctness of the off-line optimizations. We will initially target numerical programs, but longer term will explore more general purpose optimizations. Our input was a bytecode (Java class) file. We broke up the translation process as follows. First we read in the bytecode and stored it in standard Java data structures. Then we constructed an intermediate representation (IR) based on the data we had gathered. This IR was then used to drive the back-end which translated the IR into bytecode. Optimization passes will be added to the tool by acting at the IR, rather than the bytecode, level. Cetus was used as the IR of choice for our project. We converted our bytecode to Cetus by

1. reading in bytecode into an internal ClassFile data structure, as specified by Sun in the JVM documentation [15]

2. parsing the ClassFile data structure and extracting information about individual statements/declarations/ definitions and
3. constructing Cetus IR by performing a mapping between the parsed data and (Cetus) IR classes.

Our experiences with Cetus have been extremely favorable to date. The object-orientedness of the tool combined with the abundance of good documentation allowed us to construct our IR in a very short time-span. The only problem encountered was with the fact that Cetus was designed with C++ and not Java in mind and while the similarities in the approaches are self-evident, we did need to modify the code occasionally to facilitate our target language, i.e. Java. However, the fact that Cetus was written almost entirely in Java proved to be extremely beneficial since changes made were minimal and rapidly implemented.

5 Users' Influence on Cetus Development

Beginning with its first usable version and continuing throughout its development, Cetus has been used for both research projects and course projects. Feedback from the people involved in those projects provided direction for further development.

One of the first suggestions was to improve the iterators. A tree-structured IR requires that there be code to traverse the tree. Completely exposing the traversal code to pass writers places an unnecessary burden on them; hence, iterators were provided. However, the iterators were not initially provided as they have been described in this paper – only the standard form of the `next` method was provided. Pass writers noted that much of their code was spent type-checking the object returned by `next` to decide whether or not it was an object of interest to them. The solution was to allow the type of the desired object to be passed to the `next` method, allowing the method internally to skip over objects that did not match the specified type, thus hiding the type-checking code. The savings to the pass writer is only a few lines of code, but those lines are saved each time an iterator is used. Consensus was that the improved iterators allowed for shorter, more readable code.

Another example of user-driven improvement was the AssignmentExpression class. Originally, there was only a BinaryExpression class that was used to represent all types of binary expressions. The users found this to be very inconvenient because they often wished to find definitions of a variable. Finding definitions required them first to search for instances of BinaryExpression and then to test if each instance was any of the many forms of assignment that the C language provides. Not only was this process inconvenient, but inefficient, since typical programs contain a large number of binary expressions. The solution was to split the duties of the BinaryExpression class by deriving from it an AssignmentExpression class. If an object was an instance of AssignmentExpression, then users knew automatically that it was a binary expression that modified its lefthand side. Combined with the improved iterators, users could skip the other binary expressions by requesting that `next` only return AssignmentExpressions.

6 Conclusion

We briefly discussed the Cetus IR and presented five case studies that used Cetus to perform non-trivial operations. Cetus was shown to have an API sufficient for a variety of applications. The API has been improved based on users' feedback. We observe that most of the difficulties that users encountered were in using Cetus to find the part of the program they wished to transform or optimize (i.e., to find statements or expressions satisfying a certain property); few complaints dealt with using Cetus to perform the actual transformation. It is interesting to consider if developers of other compiler infrastructures have noticed a similar phenomenon.

Overall, Cetus' users have found it to be a useful tool, and with the program and source code now available for download, we expect that more people will make use of Cetus. Future development focuses on adding support for C++ and finding additional ways to shorten the code necessary for writing passes.

References

1. P. Banerjee, J. A. Chandy, M. Gupta, et al. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.
2. W. Blume, R. Doallo, R. Eigenmann, et al. Advanced Program Restructuring for High-Performance Computers with Polaris. *IEEE Computer*, pages 78–82, December 1996.
3. W. Blume, R. Eigenmann, et al. Restructuring Programs for High-Speed Computers with Polaris. In *ICPP Workshop*, pages 149–161, 1996.
4. R. P. Cook and T. J. LeBlanc. A Symbol Table Abstraction to Implement Languages with Explicit Scope Control. *IEEE Transactions on Software Engineering*, 9(1):8–12, January 1983.
5. K. A. Faigin, S. A. Weatherford, J. P. Hoeflinger, D. A. Padua, and P. M. Petersen. The Polaris Internal Representation. *International Journal of Parallel Programming*, 22(5):553–586, 1994.
6. C. N. Fischer and R. J. LeBlanc Jr. *Crafting a Compiler.* Benjamin/Cummings, 1988.
7. O. Forum. OpenMP: A Proposed Industry Standard API for Shared Memory Programming. Technical report, Oct. 1997.
8. P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.
9. G. C. Lee and S. P. Midkiff. Ninja 2: Towards fast, portable, numerical Java. In *Workshop on Compilers for Parallel Computing*, July 2004.
10. S.-I. Lee, T. A. Johnson, and R. Eigenmann. Cetus - An Extensible Compiler Infrastructure for Source-to-Source Transformation. In *16th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 539–553, October 2003.
11. S.-J. Min, A. Basumallik, and R. Eigenmann. Optimizing OpenMP programs on Software Distributed Shared Memory Systems. *International Journal of Parallel Programming*, 31(3):225–249, June 2003.

12. T. N. Nguyen, J. Gu, and Z. Li. An Interprocedural Parallelizing Compiler and Its Support for Memory Hierarchy Research. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, pages 96–110, 1995.
13. C. Polychronopoulos, M. B. Girkar, et al. The Structure of Parafrase-2: An Advanced Parallelizing Compiler for C and Fortran. In *Languages and Compilers for Parallel Computing*. MIT Press, 1990.
14. M. J. Serrano, R. Bordawekar, S. P. Midkiff, and M. Gupta. Quicksilver: a Quasi-Static Compiler for Java. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 66–82, 2000.
15. Sun Microsystems. *The Java Virtual Machine Specification*.
16. R. P. Wilson, R. S. French, et al. SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers. *SIGPLAN Notices*, 29(12):31–37, 1994.
17. P. Zhou, W. Liu, L. Fei, S. Lu, F. Qin, Y. Zhou, S. Midkiff, and J. Torrellas. AccMon: Automatically detecting memory-related bugs via program counter-based invariants. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO'04)*, 2004.