Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation

Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann

Purdue University*, West Lafayette IN 47906, USA,
{sangik,troyj,eigenman}@ecn.purdue.edu,
WWW home page: http://paramount.www.ecn.purdue.edu

Abstract. Cetus is a compiler infrastructure for the source-to-source transformation of programs. We created Cetus out of the need for a compiler research environment that facilitates the development of interprocedural analysis and parallelization techniques for C, C++, and Java programs. We will describe our rationale for creating a new compiler infrastructure and give an overview of the Cetus architecture. The design is intended to be extensible for multiple languages and will become more flexible as we incorporate feedback from any difficulties we encounter introducing other languages. We will characterize Cetus' runtime behavior of parsing and IR generation in terms of execution time, memory usage, and parallel speedup of parsing, as well as motivate its usefulness through examples of projects that use Cetus. We will then compare these results with those of the Polaris Fortran translator.

1 Introduction

Parallelizing compiler technology is most mature for the Fortran 77 language [4, 5, 16, 18]. 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 supports interprocedural analysis, exhibits state-of-the-art software engineering techniques to achieve shorter development time, and which would allow us to compile large, realistic applications. However, we feel these properties are of paramount importance. 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 would help improve this situation significantly. Hence, continuous

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

research and development in this area are among the most important tasks of the compiler community. Our paper contributes to this effort.

During an early development stage, Cetus was used in a class project. Ten students of a graduate compiler class were challenged to create a source-to-source C compiler with a number of passes fundamental to parallelization, including induction variable substitution, dependence analysis, and privatization. The students were free to choose the compiler infrastructure. Among the serious contenders were the GNU Compiler Collection (GCC) [21], the SUIF 2 [24] compiler (a.k.a. the National Compiler Infrastructure), and a "from-scratch" design building on Cetus. After an initial feasibility study, half of the students decided to pursue the GCC option and the other half the Cetus option. This provided an excellent opportunity to see if Cetus could meet our goals. The discussion of Cetus versus GCC will reflect some of the findings of the class, given in the final project review. The success of the class project led to a new infrastructure that will be made available to the research community.

Cetus has the following goals, which will be explored throughout this paper:

- The Internal Representation (IR) is visible to the pass writer (the user) through an interface, which we will refer to as the IR-API. Designing a simple, easy-to-use IR-API, that is extensible for future capabilities – especially to support other languages – is the most difficult engineering task.
- It must be easy to write source-to-source transformations and optimization passes. The implementation is an object-oriented class hierarchy with a minimal number of IR-API method names (using virtual functions and consistent naming), easy-to-use IR traversal methods, and information that can be inferred from other data strictly hidden from the user.
- Ease of debugging can be decisive for the success of any compiler projects that make use of the infrastructure. The IR-API should make it impossible to create inconsistent program representations, but we still need tools that catch common mistakes and environments that make it easy to track down bugs if problems occur.
- Cetus should run on multiple platforms with no or minimal modification. Portability of the infrastructure to a wide variety of platforms will make Cetus useful to a larger community.

2 Design Rationale and Comparison with Existing Infrastructures

From a substantial list of compiler infrastructures, we choose to discuss three open-source projects that most closely match our goals. The goals are to create a source-to-source infrastructure that supports C and is extensible to other languages. The three projects are the Polaris, SUIF, and GNU compilers. We explain our reasons for not using these infrastructures as our basis, and also discuss important features of these compilers that we want to adopt in Cetus.

2.1 The Polaris Compiler

The Polaris [5] compiler, which we have co-developed in prior work, was an important influence on the design of our new infrastructure. Polaris is written in C++ and operates on Fortran 77 programs. So far, no extensions have been made to handle Fortran 90, which provides a user-defined type system and other modern programming language features. Polaris' IR is Fortran-oriented [7] and extending it to other languages would require substantial modification. In general, Polaris is representative of compilers that are designed for one particular language, serve their purpose well, but are difficult to extend. Cetus should not be thought of as "Polaris for C" because it is designed to avoid that problem. However, there are still several Polaris features that we wanted to adopt in Cetus. Polaris' IR can be printed in the form of code that is similar to the source program. This property makes it easy for a user to review and understand the steps involved in Polaris-generated transformations. Also, Polaris' API is such that the IR is in a consistent state after each call. Common mistakes that pass writers make can be avoided in this way.

2.2 SUIF – National Compiler Infrastructure

The SUIF [24] compiler is part of the National Compiler Infrastructure (NCI), along with Zephyr [3], whose design began almost a decade ago. The infrastructure was intended as a general compiler framework for multiple languages. It is written in C++, like Polaris, and the currently available version supports analysis of C programs. SUIF 1 is a parallelizing compiler and SUIF 2 performs interprocedural analysis [2].

Both SUIF and Cetus fall into the category of extensible source-to-source compilers, so at first SUIF looked like the natural choice for our infrastructure. Three main reasons eliminated our pursuit of this option. The first was the perception that the project is no longer active – the last major release was in 2001 and does not appear to have been updated recently. The second reason was, although SUIF intends to support multiple languages, we could not find complete front ends other than for C and an old version of Java. Work began on front ends for Fortran and C++ [1, 2, 11], but they are not available in the current release. Hence, as is, SUIF essentially supports a single language, C. Finally, we had a strong preference for using Java as the compiler implementation language. Java offers several features conducive to good software engineering. It provides good debugging support, high portability, garbage collection (contributing to the ease of writing passes), and its own automatic documentation system. These facts prompted us to pursue other compiler infrastructures.

2.3 GNU Compiler Collection

GCC [21] is one of the most robust compiler infrastructures available to the research community. GCC generates highly-optimized code for a variety of architectures, which rivals in many cases the quality generated by the machine

vendor's compiler. Its open-source distribution and continuous updates make it attractive. However, GCC was not designed for source-to-source transformations. Most of its passes operate on the lower-level *RTL* representation. Only recent versions of GCC (version 3.0 onward) include an actual syntax tree representation. This representation was used in our class project for implementing a number of compiler passes. Other limitations are GCC compiles one source file at a time, performs separate analysis of procedures, and requires extensive modification to support interprocedural analysis across multiple files.

The most difficult problem faced by the students was that GCC does not provide a friendly API for pass writers. The API consists largely of macros. Passes need to be written in C and operations lack logical grouping (classes, namespaces, etc), as would be expected from a compiler developed in an object-oriented language.

GCC's IR [20] has an ad-hoc type system, which is not reflected in its implementation language (C). The type system is encoded into integers that must be decoded and manipulated by applying a series of macros. It is difficult to determine the purpose of fields in the IR from looking at the source code, since in general every field is represented by the same type. This also makes it difficult for debuggers to provide meaningful information to the user.

Documentation for GCC is abundant. The difficulty is that the sheer amount ([21] and [20] combined approach 1000 pages) easily overwhelms the user. Generally, we have found that there is a very steep learning curve in modifying GCC, with a big time investment to implement even trivial transformations.

The above difficulties were considered primarily responsible for the students that used GCC proceeding more slowly than those creating a new compiler design. The demonstrated higher efficiency of implementation was the ultimate reason for the decision to pursue the full design of Cetus.

2.4 Cetus

Among the most important Cetus design choices were the implementation language, the parser, and the internal representation with its pass-writer interface. We will not present any language discussion in this paper. As mentioned above, the language of choice for the new infrastructure is Java.

Cetus does not contain any proprietary code and relies on freely available tools. For creating a Cetus parser we considered using the parser generators Yacc [13] and Bison [9], which use lex [14] or flex [10] for scanning, and Antlr [17], which is bundled with its own scanner generator. Yacc and Bison generate efficient code in C for an LALR(1) parser, which handles most languages of interest. However, neither generates Java code. By contrast, Antlr generates code in C, C++, Java, or C#. It is an LL(k) parser, which can be more restrictive; however, there is good support for influencing parse decisions using semantic information. Antlr grammars for C and Java exist, but to our knowledge there have not been any successful attempts to use Antlr for parsing arbitrary C++ programs, though Antlr has successfully been used to parse subsets of the language. We selected Antlr for the C front end, because it generates Java code that easily

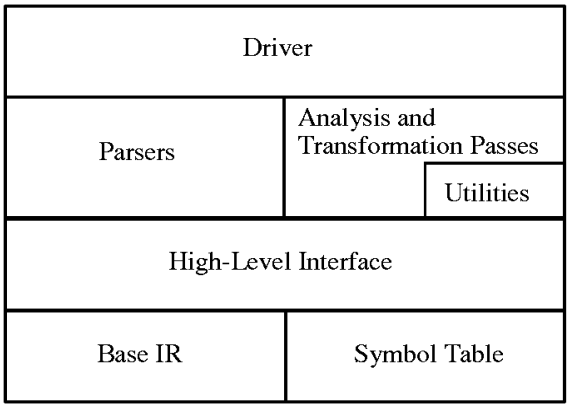

Fig. 1. Cetus components and interfaces: Components of Cetus only call methods of the components beneath them. The driver interprets command-line arguments and initiates the appropriate parser for the input language, which in turn uses the high-level interface to build the IR. The driver then initiates analysis and transformation passes. Utilities are provided to perform complex operations in order to keep the base and interface as uncluttered as possible.

interfaces with Cetus' Java code. Extending Cetus with front ends for other languages is discussed in Section 3.4.

Instead of implementing our own preprocessor for the C language, we rely on the GNU preprocessor, cpp. Preprocessing accessible through GCC with the -E option may be necessary for programs that rely on certain GNU extensions. The preprocessed files are given to the parser, which builds the IR.

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. The next section describes this architecture. 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 implementation is ready. These concepts had already proved their value in the implementation of the Polaris infrastructure – the Polaris *base* was rewritten three to four times over its lifetime while keeping the interface, and hence all compilation passes, nearly unmodified [7]. Cetus has a similar design, shown in Figure 1, where the high-level interface insulates the pass writer from changes in the base.

3 Implementation

3.1 IR Class Hierarchy

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 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 abstract classes, which represent generic control constructs. Generic passes can then be written using the abstract interface, while more language-specific passes can use the derived classes. We feel it is important to work with multiple languages at an early stage, so that our result is not simply a design that is extensible in theory but also in practice. Toward this goal, we have begun adding a C++ front end and generalizing the IR so that we can evaluate these design trade-offs. Preliminary work in this area is discussed below in Section 3.4. Other potential front ends are Java and Fortran 90.

3.2 Navigating the IR

Traversing the IR is a fundamental operation that will be used by every compiler pass. For a block-structured IR, one important question is whether to support *flat* or *deep* traversal of the statement lists. In a flat traversal the compiler pass steps through a list of statements at a specific block level, where each statement is either simple or compound. Moving into inner or outer blocks must be done by explicitly inspecting the type of a compound statement. By contrast, deep traversal would visit each statement and expression in lexical order, regardless of the block structure. Deep traversal is useful for tasks that need to inspect

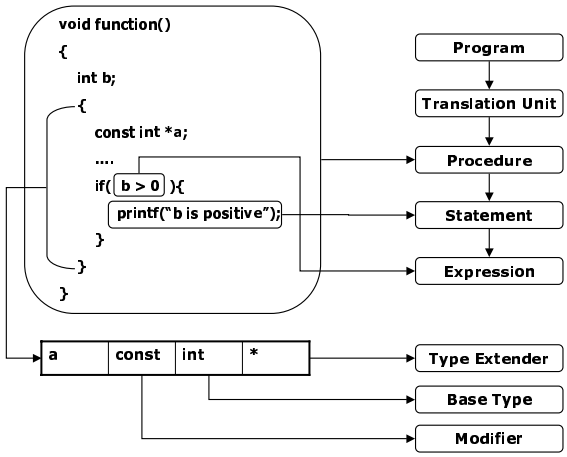

Fig. 2. A program fragment and its IR in Cetus. IR relationships are similar to the program structure and a symbol table is associated with each block scope.

all expressions, independent of the statements they are in. An example is flow-insensitive analysis of defined and used variables in a procedure. Flat traversal is needed by all passes whose actions depend on the type of statements encountered. Most passes belong to this latter category. Therefore, the Cetus base supports flat traversal.

The IR-API is the interface presented by Cetus' base. In general the Cetus base is kept minimal and free of redundant functionality, so as to make it easy to learn about its basic operation and easy to debug. Cetus also provides a utility package, that will offer 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. Deep traversal is an example of a utility function.

3.3 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 sequential job in nature. We use a list structure to hold type information so that types could be easily understood 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 repository of type information for a variable which is declared in a certain scope. As a result, scope must always be considered when dealing with symbols. In C, a block structure serves as a scope. Therefore, structs in C are also scopes and their members are represented as local symbols within that scope. Normally a compiler uses one large symbol table with hashing to locate symbols [6], but 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 [8], 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. Another benefit is reducing symbol table contention during parallel parsing, which we discuss in Section 5.1.

3.4 Extensions

Cetus is designed to handle additional languages. We have begun adding support for C++ and plan to add support for Java. Cetus' IR can represent expressions and statements in these languages with the addition of new IR nodes to represent exceptions. The type system supports user-defined types that can include both data and functions. Coupled with the distributed symbol table, Cetus can represent classes and their inheritance relationships.

Additional analysis and transformation passes are written using the same IR-API, so they can interoperate with existing passes. The standard IR-API makes common operations among passes more obvious, and the most useful operations can be moved into the utilities module. Future passes then become easier to write because they can make use of the new utilities, and the cycle continues.

Parsing was intentionally separated from the IR-building methods in the high-level interface so that other front ends could be added independently. Some front ends may require more effort than others. For example, writing a parser for C++ is a challenge because its grammar does not fit easily into any of the grammar classes supported by standard generators. The GNU C++ compiler was able to use an LALR(1) grammar, but it looks nothing like the ISO C++ grammar. If any rules must be rearranged to add actions in a particular location,

it must be done with extreme care to avoid breaking the grammar. Another problem is C++ has much more complicated rules than C as far as determining which symbols are identifiers versus type names, requiring substantial symbol table maintenance while parsing [22].

We are extending Cetus for C++ by using a Generalized LR (GLR [1]) parser generator [23]. Such parsers allow grammars that accept any language and defer semantic analysis to a later pass. GLR support has recently been added to GNU Bison [9] and provides a way to create a C++ parser that accepts the entire language without using a symbol table [12]. An important benefit is the grammar can be kept very close to the ISO grammar. We have developed a parser for the complete C++ language plus some GCC extensions using Bison [2]. We believe it is due to the language's complexity that there are fewer research papers dealing with C++ than with other languages, despite C++'s wide use in industry. The above reasons should allow Cetus to provide an easy-to-use C++ infrastructure, making it a very important research tool.

4 Cetus Features

In this section we discuss a number of distinguishing features that may become important for users of this new infrastructure. They deal with debugging support, readability of the transformed source code, expression manipulation capabilities, and the parallel execution of Cetus.

4.1 Debugging Aids

One important aspect that makes an infrastructure useful is providing a good set of tools to help debug future compiler passes. Cetus provides basic debugging support through the Java language, which contains exceptions and assertions as built-in features. Cetus executes within a Java virtual machine, so a full stack trace including source line numbers is available whenever an exception is caught or the compiler terminates abnormally.

Furthermore, the IR-API is designed to prevent programmers from corrupting the program representation. For instance, the IR-API will throw an exception if a compiler pass illegally uses the same nodes for representing two similar but separate expressions. Internally, Cetus will detect a cycle in the IR, indicating an illegal operation. Other errors may be detected through language-specific semantic checks.

4.2 Readability of the Transformed Code

An important consideration is the presentation of header files in the output. Internally, header files are expanded, resulting in a program that is much larger

[1] Also called stack-forking or Tomita parsing.

[2] The parser is not written in Java, so it must interface with Cetus by writing the parse tree to a file.

than the original source. This form is more difficult to understand. By default, Cetus detects code sections that were included from header files and replaces them by the original `#include` directives. Similarly, undoing macro substitutions would make the output code more readable. However, the IR cannot store macro definitions and uses because it is constructed from preprocessed source code with macros already expanded. Even if the information were available, nested macros and macros that use unusual features, like token pasting, may make the inverse operation impossible. Therefore, Cetus prints the expanded macros as part of the output. Cetus also "pretty prints" the output with appropriate spacing and indentation, potentially improving the structure of the original source program, although comments are removed.

4.3 Expression Simplifier

The expression simplifier provides a very important service to pass writers. Our experience with the class project showed that source-to-source transformations implemented within GCC often resulted in large, unreadable expressions. GCC does not provide a symbolic expression simplifier and the students determined it would not be easy to add one given the problems we have mentioned with GCC's IR. The Cetus API, however, made it possible to add a powerful expression simplifier with a modest effort. While it is not as powerful as the simplifiers provided by math packages, such as Maple, Matlab, or Mathematica, it does reduce the expressions to a canonical form and has been able to eliminate the redundancy in the expressions we have encountered in our experiments. Expression simplification enhances the readability of the compiler output and enables other optimizations because it transforms the program into a canonical form. For instance, idiom recognition and induction variable substitution benefited most from the expression simplifier [19]. Recognition is easier because there are fewer complicated expressions to analyze and all expressions have a consistent structure. Substitution can create very long expressions that make later passes less effective unless it is immediately followed by simplification.

4.4 Parallel Parsing

Cetus is written in Java which is slower and requires more memory than C or C++. These factors contributed to Cetus taking a noticeably longer time to process its input than, for instance, the GCC compiler. The Antlr parser is reentrant, so we use Java threads to parse and generate IR for several input files at once. Some interesting observations about this approach appear next in the evaluation section.

5 Evaluation

One aspect of evaluating a compiler infrastructure is its efficiency in terms of run time and memory usage when dealing with realistic applications. Another aspect

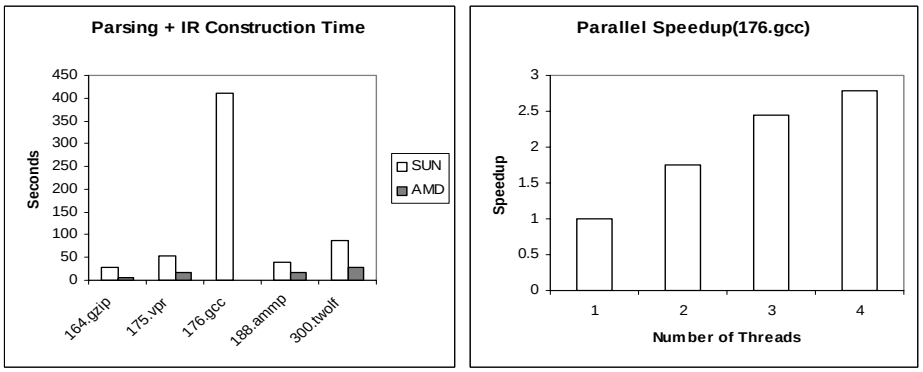

Fig. 3. Parse time and speedup *of the compiler* for some SPEC CPU2000 benchmarks. *SUN* is a four-processor, 480MHz Sun Enterprise 450 running Solaris and *AMD* is a two processor 1.533GHz AMD Athlon system running Linux. The JVM is version 1.4.1_01 from Sun Microsystems. Parse time for 176.gcc on Linux is not shown due to the gcc "statement-expression" extension (which Cetus does not yet support) used in code for that platform.

is looking at how practical it is to use for research projects. In this section, we deal with efficiency issues and in Section 6 we explain Cetus' role in research projects.

5.1 Parsing and IR Construction Time

Cetus is able to parse all of the SPEC CPU2000 benchmarks that are written in C. Parsing and IR construction time for some of them are shown in the left graph of Figure 3. Parsing time is not a serious issue for modestly-sized programs, but it can be a problem for large benchmarks like 176.gcc. On the SUN platform it requires 410 seconds to parse and completely generate IR. In this case, parallel parsing and IR generation is useful to reduce the time overhead. Cetus can parse multiple files at a time using Java threads and Antlr's reentrant parser. The right graph in Figure 3 shows speedup for 176.gcc using up to 4 threads on our SUN platform. Cetus does not show ideal speedup since symbol table accesses are synchronized.

5.2 Memory Usage

Efficient memory usage is important to source-to-source compilers because the entire program must be kept in memory for interprocedural analysis and transformation, requiring large amounts of memory. Figure 4 shows memory usage of Cetus.

The left graph shows the size of the Java heap after IR construction and the right graph shows the ratio of the Java heap size to preprocessed input source file size (without comments). All measurements were done on the SUN platform.

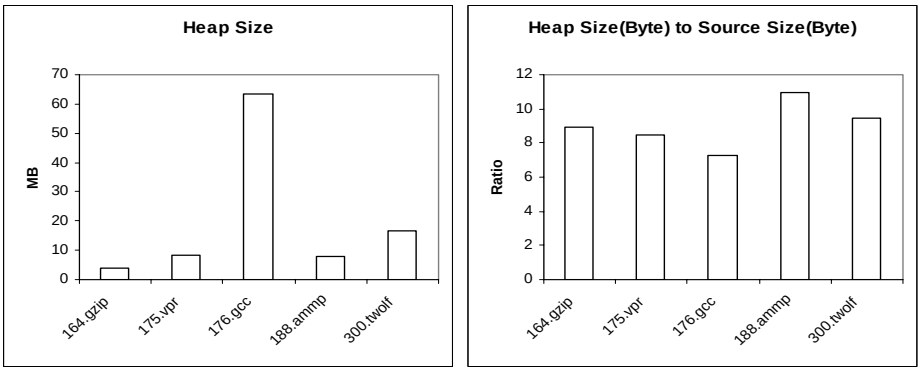

Fig. 4. Memory usage and efficiency of Cetus for some SPEC CPU2000 benchmarks.

Currently, Cetus requires around 10 times more memory compared to the input source size.

In addition to optimizing the usage of Java collection classes, such as using a smaller initial size for each collection to save memory, we applied the following improvements in order to reduce Cetus' working set. Taken together, these modifications reduced the memory requirement of 176.gcc from 250MB to 64MB.

Merging Header Files The largest reduction in memory usage was achieved by merging multiple uses of header files. Initially, if two different source files included the same header file, Cetus' IR would contain two copies of the symbol information for the header file. Using a single copy saved a lot of memory. Parallel parsing remains possible because Java hash tables are synchronized, so multiple parser threads entering symbols into the same symbol table do not interfere with each other. Only the first occurrence of a symbol is used.

Preventing Creation of Unnecessary Objects Another improvement was the elimination of temporary objects. To this end we rewrote the Cetus code so as to avoid the need for temporary data structures. This change reduced the memory usage and also resulted in speed improvements, primarily due to the reduced need for garbage collection.

Simplifying the IR Classes Some nodes in the IR were placeholders with no additional information. They only served to preserve the structure of the tree. Using the same object over again for each placeholder, we were able to reduce memory usage. Common IR nodes such as identifiers and expressions are used extensively so reducing the space needed to represent one of them has a large impact on the overall memory usage. Certain data members of these nodes were found to be unnecessary and were eliminated, or were able to be moved to nodes that occurred less frequently.

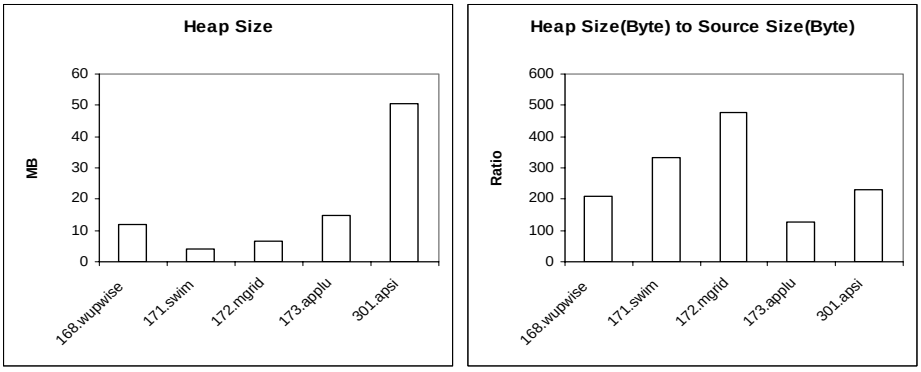

Fig. 5. Memory usage and efficiency of Polaris for some SPEC CPU2000 benchmarks

```
/*
 *  Get name of private variables from priv_list
 *  add a new Symbol for each private variable and
 *  replace all references to older Symbol
 *  "stmt" is a CompoundStatement
 */
for (i = 0; i < priv_list.size(); i++) {
        name = (String) priv_list.get(i);     // get the variable name
        orig_var = stmt.get(name);              // get older(existing) Symbol
        // add a new Symbol for Private variable
        priv_var = stmt.add(original_var.getTypeList(), name);
        // replace all reference to Symbol orig_var with  reference to Symbol priv_var
        stmt.replace(new IdExpression(orig_var),new IdExpression(priv_var));
}
```

Fig. 6. Code excerpt from an OpenMP to POSIX threads translator.

5.3 Comparing with Polaris

Figure 5 shows heap memory usage of Polaris after parsing and constructing IR on our SUN platform. Directly comparing heap sizes of Cetus and Polaris is difficult since they are written in different languages and work with different languages. However, comparing Figure 4 and Figure 5 indicates Cetus uses a reasonable amount of memory. It also seems likely that Cetus has a more memory-efficient IR than Polaris since its ratio of IR size to program size is an order of magnitude less.

Another interesting comparison is parsing speed. Cetus parses about 20K characters per second while Polaris handles 10K characters per second on average for the benchmarks in Figure 4 and Figure 5, running on the same system.

6 Using Cetus for Translation of OpenMP Applications

OpenMP is currently one of the most popular paradigms for programming shared memory parallel applications. Unlike MPI, where programmers only insert li-

brary calls, OpenMP programmers also use directives. Compiler support is required to recognize these directives and implement the OpenMP specification. In this section, we discuss a translator for OpenMP C applications implemented using Cetus, which demonstrates the strength and completeness of Cetus.

Compiler functionality for supporting translation of OpenMP falls into two broad categories. The first category deals with the translation of the OpenMP work sharing constructs to the micro-tasking format. This entails the extraction of the work sharing code to separate microtasking subroutines and insertion of the corresponding function calls and synchronization. Cetus provides an API sufficient for these transformations. The second category deals with the translation of the 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. Figure 6 shows a section of the code used to handle the *private* data clause in OpenMP.

There are currently two different OpenMP translators which have been implemented using Cetus. Both of these 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 [15]. Although the entire OpenMP 2.0 specification is not supported yet, the translators are powerful enough to handle benchmarks such as 330.art_m and 320.equake_m from the SPEC OMPM2001 suite.

7 Conclusion

We have presented an extensible compiler infrastructure, named Cetus, that has proved useful in dealing with C programs. In particular, Cetus has been used for source transformations on the SPEC OMPM2001 benchmark suite. The infrastructure's design is such that adding support for other languages, analysis passes, or transformations will not require a large effort. Preliminary work on extending Cetus for C++ was used as an example of how we have prepared Cetus for future growth.

Future work involves finishing other front ends and providing more abstractions for pass writers. We consider the high-level interface and the utility functions to be a kind of programming language for the pass writers. The motivation behind expanding and generalizing that language is the need to bring the amount of code written by a pass writer closer to the pseudocode they see in a textbook or research paper. By providing more ways to abstract away the details of the language and providing more high-level operations to the pass writers, large portions of the passes should become reusable. Starting to add other languages early in the development process is vital to proving this hypothesis.

References

1. Portland Group Homepage. *http://nci.pgroup.com.*

2. SUIF Homepage. *http://suif.stanford.edu*.
3. A. Appel, J. Davidson, and N. Ramsey. The Zephyr Compiler Infrastructure. 1998.
4. P. Banerjee, J. A. Chandy, M. Gupta, et al. The PARADIGM Compiler for Distributed-Memory Multicomputers. *IEEE Computer*, 28(10):37–47, October 1995.
5. W. Blume, R. Eigenmann, et al. Restructuring Programs for High-Speed Computers with Polaris. In *ICPP Workshop*, pages 149–161, 1996.
6. 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.
7. 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.
8. C. N. Fischer and R. J. LeBlanc Jr. *Crafting a Compiler*. Benjamin/Cummings, 1988.
9. Free Software Foundation. *GNU Bison 1.875a Manual*, January 2003.
10. Free Software Foundation. *GNU Flex 2.5.31 Manual*, March 2003.
11. D. L. Heine and M. S. Lam. A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector. *PLDI*, 2003.
12. W. Irwin and N. Churcher. A Generated Parser of C++. 2001.
13. S. C. Johnson. Yacc: Yet Another Compiler Compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.
14. M. Lesk and E. Schmidt. Lex-A Lexical Analyzer Generator. Technical report, AT&T Bell Laboratories, 1975.
15. S.-J. Min, A. Basumallik, and R. Eigenmann. Supporting Realistic OpenMP Applications on a Commodity Cluster of Workstations. *WOMPAT*, 2003.
16. 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.
17. T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator. *Software - Practice and Experience*, 25(7):789–810, 1995.
18. 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.
19. B. Pottenger and R. Eigenmann. Idiom Recognition in the Polaris Parallelizing Compiler. *International Conference on Supercomputing*, 1995.
20. R. M. Stallman. *GNU Compiler Collection Internals*. Free Software Foundation, December 2002.
21. R. M. Stallman. *Using and Porting the GNU Compiler Collection*. Free Software Foundation, December 2002.
22. B. Stroustrup. *The C++ Programming Language - 3rd Edition*. Addison-Wesley, 1997.
23. M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1986.
24. 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.