Journal of Information & Optimization Sciences
ISSN 0252-2667 (Print), ISSN 2169-0103 (Online)
Vol. 44 (2023), No. 3, pp. 407–416
DOI : 10.47974/JIOS-1350

Source-to-source translation for code-optimization

K. R. Chowdhary *
Rajendra Purohit †
Department of Computer Science and Engineering
JIET Jodhpur
Jodhpur 342008
Rajasthan
India

Sunil Dutt Purohit §
Department of HEAS
RTU Kota
Kota 324010
Rajasthan
India

Abstract

Multi-core design intends to serve a large market with user-oriented and highproductivity management as opposed to any other parallel system. Small numbers of processors, a frequent feature of current multi-core systems, are ideal for future generation of CPUs, where automated parallelization succeeds on shared space architectures. The multi-core compiler optimization platform CETUS (high-level to high-level compiler) offers initiates automatic parallelization in compiled programmes. This compiler's infrastructure is built with C programmes in mind and is user-friendly and simple to use. It offers the significant parallelization passes and also the underlying empowering techniques, allows source-to-source conversions, and delivers these features. This compiler has undergone numerous benchmark investigations (techniques) and approach implementation iterations. It might enhance the programs' parallel performance. The main drawback of advanced optimising compilers, however, is that they don't provide runtime details like the program's input data. The approaches presented in this paper facilitatedynamic optimization using CETUS. The large amount of proposed compiler analyses and modifications for parallelization is the last

* *E-mail:* kr.chowdhary@gmail.com (Corresponding Author)
† *E-mail:* rajendra.purohit@jietjodhpur.ac.in
§ *E-mail:* sdpurohit@rtu.ac.in

point. To research the behaviour as well as the throughput gains, we investigated both non-CETUS based and CETUS based parallelized program features in this work.

Subject Classification: *68N20 Theory of compilers and interpreters.*

Keywords: *CETUS (high-level to high-level compiler), Non-CETUS, (IR) Internal Representation.*

1. Introduction

The high-level-to-high-level compiler CETUS (which adds automatic parallelization in built programs) offers a platform for study on multi-core compiler optimizations. The architecture in this compiler is user-friendly and designed with C programs in mind. It offers the key parallelization passes, the underlying enabling mechanisms, and source-to-source transformation capabilities. The most sophisticated research infrastructure for compiler optimization on parallel processors may have been the CETUS infrastructure, which was built on Polaris. We have covered parallelization optimization method in this paper. Reduction-Substitution, Recurrence-Substitution, Induction-Variable-Elimination, Scalar-Expansion, Forward-Substitution, Strip-Mining, and Loop-Exchange are some of the methods examined [1], [4]. The nearly 15,000 lines of code (in Java) that make up CETUS's 1,500-line-long intermediate representation (IR) are used to execute analysis and transformations at the source level [2].

2. CETUS Components

CETUS includes a C-parser, an Internal Representation (IR), as well as a set of analysis and transformation stages, with an emphasis on automatic parallelization. This section gives an overview of the many passes that are available right now by the CETUS compiler as well as the Internal Representation [1], [3] & [4].

2.1 *IR in CETUS*

IR is implemented in the same manner as the Java class hierarchy in CETUS. A high-level vision offers the pass writer with a syntactic picture of the source program that is simple to comprehend, access, and alter the input 2.

Figure 1 shows, how a program in the CETUS program operates. Class type in the program stands in for the complete program, which may include numerous source files. A Translation Unit is used to represent

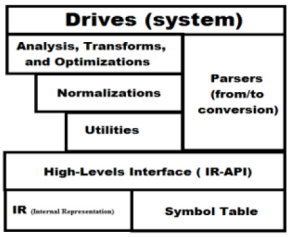

Figure 1

How a program work in CETUS

every source file. Complete (adequate) data abstraction prevents pass writers from modifying the IR in any other ways than through access functions [1], [4].

The input program in CETUS is demonstrated in figure 1. Pass writers can only alter the IR via access functions because there is total data abstraction. These are some of the IR's key characteristics: movable things IR objects in CETUS are descended from the basic class "Traversable." The power to generically iterate over lists of things is provided by the traversable class [12].

2.2 *Symbol chart*

Information on identifiers and data kinds is available from the symbol table interface. Declarative statements are used to hold data that is directly used by symbol tables. There is no duplicate and separate storage for symbol tables.

Annotations: Annotation objects can be used to store remarks, pragmas, instructions, and other supporting data about IR objects. A statement might be connected to an annotation.

How to tie a C program to the CETUS process in the original Figure 2 is how the IR is organized. It is possible to duplicate [12] source programs, but this presents a problem for source-to-source translators. The user will find it simple to identify the transformations made by the compiler if the IR/output are kept comparable to the input. In contrast, maintaining the IR language independence results in simpler compiler architecture but may prevent the output from being created from the source code in its original form. Since the structure of CETUS statements and expressions is similar to that of the C programming language, translating from one source to another is a straightforward process. In figure 2, it is demonstrated how syntax and IR correspond.

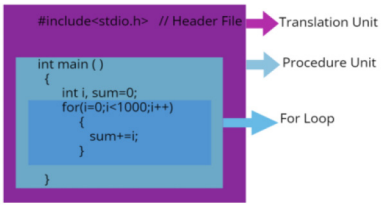

Figure 2

How to relate a C-program to CETUS procedure

2.3 *Transformations and Analysis*

The CETUS compiler's primary feature is automated parallelization, and many of its passes were created to accommodate it. Good automatic parallelization requires the use of program analysis and transformation techniques. These methods progress from generic analysis to strategies tailored to parallelization, including privatization. This paper discussed the implementation of passes for two types of techniques: general passes and parallelization passes [5], [6] , in figure 1. Automatic parallelization is supported by general passes as an enabling approach [5].

Using symbols to manipulate: Like its forerunner Polaris, CETUS ability to manipulate symbols is a vital characteristic that allows it to represent program symbolically. When converting a program, the capacity to handle symbolic expressions is crucial 3 Common passes.

Parallelism is successful Symbolic evaluation Identification and replacement of induction variables Analysis of points-to/aliases Recognition of and transformation of reduction Workflow in-lining Privatization using scalar/array Analysis of the USE/DEF chain Analysis

Table 1

Automatic parallelization passes versus general passes [5], [6]

General Passes	Parallelization passes
Symbolic Analysis Points-to/alias analysis **[5], [6]**	Induction Variables recognition / substitution Reduction recognition / transformation **[5], [6]**
Function In-lining USE/DEF chain analysis **[5], [6]**	Scalar/array privatization Data dependence analysis **[5], [6]**
Miscellaneous Loop	Parallelizer

```
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 EVALUATION)
```

Figure 3

Types of mathematical expression

of data dependence Differential Loop At the source level, a Parallelizer to simplified form is used in program analysis and optimization techniques as mentioned in Figure 3 and Table 1 [5].

Through the provision of a set of utilities that streamline and normalize the symbolic expressions, CETUS offers such expression manipulation. The few examples highlight the simplifying benefit in CETUS's capabilities.

Analysis of an array section: It describes the collection of array (range) elements that program statements can access. Compiler passes that have array-section analysis enabled are more precise and effective than others.

By calculating the value ranges of the array-subscripts, array-section analysis in CETUS provides a USE/DEF analysis of array variables for a program. All array-subscript expressions must be simplified before array section analysis is carried out inside a loop. If the analysis is applied to blocks of code containing many statements, these array portions are grouped together. The output of array section analysis is displayed in the following code example as a pragma annotation for the specified loop:

```
1   c =2;
2   N= 100;
3  %#pragmaCETUS USE(A [0:100] [0:100]) DEF (B [1:99] [1:99])
4   for (i=1; i<N; i++)
5   {
6      for (j=1; j<N; j++)
7      {
8         B[c*i -i][j] = (A[i-1][j]+A[i+1] [j]+A[i][j-1]+A[i][j+1])/4;
9      }
10  }
```

Analysis of data dependence: The data dependence analysis generates a data dependence graph by collecting dependence information (data) for array accesses within loops and nests. This study describes dependencies between data references and gives data references that, when used in a program, all visit the same memory location. In order to

examine the dependencies between array accesses and find solutions to systems of equations, data-dependence is used.

Range analysis: Range analysis determines the value ranges of integer variables at each step in the programming and then provides a map that connects each statement to the different value ranges that were valid prior to each statement. Together with utility functions, this range repository gives other passes access to information about symbolic terms, such as the boundaries of variables, expressions, and it uses symbolic comparison to evaluate whether one expression is more semantically significant than another. For instance, based on the value range of the equation $vv_1 + vv_2$ is [-10, 10], our range analysis framework may determine that $vv_1 + vv_2 = vv_3$ at a program point where vv_1 = [0, 20], $vv_2 = vv_1 - 10$, and $vv_3 = -10$.

Passing parallel transformations: Privatization and reduction variable recognition are the fundamental parallelizing transformation strategies of CETUS, which have been found to be the most effective for automatically parallelizing compilers [1], [5].

Privatization: A crucial step that an automatic Parallelizer must complete is identifying private variables within a loop. Private array variables have temporary homes in array sections. The data-dependence analyzer can reasonably presume that these variables don't have dependencies because they don't have to be made available to the other threads at runtime. The array privatizer in CETUS transitions from the most inside to the most exterior of the loop while traversing a nested loop that contains symbolic terms. The array privatizer then uses traversal to find private variables in step 4. When applying a symbolic analysis technique while range-analysis executes array-section operations, this privatizer's accuracy increases. The privatizer often computes big must-defined sections, and if the expression comparison tool can determine that n=m, the intersection of the two must-defined sections, [1: m] n [1: n], yields [1: n] rather than [1: min (m, n)].

Recognition of the reduction variable: It is employed in a variety of contexts and typically accepts the expression $S_v = S_v$ + expr. Recognizing in multiple loops requires auto parallelizing, which is crucial to success. On a reduction operation, a data-dependency will produce a dependence log. The following requirements must be met in order for the CETUS reduction variable analyzer to identify additive reduction variables:

- The loop comprises one or more assignment expr of the type $S_v = S_v$ + expr, where S_v is a scalar variable or array access and expr is often a real-valued, loop-variable expression; and
- S_v does not seem in the loop, anywhere.

3. Practical Evaluation

We evaluate CETUS using two methods: providing its own qualities and reviewing the state-of-the-art in automatic Parallelization's while contrasting CETUS with the properties of CETUS parallelized program.

Code: Placed in Normal Sample.c

```
1   #include <stdio.h>
2   int main ()
3   {
4       int i;
5       int sum = 0;
6       for (i   = 0; i < 10000; i++)
7       {
8           sum += i;
9       }
10  }
```

Code: Placed in Cetus Performed Sample.c

```
1   #include<stdio.h>
2   int  main()
3   {
4       int i;
5       int sum = 0;
6       int _ret_val_0;
7       \#pragma cetus private (i) //this is an annotation internally used by Cetus
8       \#pragma loop name main#0 //this is an annotation internally used by Cetus
9       \#pragma Cetus parallel //this is an annotation internally used by Cetus
10      \#pragma omp parallel for private (i) // this loop is parallelized with OpenMP
11      for (i=0; i < 10000; i++ )
12      {       //Cetus retained the empty loop. Removing them is left to low-level compilers, such as gcc or
13              icc
14      }
15      sum= constant;
16  }
```

4. Ability to Write Passes using a Graph

For pass authors, CETUS provides a lot of features that are helpful; the equation below discusses classes that support program analysis, normalization, and change.

Call Graph: A Call-Graph class is provided by CETUS that generates a call graph from a Program object. Callers' are mapped to callees in the call graph, and callees are mapped to callers. To find out if a process is recursive or a leaf of call graph, a pass can query the call graph.

A control-flow graph can be created from a program object using the Control-Flow Graph class that CETUS offers. The main building pieces of each method are represented as edges in the graph, which connect them.

4.1 *Changing the Program*

Annotation System: The CETUS IR includes a class called Annotation that is derived from the Declaration class in general. Annotations can appear inside or outside of procedures because they can appear anywhere declarations can. They serve as a database for information sharing between passes and can be used to enter comments, pragmas, raw text, or a hash map.

Inserting New Code:- The CETUS-IR's constructors are available for usage in all of the statement and expression classes. The parser uses these constructors to produce the IR for the program. As a result, pass writers have the same ability as the parser to insert new IR. Figure 4.

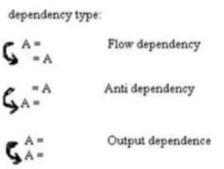

Figure 4

Dependency Types

The IR for the original code is created using dependency types. The IR classes' constructors and other methods examine their parameters to make sure the IR is always consistent. An exception is raised, for instance, if a duplicate symbol is going to be produced or if the same IR object is attempted to be placed in two different locations within the IR tree. A program is a set of statements or a directed set of expressions, the scheduling and ordering of which are determined by dependencies [13]. Dependencies can be roughly divided into two groups:

- Data dependencies occur when statements compute data that is subsequently needed by other statements.
- Control Dependencies: They result from the program's orderly control flow. Drawing edges that link dependent activities together creates a dependence graph.

These arcs impose a partial ordering on operations, preventing a program from running entirely concurrently. Although use-definition chaining is a type of dependency analysis, the estimations of data reliance it

$$X = A + B + C; \quad (1)$$
$$Y = E + F + G; \quad (2)$$
$$Z = E + Y; \quad (3)$$

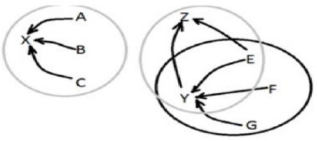

Figure 5

Equation allows the user to locate two sets of concurrent tasks

produces are unduly conservative. Statement number I may be dependent on a common control channel in one of four ways, mentioned in figure 5.

- Flow-Dependence: M_j is flow-dependent on M_i if available variable that M_j is using was calculated by M_i.
- Anti-dependence: M_j is anti-dependent on M_i if M_j re-calculate value of a variable utilized by M_i.
- Output-Dependence: If two programs compute similar variable and target value of variable from M_j must be saved subsequently that from M_i then M_j is output-dependent on M_i.
- Control Dependency: M_j is control-dependent on a conditional statement, depending on the execution of each task and the subsequent path (M_i must execute before M_j).

5. Conclusion

The strategies presented in this research utilize CETUS to facilitate dynamic optimization and auto parallelism. There is also a large number of suggested compiler analyses and parallelization changes. By spotting dependencies within a program, it may be able to enhance the parallel performance of the programs and increase output overall. In order to explore the behaviour as well as the increases in throughput using parallel data dependence graphs, we have analyzed the characteristics of both non-CETUS and CETUS parallelized programs.

References

[1] CETUS: A Source To-Source Compiler Infrastructure for Multicores in Computer", (January 2010). DOI: 10.1109 /MC.2009.385 @ Source: IEEE Xplore," by Chirag Dave et. al.

[2] Parallel Programming with Polaris", Computer, pp. 78–82, by W. Blume et. al. (December 1996).

[3] Portable Compilers for OpenMP, OpenMP Shared-Memory Parallel Programming", LNCS 2104, Springer Verlag, pp. 11–19, by S.-J. Min et. al. (2001).

[4] Automatic Parallelization with Cetus", High-Performance Computing Laboratory, Purdue University School of Electrical and Computer Engineering, by Bae H. et. al. (2008).

[5] The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation", Int J Parallel Prog, 41, 753, Springer Verlag Media, LLC (2013), doi:10.1007/s10766-012-0211-z 2012, by Bae, H. et. al.

[6] Automatic Parallelization Tools", Proceedings of the World Congress on Engineering and Computer Science, Vol I, San Francisco, USA, October 24-26, 2012, by Ying Qian (2012).

[7] Programming Language Design and Implementation", (PLDI 06), Proceedings of the 2006 ACM SIGPLAN Conference, Practical Runtime Monitoring of Applications for Execution Anomalies, ACM Press, pp. 84–95, by L. Fei et. al.

[8] The Open Transactional Application Programming Interface", IEEE CS Press, pp. 376-387, Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 07), by W. Baek et. al. (2007).

[9] Parallelizing Irregular C Codes Assisted by Interproceduralhape Analysis", Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 08), IEEE Press, pp. 1–12, by R. Asenjo et. al. (2008).

[10] Optimizing Irregular Shared-Memory Applications for Distributed-Memory Systems," in Principles and Practice of Parallel Programming (PPoPP 06), Proceedings of the 11th ACM SIGPLAN Symposium, ACM Press, pp. 119–128, by A. Basumallik et. al. (2006).

[11] Parallel Architectures and Compilation Techniques", In PACT '06 Proceedings of the 15th International Conference on, Fast, Automatic, Procedure-Level Performance Tuning, Pages 173–181, by Zhelong Pan et. al.

[12] Experiences in Using CETUS for Source-to-Source Transformations", 17th International Workshop on Languages and Compilers for HP-Computing, USA, pp. 114, Springer-Verlag Berlin Heidelberg 2005, by Troy A. Johnson et. al.

[13] Implementation of Deadlock Analysis in Data Flow Graphs", *Journal of Information and Optimization Sciences*, 13:1, 7383, DOI:10.1080/02522667. 1992. 10699093, by Bruce Brocka (1992).