Reprinted from the

Proceedings of the GCC Developers' Summit

July 18th–20th, 2007
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
C. Craig Ross, *Linux Symposium*

Review Committee

Ben Elliston, *IBM*
Andrew J. Hutton, *Steamballoon, Inc., Linux Symposium, Thin Lines Mountaineering*
Janis Johnson, *IBM*
Mark Mitchell, *CodeSourcery*
Toshi Morita
Diego Novillo, *Red Hat*
Gerald Pfeifer, *Novell*
C. Craig Ross, *Linux Symposium*
Ian Lance Taylor, *Google*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

Using Backward Dynamic Program Slicing to Isolate Influencing Statements in GDB

Árpád Beszédes, Tibor Gyimóthy, Gábor Lóki
Department of Software Engineering, University of Szeged, Hungary
{beszedes,gyimi,loki}@inf.u-szeged.hu

Gergely Diós, Ferenc Kovács
Department of Software Engineering, University of Szeged, Hungary
{Dios.Gergely,Kovacs.Ferenc.5}@stud.u-szeged.hu

Abstract

Program slicing is a program analysis technique initially introduced to assist debugging, based on the observation that programmers mentally form program slices when they debug and understand programs. Namely, only those statements need to be investigated that actually influenced the erroneous value, and eventually, these statements constitute the *backward dynamic program slice*. An efficient algorithm to compute such slices has been implemented in the GCC/GDB environment, which adds a new `slice` command to retrieve the slice for a given program entity. In this paper, a background on program slicing is given, followed by the details of implementation. The dependences are computed after 'gimplification' in GCC, while STABS format is used to transfer them to GDB. The initial experimental results are presented as well.

1 Introduction

Debugging is a methodical process of finding and removing failures in a computer program. One of the most difficult steps in this process is isolating the source of the bug after a failure has been observed. The symptoms of a bug hardly ever bear any clues about the actual source of the problem. So, locating the cause of a bug is a cumbersome task for programmers. Unfortunately, most modern debugging tools do not provide any sophisticated means to aid this process. Generally, a simple "step-by-step" or maybe a "binary search" approach is applied for this task. Although it has some nice features in this respect (like *reverse execution*), the situation is similar with the GDB debugger as well.

In the scientific world, however, this issue has been extensively researched during the past decades. Remarkable results have been reported within the *program slicing* community, for example. Program slicing is an analysis technique for extracting parts of a program which represent a specific sub-computation of interest. It has been originally introduced by Weiser [13] to assist debugging, in which case a set of program points is sought which affects the variables of interest at a chosen program point, called the *slicing criterion*. The reduced program is called a *slice*. This definition is more precisely referred to as *backward slice*, since it associates a slicing criterion with a set of program locations whose earlier execution affected the value computed at the criterion (as opposed to *forward slice*, which is a set of locations that are affected by the criterion). Slicing can be categorized as *static* or *dynamic*. In static slicing, the input to the program is unknown and the slice must therefore preserve meaning for all possible inputs. By contrast, in dynamic slicing, the input to the program is known, and so the slice needs only preserve meaning for the input under consideration. Hence, by using backward dynamic slicing in debugging, the program parts which influence a program point where an error has been manifested in a specific execution of the program can be isolated.

One of the reasons why dynamic slicing is not yet present in leading debuggers is the lack of efficient algorithms to perform this operation. The algorithm introduced by the authors of this article is a promising one for this application, since it is more efficient than the previous approaches [3, 4, 7]. We started the implementation of the algorithm in the GCC/GDB environment, currently having a fully operational version for C programs. In this paper, we elaborate on the algo-

rithm and the details of implementation, and give initial experimental results. These are promising, but there is still plenty of work to be done: stabilization, testing of other front ends and the scalability, algorithm variations and enhancements, other user features and integration into popular debugger GUIs. The source code is accessible from our web page at `http://www.inf.u-szeged.hu/opensource/`. If this feature is welcomed by the community, an official development branch could be created to attract further volunteer developers, and to possibly integrate it into a future official release.

The paper is organized as follows. In the next section we first overview the dynamic slicing technique and the method implemented. Section 3 contains all the details of implementation in GCC/GDB. In Section 4, we overview the current status and initial experimental results, while Section 5 deals with the ongoing work in the project.

2 Program slicing in debugging

One of the most frequently cited applications of the program analysis technique of slicing is program debugging. A *backward program slice* of a program consists of all statements and predicates that might affect the variables in a set V at a program point p [13]. A slice may be an executable program or merely a subset of the program code. In the first case, the reduced program may be executed, and its behavior with respect to a variable v and a program point p is the same as that of the original program. In this paper, we are concerned with using slicing methods in program debugging. During debugging we generally investigate the program behavior under the test case that revealed the error, not under any generic test case. Therefore, dynamic slicing methods are more appropriate for this task than static ones. We are not interested in executable slices either.

By using dynamic slices, certain statements can be ignored in the process of localizing a bug. For example, consider the program in Figure 1. If it is run with input ⟨a=1⟩ and an erroneous value of z is detected at line 8, a backward dynamic slice can help narrow the possible causes for this error. Namely, only the shaded statements contributed to the value in question, and these will eventually constitute the backward dynamic slice.[1] A dynamic slice includes the statements that have direct or indirect influence on the statement in question. "Influence" can basically mean two different things: data- and control dependence [12]. Informally, data dependence means data flow to the observation point, while control dependence is the influence on the mere execution of the dependent statement. Note, that with a different execution of the program the result may be a different set of influencing statements. For example, if the input were ⟨a = 2⟩, statement number 6 could contribute to the value of z, but in this case statement 3 could not.

```
1  read(a)          read(a)
2  y=0               y=0
3  x=1              x=1
4  while(a>0)       while(a>0)
5    y=x              y=x
6    x=2              x=2
7    a=a-1            a=a-1
8  z=y              z=y
```

Figure 1: An example program and a dynamic slice

Dynamic slices are computed with respect to the *dynamic slicing criterion*, which is $(\mathbf{x}, i^j, V)$, where $\mathbf{x}$ is the program input, i is the statement number where the error is manifested at execution step j, and V is a set of variables which have faulty values at i [3]. In the previous example, the slicing criterion was $(\langle \mathtt{a}=1\rangle, 8^9, \{\mathtt{z}\})$. 2Step j is required since different slices can be obtained for the same statement at different occurrences during the execution (for example, the slices for y at statement 5 are different in different iterations of the loop). As a concrete scenario in a debugging session, the programmer sets a breakpoint at the slicing criterion (the point where the error has been manifested) and executes the program. By computing the backward dynamic slice at the breakpoint, a set of influencing statements is attained, which can be of great help in localizing the bug.

Various approaches have been proposed to compute dynamic slices, e.g. [2, 3, 8, 9]. In previous work, we elaborated on different algorithms for this task, which significantly differ from previous approaches [3, 4, 7]. We use specialized data structures based on program dependences, which enable different usage scenarios with optimal space and/or time requirements, instead of having a common program representation as the previous

[1] The reader may notice that the error could have also been caused by a bug in statement number 7, but it is not part of the dynamic slice according to the original definition. This issue is addressed by the so-called *relevant slices* [7], which we plan to deal with in the future.

approaches suggested (cf. the Dynamic Dependence Graph by Agrawal and Horgan [2]). For the purpose of debugging two strategies are possible. One may compute the backward dynamic slice of interest by tracking back the dependences starting from the criterion (called the *demand driven* approach), but we can also globally compute all occurring dependences (and slices) in a forward fashion as the execution of the program advances (*global method*) [3]. In the current work we apply the global method, but the demand driven algorithm could also be tried to compare its efficiency attributes (this is planned in the near future). In the following, we will overview this algorithm on a principal level, while in the next section we will deal with concrete implementation issues in the GCC/GDB environment.

The algorithm operates on two data sets: the execution history (list of statements executed for the run under consideration) and the concise static representation of the program. It processes the execution history in a forward way (i.e., after each executed statement certain required calculation is done) to follow dynamic dependences. The static representation of the program needed by the slicing algorithms is called the *D/U program representation*. It captures local definition-use relationships between the variable occurrences within each statement. For the time being we will assume that each statement defines one variable and uses zero or more variables. A statement of a program has the following D/U representation:

$$i.\ d_i :\ U_i\,,$$

where i is a statement serial number. The defined variable at the ith statement is $d(i) = d_i$, while $U(i) = U_i$ is used to denote the use set that is utilized for computing the value of d_i. A useful property of our approach is that by using the same D/U representation, we are able to capture not only the data dependences but the control dependences as well, which significantly simplifies the algorithm. Namely, each d_i defined and $u_k \in U_i$ used variable can have a special meaning that we call a *predicate variable*. Predicate variables are virtual variables that are not part of the program, but are generated for each predicate instruction in the program (these are the conditional branching instructions like 'if' and 'for'). Predicate instructions determine the control dependences among the statements, and we can treat the corresponding predicate variables as regular variables that can serve both as defined and used ones. More precisely, if statement i is a predicate instruction, a generated predicate variable p_i will be the defined variable at i, $d(i) = p_i$. Furthermore, for any statements i' its use set $U(i')$ will be extended with a corresponding predicate variable for each predicate instruction on which i' is directly control dependent. Our example has the D/U representation as follows:

i		$d : U$
1	`read(a)`	$a : \emptyset$
2	`y=0`	$y : \emptyset$
3	`x=1`	$x : \emptyset$
4	`while(a>0)`	$p : \{a\}$
5	`  y=x`	$y : \{x, p\}$
6	`  x=2`	$x : \{p\}$
7	`  a=a-1`	$a : \{a, p\}$
8	`z=y`	$z : \{y\}$

The conceptual algorithm is shown in Figure 2, where $LS(v)$ denotes the statement number at which variable v has been last defined.

program GlobalAlgorithm($P, \mathbf{x}$)
input: P : a program
$\mathbf{x}$: a program input
output: backward slices for all $(\mathbf{x}, i^j, U(i))$ criteria
begin
1 Read execution history
2 **for** $j = 1$ **to** steps executed
3 i := statement at the jth step
4 $DynDep(d(i))$:=
$\bigcup_{u_k \in U(i)} \big(DynDep(u_k) \cup \{LS(u_k)\}\big)$
5 $LS(d(i))$:= i
6 Output $DynDep(d(i))$ as the backward dynamic slice for crit. $(\mathbf{x}, i^j, U(i))$
endfor
end

Figure 2: Global algorithm for backward dynamic slices

In essence, the algorithm computes the dynamic slice for a given variable defined at the current execution step ($DynDep(d(i))$) based on the previously computed dynamic slices for variables used at that point ($DynDep(u_k)$). The statements at which the used variables have been defined last are also added to the resulting slice set. The *LS* point for the defined variable is naturally set after the slice has been computed, for we are interested in the previously existing dependences at

the actual step. If the algorithm reaches a breakpoint at some j (or the end of the execution) the actual slices for all defined variables are instantly available. More details on the algorithm can be found in the cited works.

The algorithm requires different extensions in order to be applied to a real programming language and environment. In the current version, the GCC/GDB implementation handles C, for which several issues had to be dealt with. The most important enhancement is the handling of data dependences that occur through different memory manipulations, like pointers and arrays. Namely, we always track the dependences on memory locations instead of individual variables, so any kind of data access is converted into memory addresses. Another important issue to be solved was the handling of interprocedural (cross-function) dependences. Some of these are more elaborated in our previous work on dynamic slicing C programs [4], while the concrete details regarding the present work are found in the following section.

3 Implementation in GCC/GDB

Our implementation approach of dynamic program slicing consists of two separate phases. The first phase collects the necessary static data for the second phase, which needs this data to run the algorithm. This static data is a set of D/U pairs for every single line of the source code. Naturally, this data is collected by GCC in the first phase (referred to as the *static phase*), and then it is transferred to GDB to run the global slicing algorithm (the *dynamic phase*).

The source code has to be compiled with the `-O0 -gstabs -fdu-analysis` flags in order to use the dynamic program slicing algorithm in GDB. We use the STABS debugging information format to communicate with GDB. So, every kind of optimization should be avoided during compilation. When GCC is executed with the given flags, it additionally generates the D/U information to the final assembly code. At this point, we have all the static information in the resulting object needed by GDB. When this kind of object is loaded in GDB, the D/U information is saved in the memory. When the user steps through an instruction, GDB computes all the dependences which are relevant in the currently executed source line. The resulting slices are stored in form of a set of lines in the original source code. The overall process is shown in Figure 3.

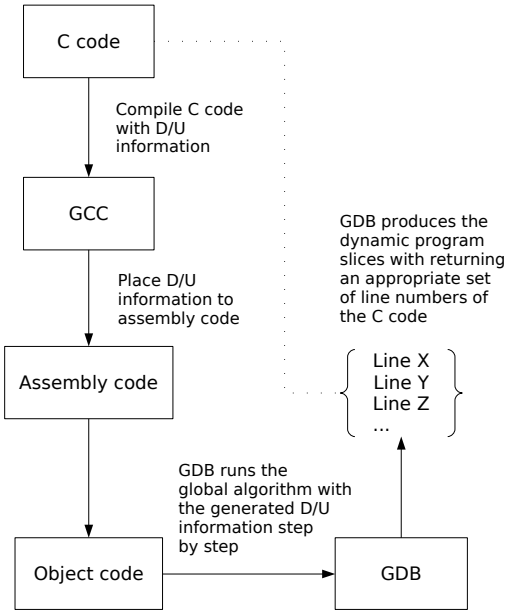

Figure 3: Architecture of dynamic slicing in GCC/GDB

3.1 GCC

Our D/U analysis algorithm is implemented as a new GCC pass. The new pass runs after gimplification, but before any optimization in `pass_all_optimizations`. It is run in the middle end, and not in the front end, in contrast with some other dynamic program slicing tools, such as "Spyder" [1]. In GIMPLE form there is no need to parse language dependent statements, as we should do if the algorithm was run in the front end.

3.1.1 Obtaining the D/U information

For obtaining the D/U information the algorithm walks through the CFG and for each GIMPLE statement a single D/U element is generated. Since a line of the source code mostly contains several GIMPLE statements, we need some additional processing at the end to produce the expected results. This whole process can be divided into the following basic steps:

- Find and store control dependences
- Find and store data dependences
- Eliminate temporary variables
- Stringify to assembly file

Slicing in GDB debugger is supported by the `-fdump-tree-du-analysis` GCC flag.

3.1.2 Control dependences

The main logic for discovering control dependences among basic blocks was borrowed from the DCE (dead code elimination) pass. We used `calculate_dominance_info` to find dominators in the CFG using the classical algorithm by Lengauer and Tarjan [10].

We can find all the control dependences between basic blocks with the algorithm described in [11], but this information should appear for each D/U pair. So, first all the control dependences between basic blocks are discovered by using the mentioned function already present in GCC. Later when the D/U pairs are collected from GIMPLE, the control dependences are appended to the D/U pairs (recall that our algorithm handles data dependence and control dependence uniformly). Control dependence relations appear as `T_PRED` pseudo-variablesin the list of used variables. These `T_PRED` pseudo-variables are defined for each conditional statement.

3.1.3 Data dependences

The algorithm processes GIMPLE statements recursively, and specifically the following language constructs are of interest:

`{VAR,PARM}_DECL` nodes: In our case these are the atomic items of the program (these are the `T_NORM` variables). The temporary variables introduced by gimplification are eliminated later (see Section 3.1.4).

`MODIFY_EXPR` nodes: We can distinguish defined and used variables with the help of these nodes. In a simple case, on the left hand side of such an expression, there is the defined variable and on the other side there are the used variables. In real programs however, there can be more defined variables for a single line. In GIMPLE, these definitions appear as different statements.

`{ARRAY,INDIRECT}_REF` nodes: In the case of these nodes instrumentation is necessary to help processing D/U information within GDB (see Section 3.1.5).

`{COND,SWITCH}_EXPR` nodes: If we find such a tree node in the program a `T_PRED` pseudo-variable is defined. The variables used to form the condition will be the used variables of this pseudo-variable.

`CALL_EXPR` nodes: These nodes need special attention. We must take care of the function itself and its arguments as well. The algorithm handles normal function calls, library function calls and function calls via pointers in different ways. GDB can retrieve the address of functions at run time, that is why we only store their name (like we did in the case of variables). The only exception is function pointers. At compile time, we do not know which function will be called at run time. That makes the instrumentation of function pointers necessary. Under library functions we mean precompiled functions, for which we cannot produce the D/U representation because they are not available. The only thing we can do is to make all the arguments of the function data dependent on each other. This is needed only when there is at least one argument passed by address. For each argument a `T_ARG` variable is generated, or `T_ARG_FP` for function pointer arguments.

`COMPONENT_REF` nodes: Our algorithm handles complex data structures in a conservative way. If one member of a structure is defined, the whole structure is defined and we stop recursion here.

`RETURN_EXPR` nodes: We define `T_RET` variables in the case of these tree nodes. These variables are needed to make the algorithm interprocedural.

3.1.4 Temporary variables

In GIMPLE, all expressions are in 3-address form. We should take care of the temporary variables holding intermediate values. This kind of variables should not appear in the output of our algorithm, that is why we have to eliminate them after we collected all the necessary information from the GIMPLE representation. The elimination is done by a simple recursive substitution of the used variables. We replace every occurrence of a temporary variable in all the used lists with its definition, starting with the first D/U pair.

3.1.5 Instrumentation

This is the only part of our algorithm which modifies the original code (additional instructions are inserted and

new variables are declared). The algorithm handles every element of a program in a general way. Namely, everything is treated as an address in the memory [4]. This will simplify the algorithm in GDB, since it will work always on addresses regardless of whether the dependences are to be computed between scalar variables, pointers, pointer dereferences, or whatever.

Generally, instrumentation is done at all points where a pointer or a pointer with an offset is dereferenced in the code (this includes any combination of pointer dereference, array indexing, field access and pointer arithmetic). If this offset is formed using a complex expression the whole expression (as a name) will be handled the same way as if it were an ordinary variable in the D/U representation. So we declare a new variable in these cases and store the result of this complex expression into it. GDB will then simply need to look up the address of this variable to find out the address of the complex expression involving memory accesses.

3.2 GCC/GDB communication interface

The communication interface between GCC and GDB is the object file itself, for which we used the STABS debugging information format. For each line the textual representation of the D/U pairs is written before the line number symbol. Figure 4 shows the grammar which defines the syntactical structure of our textual D/U representation.

```
dulist : duelem+
duelem : '(' duvar ','
         duvar+ ')'
duvar  : durole ':'
         dutype ':'
         duid
durole : R_DEF | R_USE
dutype : T_NORM | T_PRED
       | T_ENTRY | T_RET
       | T_DEREF | T_DEREF_FP
       | T_ARG | T_ARG_FP
       | T_OUT
duid   : [_0-9A-Za-z]+
```

Figure 4: Simple grammar for textual D/U representation

Each D/U pair is between brackets to support multiple D/U pairs per line. Each D/U pair consists of a defined variable and one or more used variables, and the variables are separated by commas. The defined variable is always the first, and must be present. Each variable has different fields separated by colons, depending on the variable's type, but the first two fields are always present. The first field defines the variable's role (`R_DEF` or `R_USE`), while the second field defines the variable's type. These types are described in detail in Section 3.3.3.

Each type has different characteristics and has different kinds of identifier fields. For example, the identifier field is a character string (name of a variable or function) in case of `T_NORM` or `T_RET` variables, while `T_PRED` pseudo-variables have their basic block index as the identifier field.

3.3 GDB

In the case of debugging a program compiled with `-O0 -gstabs -fdu-analysis`, GDB reads the D/U information immediately when loading the object file. We modified the `step` command to process the D/U information for the actual source line. So, when the user is debugging a program and calls the modified `step` command, the algorithm processes the D/U information. This is done after the actual line of the program code has been executed. After that, when the program execution is stopped, the user can obtain the dynamic program slices by a new command, called `slice`. Since we implemented the global algorithm overviewed in Section 2, all dynamic dependence sets are available and current.

Running the program in the traditional way is not possible because GDB does not know anything about the program when it is executed (with `run`, `step` or `next` the D/U information would not be processed). Instead, in our implementation of `run` we `step` over each instruction (or at least the code examined) using our modified `step` command.

3.3.1 Reading in the D/U information

Since we have chosen the STABS format, reading in the GCC generated D/U debug symbols is done by the STABS reader, which has been extended to handle our new STABS type, called `N_DUANAL`. GDB immediately reads all debug symbols when an object file is read, so the D/U information is available when the debug session begins. All D/U information is stored in

a hashtable using the duanal structure. It contains a file_id which is the string hash of the source file's name where the given symbol resides. An int for the source_line is enough, and we store the D/U itself in a char *. The file_id and the source_line make the D/U unique for later identification. When an object is loaded, the source file names are added to a hashtable, called source_files. We need this structure to look up which file the D/U information belongs to.

The textual D/U format described above is read using some special functions. The struct called variable contains the variable's name, lastdef (last defined line), its memaddr (memory address, stored as CORE_ADDR) and its dyndep (dynamic dependence set). name is a string and lastdef is a pointer to a dunal item. Finally, the dyndep dependence set is a hashtable which contains pointers to duanal items. These variables are stored in a global hashtable, called vardep.

3.3.2 Implementation of the Global algorithm

The algorithm overviewed in Section 2 is implemented in the following way. The input of the algorithm is the program itself, and the input of the program. Since the program is also the input of the GDB and the inputs are also user-given like in the case of a normal program execution, we do not have to care about these parts of the algorithm. The execution history is also given since we are working in a debug session. The output is the set of backward slices of the D/U variables which are stored in their dyndep fields. Computing the dyndep set of a DEF variable is a set union of the USE variables' dyndep and their lastdef. After that the DEF variable's lastdef is set to the line (and file) which the processed D/U info belongs to. The set union is implemented such that we simply check every element in the USE's set, and even if it is in the DEF's set already, we insert it. This can be done simply by hashtab functions from libiberty.

3.3.3 Variable types in GDB

In the following, the handling of different variable types in GDB is described.

T_NORM: There is nothing to do with a normal variable. It has its exact physical memory address during debugging, we just need to look up that address.

T_PRED: Predicate variables are pseudo-variables, which do not exist in the code. They represent control dependences in the program and their names are unique (generated by GCC). Since they must not have the memory addresses of any other variables, we use a predefined memory address which is taken from the code section and incremented for each virtual variable. When looking up a virtual variable's memory address we just simply look up the hashtable for it.

T_ENTRY: These are the top level predicate variables for each function. They are handled like T_NORM variables, where the variable is the function name.

T_RET: Return type is needed to bind variables when a function call is on the right hand side of an expression. These are handled similarly to T_NORM.

T_DEREF: When dereferencing a pointer as mentioned above, this GCC created variable contains the memory address of the dereferenced expression. So all we need to do is use this variable's value instead of its address.

T_DEREF_FP: This type is similar to T_DEREF, with the difference that it is used to handle function pointers.

T_ARG: An argument type is handled as a predicate type, if it refers to a normal variable. If it refers to a pointer, it is evaluated as a dereference type.

T_ARG_FP: This type is similiar to T_ARG. The only difference is that the function call is achieved by a function pointer.

T_OUT: Used in the cases when the function's return value is not used or it doesn't have any. The function and its arguments will be the used variables of this kind of pseudo-variables.

3.3.4 User commands

The most important user command is the one for obtaining slices, which is called slice. This is simply done by looking up the given variables' memory addresses and searching the corresponding dependence sets in the variable hashtable (recall that our algorithm stores globally all available slices). The command's parameter

could be a variable, or an expression which could be evaluated to a memory address. The slice output consists of the last defined line and the dependent lines, in the form `source_file.c:<line>`. Another important command is the modified step (in `infcmd.c`), `dyn-step`, which does a normal `step` augmented with processing the actual D/U information. Our modification is simply a lookup for the D/U information and a call to the processing function. A similar command, `dyn-next` is also available, furthermore a `dyn-run` command is also implemented. (This `run` is different than the original one, since GDB does not run the program actually, but applies a number of `step`-s automatically.)

4 Status and experiments

The current implementation consists of two patches: one for GCC and one for GDB. None of these cause any regression. In both cases our code is separated from the normal program logic by a new flag and some new internal commands.

We performed the validation of the implementation on the projects of the CSiBE benchmark [5]. Additionally, small examples were used to test special language elements. Currently we concentrated on purely C code, but in the near future we will be start experimenting with C++ as well. In general, currently we are in the middle of the stabilization process. The web page of the project [6] contains all relevant up-to-date information about this work.

4.1 GCC

The algorithm runs on CSiBE without any regression (using compiler version GCC 4.2.0 20070221). In Table 1 some compilation time and code size measurements are shown. The increase in compilation time was approximately 70% on most of the projects. Most of the time was spent on GIMPLE statement decomposition and temporary variable elimination. The code size increase was not so significant, generally at most 50%. This increase is due to the additional debugging information generated into the object code and the new statements and variables introduced by instrumentation.

Project	Time inc. (%)	Size inc. (%)
libmspack	70.6	52.3
zlib	79.2	58.7
mpgcut	60.0	47.6
flex	56.6	15.1
OpenTCP	38.2	24.7
teem	65.6	41.2
jpeg-6b	41.1	58.0
jikespg	92.0	35.0
linux-2.4	25.9	39.0
libpng	56.9	50.9
bzip2	96.5	49.6

Table 1: Compilation time and code size increase on CSiBE

4.2 GDB

As mentioned earlier, running the debugged program in a traditional way is not possible when dynamic slicing is enabled, since repetitive `step`-s are performed and not a `run`. Therefore we compared the normal `step` command with our modified `dyn-step`. For this we used some test programs and recorded the execution times for a fixed number of steps. Table 2 shows this comparison.

Project	100 steps	500 steps	1000 steps
flex	3.25	4.28	15.54
jpeg-6b	1.56	1.98	2.48
bzip2	3.13	6.496	36.56
libpng-1.2.5	2.54	4.28	15.54

Table 2: Runtime increase (percent)

The runtime increase is caused by processing the D/U information for the actual line stepped. The values show the increase in percent for the given amount of steps when running the global algorithm on the same program. It can be seen that in most cases it is very small, under 10%. However, in the case of complex programs this could reach more.

In our next experiment we measured the memory usage of the global algorithm. Since it maintains every variable's dynamic dependence set, the memory usage can be significantly increased compared to the original GDB. However, one must not forget that this algorithm

is significantly more efficient in terms of memory usage compared to other existing slicing methods. We compared memory usage by executing 1000 `step` and `dyn-step` commands. Table 3 shows these results, which are the increase in percentage.

Project	Memory usage inc. (%)
flex	293
jpeg-6b	207
bzip2	1543
libpng-1.2.5	196

Table 3: Memory usage increase

We have not done any optimization on our algorithms yet, so we hope that we can improve these numbers in the near future to make the implementation more practical.

5 Ongoing work

The basic functionality of the dynamic slicing algorithm is implemented for C, but there still are a lot of things to do. Here is a brief summary of the features that we are working on or plan to implement in the near future:

- Less conservative approach when generating the D/U representation for data structures. We should handle dependences between the fields of data structures instead of dependences between whole data structures.
- Prepare for type casts while generating the D/U representation, and the ability to handle different addressable parts of a variable as different entities in the D/U representation. At the moment, these are simply ignored.
- XML output generation from GDB to help verification.
- The `dyn-slice` command should accept also expressions or source lines instead of only variable names.
- Slicing support has recently been added for the KDbg GDB front end. Adding slicing support for other popular IDEs should be considered too.
- Other front ends should be checked and the necessary development should be made. Our next goal is to be able to slice C++ programs too, for which—theoretically—no significant development is required.
- The alternative algorithm, the *demand driven* [3] should be implemented too, to compare these two different concepts and find out which is more practical in realistic situations.
- An extension to the basic algorithm called *relevant slicing* should be considered as it improves the bug-finding possibilities [7].

References

[1] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software – Practice and Experience (SPE)*, 23(6):589–616, 1993.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN'90 Conference on Programming Language Design and Implementation*, number 6 in SIGPLAN Notices, pages 246–256, June 1990.

[3] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'06)*, pages 21–30, September 2006.

[4] Árpád Beszédes, Tamás Gergely, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Dynamic slicing method for maintenance of large C programs. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering (CSMR'01)*, pages 105–113, March 2001.

[5] Homepage of GCC Code-Size Benchmark Environment. `http://www.csibe.org`.

[6] Homepage of GNU GDB Slice project. `http://www.inf.u-szeged.hu/sed/gdbslice`.

[7] Tibor Gyimóthy, Árpád Beszédes, and István Forgács. An efficient relevant slicing method for debugging. In *Proceedings of ESEC/FSE'99*, number 1687 in Lecture Notes in Computer Science, pages 303–321. Springer-Verlag, September 1999.

[8] Mariam Kamkar, Nahid Shahmehri, and Peter Fritzson. Interprocedural dynamic slicing. In *Proceedings of PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 370–384. Springer-Verlag, 1992.

[9] Bogdan Korel and Janusz W. Laski. Dynamic slicing in computer programs. *The Journal of Systems and Software*, 13(3):187–195, 1990.

[10] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[11] Robert Morgan. *Building an Optimizing Compiler*. Digital Press, February 1998.

[12] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, number 19(5) in SIGPLAN Notices, pages 177–184, May 1984.

[13] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.