# Proceedings of the GCC Developers' Summit

June 2nd–4th, 2004
Ottawa, Ontario
Canada

## Contents

## Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

## Review Committee

Eric Christopher, *Red Hat, Inc.*
Janis Johnson, *IBM*
Toshi Morita, *Renesas Technologies*
Zack Weinberg, *CodeSourcery*
Al Stone, *Hewlett-Packard*
Richard Henderson, *Red Hat, Inc.*
Andrew Hutton, *Steamballoon, Inc.*
Gerald Pfeifer, *SuSE, GmbH*

## Proceedings Formatting Team

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

# CSiBE Benchmark: One Year Perspective and Plans

*Árpád Beszédes, Rudolf Ferenc, Tamás Gergely,*
*Tibor Gyimóthy, Gábor Lóki, and László Vidács*
Department of Software Engineering
University of Szeged, Hungary
{beszedes,ferenc,gertom,gyimi,loki,lac}@inf.u-szeged.hu

## Abstract

In this paper we summarize our experiences in designing and running CSiBE, the new code size benchmark for GCC. Since its introduction in 2003, it has been widely used by GCC developers in their daily work to help them keep the size of the generated code as small as possible. We have been making continuous observations on the latest results and informing GCC developers of any problem when necessary. We overview some concrete "success stories" of where GCC benefited from the benchmark. This paper overviews the measurement methodology, providing some information about the test bed, the measuring method, and the hardware/software infrastructure. The new version of CSiBE, launched in May 2004, has been extended with new features such as code performance measurements and a test bed—four times larger—with even more versatile programs.

## 1 Introduction

Maintaining a compact code size is important from several aspects, such as reducing the network traffic and the ability to produce software for embedded systems that require little memory space and are energy-efficient. The size of the program code in its executable binary format highly depends on the compiler's ability to produce compact code. Compilers are generally able to optimize for code speed or code size. However, performance has been more extensively investigated and little effort has been made on optimizing for code size. This is true for GCC as well; the majority of the compiler's developers are interested in the performance of the generated code, not its size. Therefore optimizations for space and the (side) effects of modifications regarding code size are often neglected.

At the first GCC summit in 2003, we presented our work related to the measurement of the code size generated by GCC [1]. We compared the size of the generated code to two non-free compilers for the ARM architecture and found that GCC was not too much behind a high-performance ARM compiler, which generated code about 16% smaller than GCC 3.3. However, at the same time we were able to document several problems related to code size as well, and more importantly we have demonstrated examples where incautious modifications to the code base produced code size penalties. At that time we had the idea of creating an automatic benchmark for code size.

To maintain a continuous quality of GCC generated code, several benchmarks have been used for a long time that measure the performance of the generated code on a daily basis [4]. However this new benchmark for

code size (called CSiBE for GCC **C**ode **Si**ze **BE**nchmark) was launched only in 2003 [2]. This benchmark has been developed by and is maintained at the Department of Software Engineering at the University of Szeged in Hungary [3]. Since its original introduction CSiBE has been used by GCC developers in their daily work to help keep the size of the generated code as small as possible. We have been making continuous observations on the latest results and informing GCC developers of any problems when necessary.

The new version of CSiBE, launched in May 2004, has been extended with new features such as code performance measurements and a test bed—four times larger—with even more versatile programs. The benchmark consists of a test bed of several typical C applications, a database which stores daily results and an easy-to-use web interface with sophisticated query mechanisms. GCC source code is automatically checked out daily from the central source code repository, the compiler is built and measurements are performed on the test bed. The results are stored in the database (the data goes back to May 2003), which is accessible via the CSiBE website using several kinds of queries. Code size, compilation time, and performance data are available via raw data tables or using appropriate diagrams generated on demand.

Thanks to the existence of this benchmark, the compiler has been improved a number of times to generate smaller code, either by reverting some fixes with side effects or by using it to fine tune some algorithms. In the period between May 2003 and 2004 an overall improvement of 3.3% in code size of actual GCC mainline snapshots was measured (ARM target with `-Os`) which, we believe, CSiBE also has contributed to.

In this paper we summarize our experiences in designing and running CSiBE. Section 2 overviews the system architecture while in Section 3 we give some examples of our observations and other people's benefits using CSiBE. Finally, we give some ideas for future development in Section 4.

## 2 The CSiBE system

In this section we overview the measurement methodology. We provide some details about the test bed, the measuring method, and the hardware/software infrastructure. Although the CSiBE benchmark is primarily for measuring code size, it provides two additional measurements: compilation speed, and code speed (for a limited part of the test bed). GCC source code is checked out daily from the CVS, the compilers are built for the supported targets (*arm/thumb*, *x86*, *m68k*, *mips*, and *ppc*) and measurements are performed on the CSiBE test bed. The results are stored in a database, which is accessible via the CSiBE website using several kinds of queries. The test bed and the basic measurement scripts are available for download as well.

### 2.1 System architecture

In Figure 1 the overall architecture of the CSiBE system is shown.

CSiBE is composed of two subsystems. The *Front end servers* are used to download daily GCC snapshots and use them for producing the raw measurement data. The *Back end server* acts as a data server by filling a relational database with the measurement data, and it is also responsible for presenting the data to the user through its web interface. The back end server together with the web client represents a typical three-tier client/server system. It serves as a data server (Postgres), implements various query logics and supplies the HTML presentation. All the servers run Linux.

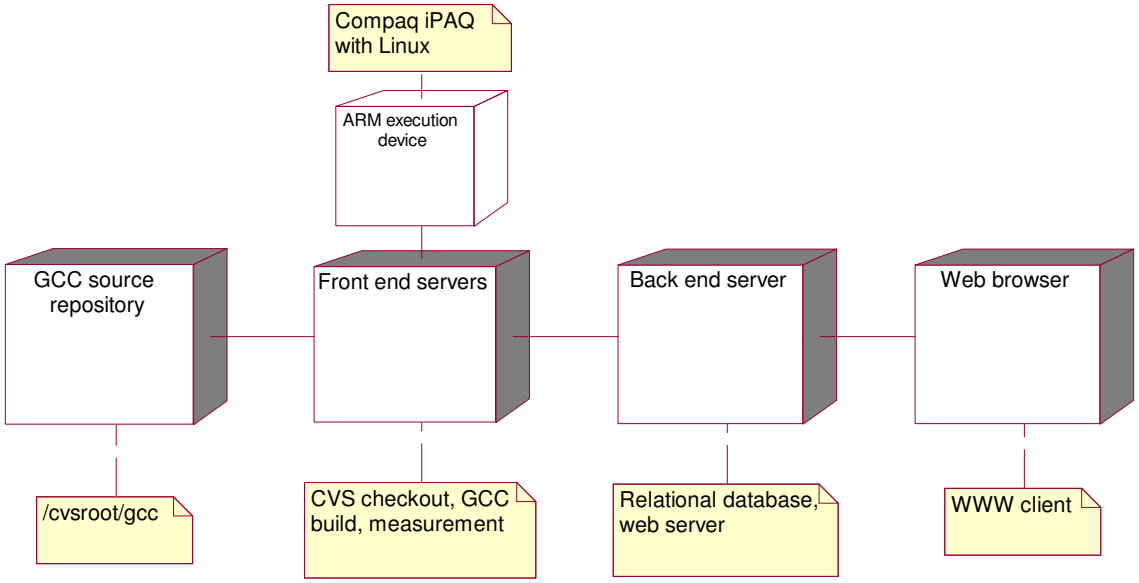



Figure 1: The CSiBE architecture

### 2.2 Front end servers

The core of CSiBE is the *offline CSiBE benchmark*, which consists of the test bed and required measurement scripts. This package is downloadable from the website, so it can also be used independently of the online system. The front end servers utilize this offline package as well.

The online system is controlled by a so-called *master phase* on the front end servers, which is responsible for the timely CVS checkout, compiler build, measurements using the offline CSiBE, and uploading the data to the relational database.

#### Hardware and software

The actual setup of the front end servers is flexible. At present, it is composed of three Linux machines, one used for CVS checkout that is shared with other university projects, and two dedicated PCs for the other front end phases. These two PCs are really siblings, having the same hardware and software parameters that are summarized below:

- Asus A7N8x Deluxe
- AMD AthlonXP 2500+ 333FSB @ 1.8GHz
- 2x 512MB DDR (200MHz)
- 2x Seagate 120GB 7200rpm HDD
- Linux kernel version 2.4.26, Debian Linux (woody) 3.0

These two servers are capable of sharing the measurement tasks (like separating them by branches) and, in this way, we also have a backup possibility in case of some unexpected server failure. These two servers are also used for measuring the performance of code generated for the x86 architecture. We are working on adding performance measurements for the ARM architecture as well, which will be made on a Compaq iPAQ device with the following main parameters:

- iPAQ H 3630 with StrongARM-1110 rev 8 (v4l) core
- 16M FLASH, 32M RAM
- Familiar Linux, kernel version 2.4.19-rmk6-pxa1-hh30

### Compilers and binaries measured

We measure daily snapshots of the GCC *mainline* development branch (previously the *tree-ssa* too) along with several release versions that serve as baselines for the diagrams. These are the following GCC versions: *2.95.2.1*, *3.2.3*, *3.3.1*, and *3.4*.

The compilers are configured as cross-compilers for the supported targets. We employ standalone targets for use with the *newlib* runtime library for code size and compilation time measurements, and Linux targets with *glibc* for execution time. At present, *binutils v2.14*, *newlib v1.12.0*, and *glibc v2.3.2* are used.

When we measure code size and compilation time, we do not include linking time and code size of the executable. Furthermore, only those programs that meet certain requirements are used for performance measurements. These are the following:

- The project produces at least one executable program
- The source files are not preprocessed
- The execution environment must not contain any special elements
- The execution time is measurable (i. e. it is not too short and not too long)

### CVS checkout

Snapshots of GCC source code are retrieved from the CVS daily at 12:00:00 (UTC). The complete code base is retrieved once a week on Mondays and on the other days only the differences are downloaded.

### Configuration

The *Binutils* package is configured with no extra flags, while *newlib* is configured with the only extra flag that enables the optimization for space: `-enable-target-optspace`. We do not build *glibc*, rather we use the stock binaries. Finally, GCC is configured with the following. The common flags are `-enable-languages=c -disable-nls -disable-libgcj -disable-multilib -disable-checking -with-gnu-as -with-gnu-ld`. Furthermore for compilers using the *newlib* library, the additional flags are `-with-newlib -disable-shared -disable-threads` and for *glibc* we also use `-enable-shared`.

### Compilation

A simple `make` was used to build *binutils* and the libraries once only, and the same is used for each GCC snapshot as well.

### Measurement

The code size is measured using the program `size`. The final result is the sum of the first two columns of the output of the command. This means that only program code and constant and initialized data sizes are incorporated into the final values.

Compilation time and code execution speed are measured three times per object and per test case, respectively. These times are measured with the program `/bin/time` in user mode. For both compilation and execution times all queries through the web will provide a time value that is the median of the three values. While compilation and execution times are being measured only vital processes are running on the machine.

The results of the measurements are stored in simple files in CSV format (comma separated values) for further processing. These files are also the final outputs of the offline CSiBE.

### 2.3 The test bed

The test bed consists of 18 projects and its source size is roughly 50 MB. When compiled, it is about 3.5 MB binary code in total. The test bed consists of programs of various types such as media (gsm, mpeg), compiler, compressor, editor programs, preprocessed units. Some of the projects are suitable for measuring performance and constitute about 40% of the test bed.

In the latest version of the test bed we added some Linux kernel sources as well. With this aim in mind, we started with the S390 platform and turned it into a so-called "testplatform." On this platform we replaced all assembly code with stubs and left only C code for the important Linux modules (kernel, devices, file systems, etc.)

The test bed is composed of two parts, one for the test programs and measurement scripts, and the other consisting of the test inputs for the executable projects. This separation was carried out so the user would be able to add many different test cases. The test cases were selected to represent one typical execution of the program as our goal was not to attain a good coverage of the program. In some cases the same input is given to a program several times, while in other cases the same program is executed with different inputs. The total size of the test inputs is currently about 60 MB.

In the table in Figure 2 some statistics about the test projects are given. We listed the number of source files, size of the source code in bytes, number of objects, total size of objects as measured using CSiBE for GCC 3.4, i686 and `-O2`, and the number of executable programs for each project.

### 2.4 Back end server

User queries through the CSiBE website are processed using PHP scripts, from which the necessary SQL queries are composed. The data retrieved from the database is then presented on the HTML output in data tables, bar charts, and timeline diagrams.

The central repository in which the measured data are stored is a relational database (implemented using Postgres). The database stores the measurement results along with the time stamp of the measurement and various entities such as the compiler and library version, compiler flags and measurement type. The version of the test bed is also associated with each result, which allows it to store the results of different test beds consistently. If a query is made that spans different test bed versions this can be easily displayed on the diagrams.

The last phase in the online CSiBE benchmark is the presentation on the website. The CSiBE pages provide quick and easy access to the most important measurements like the latest results in a timeline diagram or more elaborate query possibilities. Extensive help is provided for each function, making CSiBE simple to use. In Figure 3 the opening page can be seen.

There are several ways of retrieving the re-

| *Project* | *# Src.* | *Src. bytes* | *# Obj.* | *Bin. bytes* | *# Exec.* |
|---|---|---|---|---|---|
| `bzip2-1.0.2` | 11 | 242,034 | 9 | 80,112 | 2 |
| `cg_compiler_opensrc` | 42 | 813,343 | 22 | 148,838 | — |
| `compiler` | 9 | 202,938 | 6 | 27,928 | 1 |
| `flex-2.5.31` | 33 | 658,799 | 22 | 240,206 | 1 |
| `jikespg-1.3` | 29 | 978,833 | 17 | 267,712 | 1 |
| `jpeg-6b` | 81 | 1,119,991 | 66 | 156,078 | 3 |
| `libmspack` | 40 | 319,611 | 25 | 76,506 | — |
| `libpng-1.2.5` | 21 | 859,762 | 18 | 128,941 | 2 |
| `linux-2.4.23-pre3-testpl...` | 2,430 | 34,238,976 | 271 | 993,815 | — |
| `lwip-0.5.3.preproc` | 30 | 928,538 | 30 | 86,486 | — |
| `mpeg2dec-0.3.1` | 43 | 461,047 | 29 | 62,873 | 1 |
| `mpgcut-1.1` | 1 | 28,889 | 1 | 29,845 | — |
| `OpenTCP-1.0.4` | 40 | 545,358 | 22 | 38,221 | — |
| `replaypc-0.4.0.preproc` | 39 | 1,692,413 | 39 | 64,221 | — |
| `teem-1.6.0-src` | 370 | 2,786,644 | 293 | 1,210,365 | 2 |
| `ttt-0.10.1.preproc` | 6 | 311,311 | 6 | 19,049 | — |
| `unrarlib-0.4.0` | 4 | 93,894 | 3 | 16,339 | — |
| `zlib-1.1.4` | 27 | 305,136 | 14 | 42,422 | 1 |
| **Total** | **3,256** | **46,587,517** | **893** | **3,689,957** | **14** |

Figure 2: CSiBE test bed statistics

sults. One is *Summarized queries*, which provides instant access with a click of a button to all kinds of results (code size, compilation time, and code performance) for a selected target architecture. On the *Latest results* pages the last few days or weeks can be observed in several ways: timeline, normalized timeline (the various kinds of data are shown as normalized to the last value), a comparison of different targets, and raw number data. The *Advanced queries* pages provide the possibility of retrieving the data in any desired combination; one can compare any branch and target with any other combination and timeline diagrams for arbitrary intervals. Baseline values of major GCC releases are also available for most queries, which can be optionally selected for the diagrams.

All queries can be performed by a series of selections from drop-down lists like the selection of targets, branches, and optimization switches. The results can be displayed in a diagram (Figure 4a), in a bar chart (Figure 4b), or as raw data tables. The resulting latest timeline diagrams are supplied with two automatically generated links that can be copied for further reference. The *Static URL* link will always give the same diagram since all query parameters are converted to absolute time stamp values, while the *Reference URL* link supplies the actual query parameters at the time of usage, which gives values relative to the actual time.

## 3 Experiences

CSiBE has been quickly accepted by the community. Patches with references to its usage started to appear only after 2 months. At present we have 47 hits per day on average and a total of 193 downloads of the offline benchmark. A good thing about its introduction is that more and more GCC developers

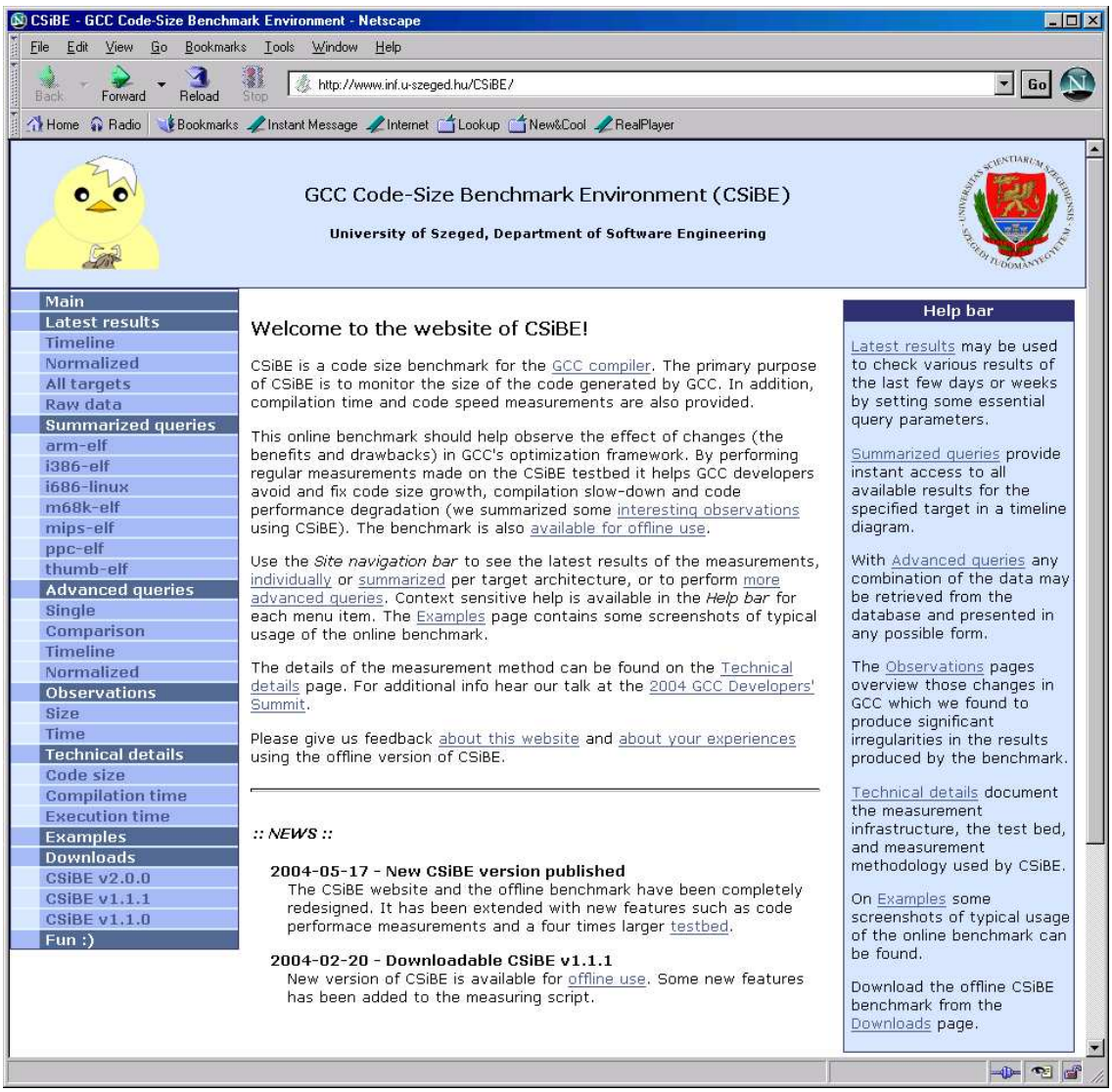
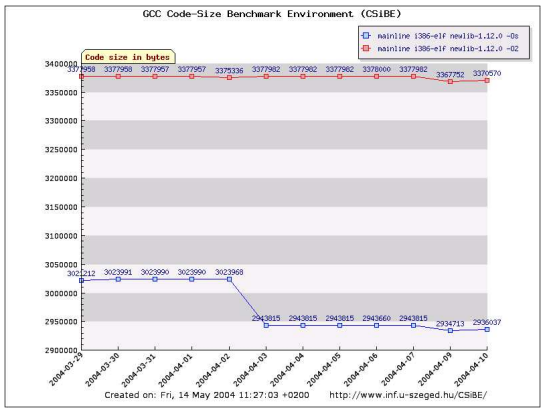


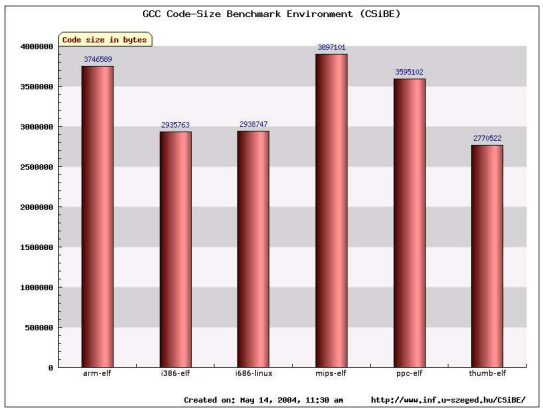

Figure 3: CSiBE website

seem to be using CSiBE in their daily work to check how their modifications affect the code size. Some people are developing patches to decrease code size, and the effect is measured with CSiBE, while others verify whether other modifications affect code size or not. Thanks to CSiBE, in 4 cases a patch was reverted or improved because of its negative effect on code size. These statistics suggest that the developers are starting to focus not only on code efficiency, but its size as well. We have been following the activity on the `gcc-patches` mailing list and found that more and more people are referring to CSiBE as a reference benchmark for code size (54 e-mails).

Our group has also contributed to the overall improvement of code optimization for size, because we are carrying out continuous observations of the results produced by CSiBE, of which the important ones are documented on the website. Where possible we also suggest a possible cause of any anomalies seen in the latest diagrams, and take steps to draw the attention of the community to the problem. In




(a) Timeline




(b) All targets

Figure 4: Diagram examples

the following we offer some examples of our observations and successful participations:

- On August 31 in 2003 a patch was applied to improve the condition for generating jump tables from switch statements by including the case when optimizing for size. This caused a code size reduction on all targets. The threshold value was determined based on the CSiBE statistics.
- In September 2003 unit-at-a-time compilation was enabled in mainline, which resulted in major code size improvement for most targets.
- A patch related to constant folding done in October 2003 increased the code size for all targets. Several days later another patch was used to disable some features when optimizing for size.
- A significant code size increase was measured on October 21, 2003 on ARM architecture when optimizing for size due to a patch that allows factorization of constants into addressing instructions when optimizing for space. One week later the patch was reverted.
- In January 2004 a patch saved code size on ARM with `-Os` but introduced a new bootstrap failure.
- A patch on April 3, 2004 saved about 1% of code size for most targets. The patch inlines very small functions that usually decrease the code size when optimizing for size.

## 4 Conclusion and future plans

In this paper we overviewed GCC's code size benchmark, CSiBE. We presented the overall architecture, the test bed and the measuring method. Although it primarily serves as a benchmark for measuring code size, other parameters such as compilation time and code execution performance are also part of the regular measurements. We offered some examples of where GCC benefited from using the benchmark, and pointed out that, in recent years, a general interest towards code size has increased among GCC developers. As a result of this, GCC mainline improved about 3.3% in terms of generated code size between May 2003 and May 2004 (measured with CSiBE test bed version 1.1.1 for the ARM target and `-Os`).

We plan to continue our work with CSiBE and

hence we welcome users' comments and suggestions. Some of the targets were added after user requests, and the bigger test bed in the latest CSiBE version is also composed of programs based on the demands of those who contacted our team. In the future we will try to follow the real needs of the GCC community, those of the developers and users.

One of the straightforward enhancements of CSiBE might be to introduce new targets and development branches, should there be an interest in it by the community. As long as the available hardware capacity permits (the measurement of one day's data currently takes about 5 hours), we may extend the test bed with new programs, should it prove necessary.

Another idea of ours for enhancing the online benchmark is to allow users to upload, via the web interface, measurement data they produced offline into the central database. This would be interesting in cases where a developer makes use of the offline benchmark to measure a custom target or examine code performance with different inputs.

## 5 Availability

The online CSiBE benchmark can be accessed at

`http://www.inf.u-szeged.hu/CSiBE/`

From here the offline version can also be downloaded.

## Acknowledgements

The CSiBE team would like to thank all those GCC developers who helped us develop the benchmark with their useful comments and constructive criticisms.

## References

[1] Árpád Beszédes, Tamás Gergely, Tibor Gyimóthy, Gábor Lóki, and László Vidács. Optimizing for space: Measurements and possibilities for improvement. In *Proceedings of the 2003 GCC Developers' Summit*, pages 7–20, May 2003.

[2] Department of Software Engineering, University of Szeged. GCC Code-Size Benchmark Environment (CSiBE). `http://www.inf.u-szeged.hu/CSiBE`.

[3] Department of Software Engineering, University of Szeged. Homepage. `http://www.inf.u-szeged.hu/tanszekek/szoftverfejlesztes/starten.xml`.

[4] The GNU Compiler Collection. GCC benchmarks homepage. `http://gcc.gnu.org/benchmarks`.

# Performance work in the libstdc++-v3 project

*Paolo Carlini*
SUSE
pcarlini@suse.de

## Abstract

The GNU Standard C++ Library v3 is a long term project aimed at implementing a fully conforming C++ runtime library, as mandated by the ISO 14882 Standard. Whereas during the first years the focus was mostly on features, recently, after my appointment as one of its official maintainers, much more attention is devoted to performance issues and contributions in the area are particularly encouraged and appreciated. In this paper the main approaches being followed are reviewed (e.g., hand-coding, exploitation of glibc extensions, caching), together with the tools used, and a number of satisfying results obtained so far, particularly, in the iostreams and locales chapters. Quantitative comparisons on x86-linux with the Icc/Dinkumware offer will be also presented, based on code snippets provided by the new performance testsuite and distilled from actual performance PRs. In the final section, a better integration with the compiler team is argued for and emphasized.

## 1 Introduction

Today, *circa* 2004, the libstdc++-v3 project delivers in a typical GCC distribution more than 420000 lines of code, including 1350 regression testcases and a growing performance testsuite. Many different architectures, both 32-bit and 64-bit, are fully supported, on many different OSes, from x86 to s390x and from Linux to Darwin.

The project was started in 1998 and release after release the "degree of conformance" to the ISO C++ Standard is becoming very high, with many features implemented satisfactorily and quickly stabilizing.

Indeed, an analysis of the 3.4.0 Release Notes reveals that major changes, that first blush may seem conformance related (e.g., UTF-8 support, generic character traits) in fact should be strictly speaking categorized as QoI improvements.

On the other hand, the users are becoming rather demanding as far as performance is concerned. Among the possible causes: the good speed in some areas (e.g., I/O) of the old, pre-standard, C++ runtime library; new offers, like Icc/Dinkumware, on the market and easily available on the widespread x86-linux platform. More generally, does not seem obvious anymore that library functions that have a C library counterpart must be necessarily slower: people want a complete "object oriented" application not renouncing to performance.

Also, some new facilities offered by the ISO Standard are recently gaining larger popularity (e.g., locale) and real world applications are able to emphasize weaknesses that went unnoticed to the implementors, naturally caring more about conformance, in the first place.

The main focus of the work is therefore slowly changing and the purpose of this paper is dis-

cussing how, using which methods, and exploiting which instruments. Of course, one of its main objectives is soliciting feedback and opening a discussion on such topics. The first part presents sort-of a chronology of the most relevant recent achievements[1], spanning the last year or so: it represents also a nice occasion to thank some of the most generous contributors. Then, three items will be discussed more throughly to convey a few specific, general points. In the last part, moving from a recent episode, a better integration with the compiler team will be wished.

## 2 A Chronology

Necessarily, there is a good amount of "fuzziness" in this type of historical reconstruction: many important contributions went in only after a long discussion, or piecewise, during a few months. Most of the changes presented below are only in 3.4 (and mainline, of course), but, also due to the above mentioned reasons, not *all* the performance related improvements in the current release branch will be exhaustively listed.

**Output of integers** For GCC 3.3 Jerry Quinn rewrote from scratch the code formatting integer types for output, avoiding going through `sprintf` for performance sake: probably for the first time, the implementation interpreted non-trivially one of those typical "as if" specifications present in the Standard.

**Separate synched `filebuf`** In this case, it could be said that a speed gain has been obtained as a (very welcome) by product: Pétur Runólfsson separate synched filebuf improved remarkably the conformance of the library in the interactions with C stdio (e.g., cin/stdin). Anyway, as a matter of fact, `cout.rdbuf()->sputc('a')` became for instance about three times faster.

**`Numpunct` cache** After some initial attempts during GCC 3.3 lifetime, finally GCC 3.4 exploits caching for formatted I/O: this important issue will be discussed in detail below. In any case, formatted output of integer types is now three time faster than in GCC 3.2.3 (Table 1).

| | |
|---|---|
| GCC 3.2.3 | 14.590u 0.010s 0:14.67 99.5% |
| GCC 3.3.3 | 4.780u 0.010s 0:04.80 99.7% |
| GCC 3.4.0 | 4.160u 0.010s 0:04.19 99.5% |
| Icc8.0 | 10.430u 0.020s 0:10.48 99.7% |

Table 1: Output of int`s` from 0 to 9999999 to `/dev/null`

**Empty `string` speedup** At the beginning of 2003, Nathan Myers, the original author of v3 `basic_string` class, noticed that the multi-processor bus contention can be reduced by comparing addresses first, and never touching the reference count of the empty object. The final patch has been committed in time for 3.4 and improves remarkably the performance on single-processor systems too: Table 2 presents satisfying timings for a simple snippet shown in Figure 1.

```
for (int i = 0; i < 2000; ++i)
  std::string a[100000];
```

Figure 1: Creating and immediately destroying lots of `string` objects

**Non-unified `filebuf`** According to the C++ Standard, a `seek` is needed in order to switch from read mode to write mode (and

[1] If not otherwise indicated, all the timings are relative to a P4-2400 machine, linux2.4, glibc-cvs, -O2, Icc8.0 Build 20040412Z.

| | |
|---|---|
| GCC 3.3.3 | 20.890u 0.020s 0:21.01 99.5% |
| GCC 3.4.0 | 0.790u 0.000s 0:00.79 100.0% |
| Icc8.0 | 17.200u 0.030s 0:17.33 99.4% |

Table 2: Execution times for the code displayed in Figure 1

vice versa) during I/O. This is a very reasonable requirement, by the way inherited from the C Standard. However, the old implementation, as a (very puzzling) QoI feature, had relaxed it: unfortunately, the upshot was that the get area and put area pointers had always to be updated in a lockstep way. Figure 2 compares GCC 3.3 and GCC 3.4 code for `sputc`: the former called `_M_out_cur_move` instead of simply bumping the put area pointer by way of `pbump`. Additionally, the `_M_out_buf_size` helper was also needed: as a result the function was not amenable to inlining anymore. The same happened of course for `sbumpc` and elsewhere. The performance suffered consequently as Table 3 demonstrates.

| | |
|---|---|
| GCC 3.3.3 | 42.440u 0.290s 0:42.91 99.5% |
| GCC 3.4.0 | 4.080u 0.300s 0:04.39 99.7% |
| Icc8.0 | 11.080u 0.360s 0:11.45 99.9% |
| 'C' (unlocked) | 6.590u 0.280s 0:06.90 99.5% |

Table 3: Char-by-char copy of 1 GB from `/dev/zero` to `/dev/null`

Fixing this required consistent, invasive changes to `streambuf`, and `stringbuf` but eventually enabled a much simpler maintenance and paved the way to the UTF-8 support.

**Input of integers** The code parsing integers could be improved rather easily, thanks to the `numpunct` caching mechanism already in place and functioning well. Interestingly, though, in this area the library sports some design choices not shared by other implementations (whereas consistent with the letter of the standard!), to be discussed below.

**Table-based `ctype`** In order to obtain fast `time_get` and `time_put` facets (not suited for caching, due to their special requirements), and also for free standing use, `ctype` functions, such as `narrow`, `widen`, and `is`, are now table-based. Thanks to a sophisticated solution devised by Jerry and refined on the discussion list, for `char` type it is even avoided the virtual function call cost. The improvement is more visible for `wchar_t`, however: once more, close to an order of magnitude with respect to the previous generation.

**`Codecvt` rewrite** During GCC 3.4 Stage 1 Pétur rewrote the `codecvt` facet, obtaining a very good support of encoding-zero (e.g., UTF-8) locales too. In the process, he provided a rather complete set of testcases. Finally, as will be discussed in the second part, performance has been also improved, thus delivering for the first time both correct and efficient support for a wide set of locales.

**Other `string` improvements** In Item 29 of his latest book, *Effective STL*, Scott Meyers proposes an elegant idiom for copying a text file into a `string` object (Figure 3).

```
string Data(istreambuf_iterator<char>(File),
            istreambuf_iterator<char>());
```

Figure 3: Istreambuf_iterators usage

In order for this proposal to be effective, the constructor from a pair of `input_iterator`*s* must be efficient: a satisfactorily fix involved redesigning the latter to exploit a centralized growth fa-

cility, previously not available at construction time. Only in 3.4.1-pre is present another unrelated improvement, very simple but appreciable in almost every use of the `string` class[2]. It consists in special-casing single char changes to avoid the general `traits::copy` and `traits::assign`, which end up calling C library functions (Table 4).

| | |
|---|---|
| GCC 3.3.3 | 1.150u 0.040s 0:01.19 100.0% |
| GCC 3.4.0 | 0.670u 0.070s 0:00.74 100.0% |
| GCC 3.4.1 pre | 0.220u 0.060s 0:00.28 100.0% |
| V2 | 0.710u 0.030s 0:00.74 100.0% |

Table 4: Ten millions of `str.append(1, 'x')`

**Monetary facets** Extending the `numpunct` caching work to `moneypunct` turned out to be easy. However, in the process, a few bugs and other opportunities for performance surfaced. Some are certainly straightforward (e.g., reordering operations on `string` objects to avoid reallocations), but, nevertheless, the overall effect is quite noticeable. For instance, Table 5 shows the time it takes to read one million of times a big monetary amount, i.e., 100,000,000,000.00, from an `istringstream` into a long double.

| | |
|---|---|
| GCC 3.3.3 | 10.610u 0.020s 0:10.69 99.4% |
| GCC 3.4.0 | 4.110u 0.000s 0:04.12 99.7% |
| GCC 3.4.1 pre | 2.910u 0.010s 0:02.93 99.6% |
| Icc8.0 | 3.280u 0.000s 0:03.29 99.6% |

Table 5: A simple `money_get` benchmark

The difference between GCC 3.4.0 and 3.4.1-pre is entirely due to the just-mentioned simple tweak to the `string` class: a similar effect can be measured in the formatted input of floating point types, much more used today.

**`Locale` functions** Probably, a large number of applications doesn't have these functions as a performance bottleneck. On the other hand, the way names were processed used to be rather dumb, due to the encoding adopted for "simple" named locales—that is, roughly, having all the categories named the same, say `de_DE`. As pointed out by library-friend Martin Sebor, most probably the sections of the standard having to do with combining named locales (22.1.1.2, 22.1.1.3) will be amended: therefore the real challenge was designing a new encoding ready for the most likely future changes. Table 6 shows the time needed to compare ten millions of times via `operator==` two "simple" locales.

| | |
|---|---|
| GCC 3.3.3 | 13.410u 0.000s 0:13.45 99.7% |
| GCC 3.4.0 | 11.640u 0.000s 0:11.67 99.7% |
| GCC 3.5.0 exp | 0.220u 0.000s 0:00.22 99.9% |
| Icc8.0 | 0.850u 0.000s 0:00.85 99.9% |

Table 6: A simple `locale::operator==` benchmark

**`Getline` speedups** A wide ranging debate ensued to the submission of PR 15002, with the participation of Matt Austern, among others. Both the `getline`s had to be improved: the member taking a `char_type*` and a `streamsize` and the function taking an `istream` and a `string`. An elegant solution, devised by Pétur, could be adopted only for the former, since it exploits *protected* `streambuf` members. It became clear that, ideally, we should have two different versions of those functions: the fast version, which takes advantage of friendship

[2] Internally to the library too, as will be quantified in the next item.

and only works for `char` and `wchar_t`, and a slow version that goes through the public interface. For the moment, profiling revealed that a large speedup could be achieved by appending to the `string` object a chunk of each line at a time (say, 128 chars), instead of one char at a time. Table 7 shows timings for reading 600000 lines, each 200 characters, from file, via `getline`.

| char_type* | |
|---|---|
| GCC 3.3.3 | 1.700u 0.090s 0:01.80 99.6% |
| GCC 3.4.0 | 1.230u 0.070s 0:01.30 100.0% |
| GCC 3.4.1 pre | 0.180u 0.130s 0:00.30 103.3% |
| Icc8.0 | 1.410u 0.090s 0:01.50 100.0% |
| `string` | |
| GCC 3.3.3 | 15.560u 0.070s 0:15.69 99.6% |
| GCC 3.4.0 | 9.030u 0.160s 0:09.22 99.6% |
| GCC 3.4.1 pre | 1.090u 0.110s 0:01.21 99.1% |
| Icc8.0 | 1.910u 0.120s 0:02.04 99.5% |

Table 7: `Getline` benchmarks

## 3 Telling Stories

### 3.1 Parsing of integer types and caching

Back in February, in the occasion of some changes to the monetary facets that were supposed to be completely uncontroversial, a long exchange started on the discussion list about the correct way to parse monetary (and numeric) quantities.

In particular, it became evident to everyone that libstdc++-v3 is probably *alone* in closely following the letter of 22.2.2.1.2. Most, if not all, the other implementations are not using `widen` and are not matching characters as prescribed in p8: instead, in order to compute the value of each specific digit `d` something equivalent to `c = narrow(*beg, '*')` is first computed, then `d` is given by `c - '0'`.

Indeed, this approach has its own virtues: there is no need for caching (and the related complexities[3]) and an efficient table-based `narrow` is sufficient alone to obtain good performance; moreover, this approach solves elegantly an issue in the Standard with the "mysterious" `find` function mentioned in p8.

In fact, if the function is interpreted (rather naturally) as `traits::find`, the serious problem ensues that any `charT`, other than plain `char` and `wchar_t`, needs an appropriate `traits<charT>::find` to be available: the Standard nowhere requires this, still clearly mandates in Table 52 to make it possible to instantiate `num_get` on *any* `charT` type[4].

Interestingly, those issues are of course well known to the LWG members, but often do *not* correspond to detailed and well debated DRs.[5]

Anyway, GCC 3.4 provides for the first time a generic `traits` class, which includes indeed a generic `traits<charT>::find`: this leads to a complete solution characterized by an excellent performance/conformance balance. Table 8 below compares the timings for reading from file ten millions of integers, from 0 to 9999999.

| | |
|---|---|
| GCC 3.3.3 | 41.180u 0.020s 0:41.37 99.5% |
| GCC 3.4.0 | 5.740u 0.030s 0:05.79 99.6% |
| Icc8.0 | 14.220u 0.120s 0:14.41 99.5% |
| V2 | 5.930u 0.060s 0:06.00 99.8% |
| Hammer | 18.660u 0.040s 0:18.78 99.5% |

Table 8: A simple `num_get` benchmark

For 3.4, integer types parsing has been rewrit-

[3] Only 3.4 finally managed to have it reliably working, fast, and... not leaking memory!

[4] A POD type.

[5] Only DR 303 [WP] and DR 427 [Open] are relevant and both the resolution of the former and the comment added in Kona to the latter are clearly *against* preferring `narrow` to `widen`.

ten, avoiding `strtol`, `strtoll`, and the other C library functions previously used, instead directly accumulating the result during the parsing. Therefore, no `string` objects are involved. The hammer-branch entry is also present in the Table in order to quantify what could be otherwise achieved within the constraints of the 3.3 ABI, basically, by improving the use of the `string`s.

On the current code base, `gprof` reports that about 58% of the total time is spent in the parsing loop itself: not much can be done about this, except, perhaps, avoiding an integer division, in principle not necessary. `Memchr`, called by `traits<char>::find`, is the second topmost entry, with about 26%: in the future, a small ABI change could make possible detecting in advance the occurrence of trivial `widen`s, very common indeed, then simply using `d = *beg - widen('0')` in such cases: `traits::find` would not be necessary at all and the QoI would be further improved. All the other entries are below 5% and `__use_cache::operator()` is below 1%, a reassuring check.

In any case, barring unexpected strong requests from the users, much more effort is planned in the area of parsing and formatting of *floating point* types, which probably could be made about two times faster, but this is another story...

### 3.2 `Codecvt` rewrite

As already mentioned, a few months ago became evident that the performance of the most important `codecvt` functions, such as `in`, `out`, and `length`, was not satisfying: that represented a major roadblock in the way of efficient encoded I/O, otherwise made finally possible by the redesigned `filebuf` virtuals.

By the way, this problem was unfortunately noticed only *months* after the `codecvt` rewrite, a sad episode less likely to happen nowadays thanks to the new performance testsuite[6], which currently includes 30 testcases and is quickly growing: most of the tests are distilled from performance PRs.

Luckily, after some preliminary attempts, only partially successful, the real fix became obvious: it involved exploiting `mbsnrtowcs` and `wcsnrtombs`, two glibc extensions that take an extra parameter with respect to the standard `mbsrtowcs` and `wcsrtombs`. Indeed, admittedly, in GCC 3.3 `codecvt` was almost broken but already fast, thanks to the use of the latter functions. Table 9 is relative to the conversion of 400000 buffers, 1024 characters each, in the C locale.

| | |
|---|---|
| GCC 3.3.3 | 1.520u 0.000s 0:01.52 100.0% |
| GCC 3.4.0 | 1.650u 0.000s 0:01.65 100.0% |
| Icc8.0 | 41.670u 0.010s 0:41.85 99.5% |

Table 9: `Codecvt::in` benchmark

The small difference between GCC 3.3.3 and GCC 3.4.0 is entirely due to the additional call of `memchr` (or `wmemchr`), which is used for splitting the input (the output, respectively) in chunks, ending in '\0' (or L'\0', respectively): each one is then processed by `mbsnrtowcs` (`wcsnrtombs`, respectively).

The numbers obtained with Icc8.0 are typical of an implementation using for correctness the *single* char C library functions `wcrtomb` and `mbrtowc`: this is still happening for libstdc++-v3 too in the so-called "generic" locale model, which doesn't have the GNU extensions available. Discussing `codecvt` is therefore also an opportunity to clarify that the QoI provided by the library in that model is sometimes lower than in the GNU model. Improving this situation is feasible but requires

[6] Established June, 2003.

more help from people on platforms not based on glibc, hereby strongly solicited!

## 4 The Weird Loop, Outlook

An interesting feature of the C++ `cmath` and `complex` facilities is the presence of additional `pow` overloads for *integer* exponent, not present in the C Standard, that, in principle at least, enable a wide range of additional opportunities for optimization. The library implements those overloads using a function that computes the power via the well known "Russian peasant algorithm" (Figure 4) which requires only `O(log n)` multiplications.

```
template<typename _Tp>
  inline _Tp
  __cmath_power(_Tp __x, unsigned int __n)
  {
    _Tp __y = __n % 2 ? __x : 1;
    while (__n >>= 1)
      {
        __x = __x * __x;
        if (__n % 2)
          __y = __y * __x;
      }
    return __y;
  }
```

Figure 4: Helper function used by `pow`

As evident from the actual code, the loop is very simple but nonetheless characterized by a *non-linear* induction variable, not handled until a few months ago neither by the old unroller nor by the new one, present in the lno-branch and actively developed by Zdenek Dvorak and others.

"Officially" Zdenek considered non-linear IVs rare and low priority[7], but actually he was just looking for an interesting example of application, nothing more! In a matter of *few* weeks a complete framework for canonical IVs creation was ready and beautifully effective: in it, loops such as the above can be fully unrolled in case of a constant `__n` thus leading to just *perfect* assembly.

Besides the technical details of the episode—who knows, perhaps by the time the lno-branch is merged the library will not use the very same algorithm—its lesson seems definitely an invitation to more frequent and strict exchanges between the library and compiler people.

## Acknowledgments

Many thanks go to SUSE for the enthusiastic support of my work; to Benjamin Kosnik, who trusted and encouraged me back in 2001 (and still does!); to Nathan "Less is More" Myers, a constant source of inspiration; to all my Italian friends, especially a little smiling hamster (")

[7] See the audit trail of PR 11710.

```
_M_out_buf_size()
{
  off_type __ret = 0;
  if (_M_out_cur)
    if (_M_out_beg == _M_buf)
      __ret = (_M_out_beg + _M_buf_size
               - _M_out_cur)
    else
      __ret = _M_out_end - _M_out_cur;
  return __ret;
}

_M_out_cur_move(off_type __n)
{
  bool __testin = _M_in_cur;
  _M_out_cur += __n;
  if (__testin && _M_buf_unified)
    _M_in_cur += __n;
  if (_M_out_cur > _M_out_end)
    {
      _M_out_end = _M_out_cur;
      if (__testin)
        _M_in_end += __n;
    }
}

sputc(char_type __c)
{
  int_type __ret;
  if (_M_out_buf_size())
    {
      *_M_out_cur = __c;
      _M_out_cur_move(1);
      __ret = traits_type::to_int_type(__c);
    }
  else
    __ret = this->overflow(
            traits_type::to_int_type(__c));
  return __ret;
}
```

(a) GCC 3.3

```
sputc(char_type __c)
{
  int_type __ret;
  if (this->pptr() < this->epptr())
    {
      *this->pptr() = __c;
      this->pbump(1);
      __ret = traits_type::to_int_type(__c);
    }
  else
    __ret = this->overflow(
            traits_type::to_int_type(__c));
  return __ret;
}
```

(b) GCC 3.4

Figure 2: GCC 3.3 (a) vs GCC 3.4 (b) code for `sputc` (slightly simplified)

# Declarative world inspiration

*Zdeněk Dvořák*
SuSE Labs
dvorakz@suse.cz, http://atrey.karlin.mff.cuni.cz/~rakdver/

## Abstract

The techniques for compilation and optimization of the declarative (logic and functional) programming languages are quite different from those used for procedural (imperative) languages, especially on the low level. There are however several reasons why they are still relevant even for the typically procedural language compilers like GCC: On higher level we can observe similarities, and due to more systematic design of the declarative languages the development in these areas is usually more advanced. In some contexts it is also considered a good style to use declarative programming techniques (recursion, generic programming, callbacks) even in imperative languages; currently the performance penalties for these constructs are usually quite large.

The paper quickly summarizes the similarities and differences between compilation of declarative and imperative languages. We then investigate the techniques used for declarative languages—tail recursion and general recursion optimizations, advanced inlining techniques (partial inlining, function specialisation, partial evaluation), program analysis, intermodular optimizations, etc., their usability and implementability in GCC.

## Introduction

The contemporary programming languages can be divided into procedural and declarative. Programs in procedural languages describe precisely the control flow and they map more or less directly to the machine code of the target platform. On the other hand declarative languages focus on describing the semantics of the program. They do not describe that much how things should be done, but rather specify what result we would like to obtain, leaving the exact way how to do it up to the compiler. Of course this division is not all that clear—many procedural languages include some constructions derived from especially functional languages, and declarative languages usually contain procedural bits in order to handle things like input and output.

It is well-known fact that the compilation of declarative languages is in some sense both easier and harder than the compilation of procedural languages. Easier since the semantic description gives more freedom to the compiler and makes the analyzes simpler. Harder since the lack of explicit control flow makes it necessary to for a compiler to select a good order of execution by itself. This in general cannot be done in compile-time, so this makes it necessary to handle partially evaluated data in runtime. Also the more high-level nature of the declarative languages invites the programmers to use the constructions whose straightforward translation would be quite ineffective.

Of course on the low-level the techniques for compilation of procedural and declarative languages are quite different (it is also true that they also differ significantly between the vari-

ous types of declarative languages). On higher-levels however the goals of the optimizations are more similar, and it is just here where the declarative languages may use the benefits of their cleaner semantics. Often it happens that even those high-level optimizations that could theoretically be used for procedural languages as well are only developed for the declarative ones, due to the problems with the level of analysis necessary to enable the alterations of the control flow prescribed by the procedural program. Also due to this diversity the research groups for declarative and procedural language compilation techniques do not communicate with each other frequently, so it may happen that the optimizations developed by one of them are either unknown or developed independently by the other one.

This paper tries to give an overview of (mostly high-level) optimizations used in declarative language compilers and to put them in context of the procedural language compiler GCC. We try to investigate their implementability and usefulness and also to derive some optimizations based on them that might be more useful for optimization of procedural languages.

First we provide a short introduction to the declarative language compiler construction and define the terms used in the area. Then we continue with the short descriptions of available optimizations, with more detailed descriptions for those that we consider relevant in the context of procedural languages. We also provide some thoughts and pointers on the eventual implementation in the current infrastructure of the GCC compiler (tree-ssa branch, since all the optimizations are only suitable for implementation on the tree level).

# 1 Compilation of Declarative Languages

In this section we provide a short introduction to the techniques used for compilation of the declarative languages. References to papers containing more detailed descriptions are provided. Of course the approaches different from the ones described below used as well, and the basic schemes can be altered to obtain variations useful for specific purposes.

We must distinguish between the different kinds of declarative languages, especially between logic (based on the predicate logic) and functional (based on the lambda calculus) ones. There are also other special purpose declarative languages (for constraint programming, database querying, scene description, etc.), but these are out of the scope of this paper.

Initial stages of compilation of all the languages, like lexical and syntactic analysis, are of course very similar and not interesting from our point of view. The optimizations (both generic and specific for the given style of the language) are then performed (some of them will be mentioned in the following sections). Usually the level of the representation is lowered during the process, finally leaving us with just the basic elements of the language. Type (and for logic languages mode—determining which arguments of predicates are input/output) checking and eventually specialization of operations happens during this process

For logic languages the basic elements are unification (that includes both construction and decomposition of data structures) and definition of predicates (that usually are recursive and use some built-in predicates for performing things like arithmetics). The language specification also defines the rule for order of evaluation of the predicates, which may be

fixed (Prolog), flexible subject to some minimal constraints (Mercury) or even alterable by the program (Goedel). The possible substitutions to the variables are processed according to this rule, backtracking whenever such a substitution fails.

This rule (or its particular variant chosen by the compiler) is then translated into a code of a low-level abstract machine (which is later mapped to the target language). One of those used is the Warren Abstract Machine ([W83]). It consists of the low-level instructions to control unification, predicate lookup and backtracking. Indeed the main challenge at the low level is to make these operations efficient.

For unification it is necessary to handle the special cases of unification with terms with known structure, and to employ efficient algorithm for matching the terms with unknown structure when this fails. This is complicated by the fact that unification is two-way process (i.e. both unified sides may get modified). Also we need to be careful about the possibility to create the cyclic structures when compiling unifications like $X = f(X)$.

Predicate lookup is usually made faster by filtering out predicates that cannot match due to the known structures of parameters; this indexing may be either shallow (only looking at the topmost level) or deep. The things get more complicated in languages like Prolog where the program may be changed dynamically.

For backtracking we need to implement the rollback mechanism, either using timestamps or a clever layout of allocated data structures (or both).

For more details on construction of logic language compilers see for example [R94], [DC01] or [HS02].

For functional languages the basic elements are pattern matching (data structure decomposition), data structure construction and function application. Local function definitions are usually replaced by the global ones, in process called lambda-lifting. The functional languages often allow polymorphic functions (whose arguments may be of different types, similar to mechanism of virtual methods in object oriented programming); these are usually lowered to explicitly passing the dictionaries of the functions.

There are two commonly used semantics for functional languages regarding the passing of the arguments to the functions. The eager evaluation (Scheme, Erlang) means that the arguments are evaluated before they are passed to the function. The lazy evaluation (Haskell) means that they are only evaluated on demand, when the called function needs to know their values. The later approach is theoretically more clean (making the identities like $(\lambda\, x.f)g = f[x := g]$ valid even in cases when evaluation of $g$ does not have to terminate), but significantly less efficient to implement (it is necessary to create thunks for unevaluated expressions whenever we pass an argument to a function) and the actual control flow is hard to predict, making the programs difficult to optimize. Nevertheless the methods of compilation of these languages are similar—even eager languages must be able to suspend evaluation of expressions when partially applied functions are passed as arguments, although their advantage is that from the type information they can often derive whether this occurs.

The examples of low-level abstract machines used for compilation of the functional languages are for example G-machine ([A84], [J84]) or the Three Instruction Machine ([FW87]). Despite the significant differences, the basic operations include manipulation and querying of the data structures (to enable their construction and pattern-matching), the param-

eter passing and the function calls.

There are two basic models used to call functions. The "eval-apply" model works by evaluating the function, in case of eager languages evaluating the arguments and applying the functions to the arguments. The "push-enter" model is used for lazy languages; it pushes the arguments of the function to the parameter stack, then enters evaluation of the function. There is no return after the end of the function in this case.

For lazy languages, it is necessary to ensure sharing. For example in $(\lambda\, x.x + x)\, f$ we want $f$ to be evaluated just once. This means that when we finish evaluation of a thunk, we need to arrange its value to be rewritten by the result.

For the reasonable performance, there are several problems to be solved. We need to arrange for a sane argument/return value passing conventions using registers, and to make this work together with the argument stack. The partially applied functions present in the form of the thunks must have a mechanism how to apply additional arguments to them (by copying the thunk, creating the linked lists of arguments, or combination of both depending on the size of the thunk). We need to distinguish between already evaluated values and thunks, which may be done either by tagging or by keeping even evaluated values as trivial thunks that just return their value. Similarly either tagging or selector function needs to be used for distinguishing the variants of values during pattern matching.

For more involved description of these decisions as well as other issues with compilation of (especially lazy) functional languages see e.g. [J92] or [JL92].

Usually either some low-level procedural language (C) or assembler is used as the target language. The former is more portable and usually produces a better code due to the low-level target specific optimizations done by the C compiler, assuming that there is a possibility to ensure that the important values (for example stack and heap boundaries) are kept in registers all the time. The later is more complicated, but it gives a better compilation speed.

Of course these are just the basic approaches, which need to be enhanced by various low-level optimizations both on the source and the resulting code. In result, the performance of the more practical languages (functional with eager evaluation, logic without implicit backtracking) is in general the same as of the higher-level procedural languages. The performance of the languages that more precisely match the clean theoretical ideas (functional languages with lazy evaluation) tends to be worse by a factor of 2–5.

## 2 Declarative Language Optimizations

Many of the optimizations in the declarative languages try to eliminate the inefficiencies of the models described above. We omit the description of the low-level optimizations completely, since they clearly are not relevant, and also require a detailed knowledge of the particular model. The more high-level examples include

- Deforestation ([W90], [G96]) attempts to eliminate the need for creating temporary structures in clean declarative languages, where by clean we mean that the functions cannot have side effects, and consequently it is impossible to rewrite the data in-place (i.e. when you need to modify something, you must create its copy). This is remotely similar to loop fusion, although the main effect we want to obtain by it

is quite different. For example programmer would usually write $map\, f\, (map\, g\, l)$ to apply two functions $f$ and $g$ to the list $l$. This however requires creation of the temporary list for the result of $map\, g\, l$. Deforestation rewrites this to $map\, (f\, .\, g)\, l$, which produces the result directly.

- In lazy languages, strictness analysis determines whether the arguments of the function will always be evaluated. If this is true, we may evaluate them directly and we do not have to create thunks for them.

- In logic languages, analysis of whether the predicate is deterministic (i.e. always giving just a single solution) can be used to omit the code necessary to handle backtracking.

Note that despite of the fact that the mentioned optimizations are quite high-level and they require nontrivial analyzes, they are obviously specific for the particular family of models and they do not seem to be directly applicable to the procedural programming languages (possibly with the exception of the limited version of deforestation in languages including map-like commands, but usually this can be handled by loop fusion as well).

Still there are some optimizations that seem relevant. The following sections are dedicated to them.

## 3 Recursion Elimination

Since the declarative languages do not in general include loop-like statements, all such constructions are achieved using recursion. Therefore it is important to handle recursion efficiently, and replace it by standard iterative loops as possible.

The simplest case is the tail call elimination (replacing the calls after that the function exits immediately by ordinary jumps). This optimization is standard in procedural languages as well, so we will not describe it in detail here. Instead we focus on some useful improvements to this basic scheme (most of them based on [LS99]):

- Provided that we have sufficient knowledge about the operations done after the call, we may be able to reorganize the computations and remove the recursion. Consider for example

```
f(x): if x == 0 then
        return 1;
      else if x % 2 then
        return 5 * f (x - 1);
      else
        return 3 + f (x - 2);
```

This can be transformed into

```
f(x): m = 1;
      a = 0;

      start:
      if x == 0 then
        return m + a;
      else if x % 2 then
        {
          x--;
          m *= 5;
          goto start;
        }
      else
        {
          x -= 2;
          a += 3 * m;
          goto start;
        }
```

This is what we currently do in GCC. Note that to get this result we needed a

plenty of knowledge about nature of the operations $+$ and $*$—distributive law, associativity, commutativity, and existence of neutral elements. In the special case when just a single such operation is used and all non-recursive exits return the same value, associativity (and in some cases commutativity) would be sufficient, but even these are quite hard to check and this restricts this approach to just a limited class of programs.

- Provided that we have a sufficient knowledge about the operations done before the call, we may turn the recursion into iteration without changing the order of operations executed, in this way:

```
f(x): if x <= 0 then
        return 1;
      else
        return g (x, f(x-1));
```

into

```
f(x): if x <= 0 then
        return 1;
      r = 1;
      for (ax = 0; ax != x; )
        {
          ax++;
          r = g (ax, r);
        }
      return r;
```

We need the function to be in somewhat restricted shape to perform this transformation (see [LS99] for details, most importantly no unhandled code can be executed before the recursive call), the increment ($x \leftarrow x - 1$) needs to be invertible (see [HK92] for some theory on the topic; in practice probably just the simple induction variable-like increments could be handled), and we need to be able to determine the start value of the counter. On the other hand effects of $g$ (or whatever code might be there) are unrestricted, since we do not change the order of execution of the calls to $g$.

- The situation becomes more complicated when one of the conditions above is not satisfied, but still sometimes it can be handled. For example if there are more exits and some code executed before the recursive call, we can still optimize the function by creating two loops—one executing the stuff done before the call and coming all the way down to the appropriate exit case, the second one identical to the one described in the previous case. This requires that those two pieces of code do not communicate with each other except for the value of the counter.

- Finally if there are also multiple recursive calls or we are unable to derive the inverse of the increment, we may eliminate the recursion by maintaining the stack ourselves. This gives less benefits than the previous cases, but still we only need to save variables that are live across the call, we save on the cost of the call itself (including parameter passing) and we expose the loops to the loop optimization (but see also the following section regarding the subject).

## 4 Loop Optimizations

As mentioned in the previous sections, loops in the declarative programming languages are almost exclusively expressed through recursion. Although we have demonstrated several powerful techniques for eliminating the recursion, in fact in many cases these approaches fail. It is therefore useful to be able to optimize such loops.

The interprocedural loops can be detected using the standard algorithms applied to the graph obtained by taking union of a callgraph and the control flow graphs of the functions. Since the strongly connected components of mutually recursive functions are usually entered at one point, the concept of the natural loop seems to be sufficiently general to cover the most important cases. On a side note, considering the ordinary intraprocedural loops in context of this graph may be useful as well, for example in order to be able to estimate instruction and data cache effects.

For the interprocedural loops the invariant motion and redundancy elimination seem to be the easiest to apply and the most useful from the standard optimizations (some other like strength reduction could work as well, but only under assumptions that are quite unlikely to happen). The implementation is straightforward:

- Determine the parameters and global variables that are just passed through unchanged, and propagate the information to determine those that are invariant.
- Run the function local invariant analysis.
- Move the computation of the invariants out of the loop. It may be necessary to create a wrapper around the header function of the loop (which is analogical to creating the preheaders) unless it is called from only one place outside of the loop.

If the moved invariants are expensive, we can create a global variable for them, since the loads from the memory will still pay up. Otherwise we must be able to reserve a register for them across the functions (which should be possible in GCC with just minor modifications). Obviously we must be very careful about the register pressure in this case.

On intraprocedural level, there are other interesting high-level loop optimizations. For example incrementalization (usage of the values computed in the previous iterations—see [LSLR02]) can be used to transform code like

```
for (i = 0; i < 100; i++)
  {
    sum[i] = 0;
    for (j = 0; j < i; j++)
      sum[i] += a[j];
  }
```

into

```
sum[0] = 0;
for (i = 1; i < 100; i++)
  sum[i] = sum[i - 1] + a[i - 1];
```

thus achieving an asymptotic speedup.

## 5 Inlining and Specialization

The declarative programs tend to be composed of small functions. To make the intraprocedural optimizations useful, it is necessary to perform function inlining intensively. See for example [JM02] for discussion of applicability and problems connected with inlining in lazy functional languages.

Also generic functions and usage of callback-type functions is a norm in these languages. They obviously carry a significant penalties for their usage with them—such functions are harder to optimize and often require passing of a partially applied function arguments or function dictionaries, which is not cheap. To overcome this, function specialization (also called cloning) is necessary. This optimization consists of creating duplicates of functions depending on the call site and optimizing them

for the particular values or types of arguments. See for example [FPP00] and [P97] for more details.

Both of these optimizations are studied in the context of procedural languages as well, and at least some of the implementation issues should be covered by Hubicka ([H04]), so we make just a few minor points here.

- It is necessary to interleave inlining and specialization with other optimizations. The approach that tries to get the best performance would have to at least optimize the functions locally (to get rid of unreachable calls, decrease the function sizes and propagate the information about values of arguments), then inline/specialize the optimized function bodies, rerun the optimizations, then run inlining and specialization again (to exploit the interprocedural information taken into account due to the first inlining pass), then again rerun the optimizations. Of course this may get compile-time expensive, so other variations of the scheme may be useful at the lower optimization levels.

- The code growth is the major problem with both of these optimizations, since it has bad effects on the instruction caches. To overcome the problems, having a call-graph with profiling information is very useful—we then may optimize just the intensively used functions and function call sites.

  An implementability note: in fact we have basically everything needed in GCC with the current profiling scheme—it would be sufficient to tag the call sites in a unique way and to emit the call $\mapsto$ basic block map before profiling (similar to the current `.gcno` files). The other possibility would be the early instrumentation of the call sites. The main problem currently is that both of these possibilities interfere with the ordinary profiling. The former possibility needs the function inlining not to be run in the training pass. The later needs to be done before inlining and changes the code, so it cannot be done simultaneously with the ordinary instrumentation that is done after inlining. One of the solutions is to do the both at the same time, which again needs the inlining to be done later in the compilation process.

- Other possibility is to inline just the relevant parts of the function (so-called partial inlining). If we identify that there is a short hot part in the inlined function, we may copy just this part and put the rest into a separate shared function. This is useful especially for functions that cache their results, or handle common special cases in advance.

- There are several approaches to limit the code growth with specialization. One of them is to first specialize all possible occurrences, optimize the bodies and then reshare those for that we were not able to improve the code sufficiently. The other one is to identify applicability of optimizations in advance and just specialize those for that we believe it will be useful.

  None of these approaches seems to be suitable for GCC. The former obviously wastes a lot of compile time, and detecting the non-improved instances also would not be straightforward. The later is difficult to implement (it would need to have a separate analysis for each optimization) and unreliable. The realistic approach seems to identify the obvious possibilities (functions with callback arguments, boolean flags passed to them and guarding parts of the code in their bodies, con-

stant integer parameters used as bounds of the loops, for example) and specialize just these. Additionally attribute mechanism could be used to give the programmer a possibility to tell that he wants the function to be specialized for the specified arguments.

- Interprocedural loop optimizations mentioned in the previous section as well as other optimizations may need to create simple wrappers around the functions. This may be useful in other cases as well. For example we do changing of calling conventions for static functions. If we detect that an exported function is often called locally (from the callgraph profiling, or just by determining that it is called recursively), creating an exported wrapper just calling its local instance may pay up. The other possibility would be to clone a local copy of it, but this would usually grow the code much more.

- Specialization on the constant arguments and specialization on types of arguments is the most commonly used option. Other possibility is to specialize according to the information from value range propagation or other analyzes, but currently there is not the infrastructure necessary to exploit this possibility in GCC.

## 6 Data Structure Analysis and Optimizations

This section describes some optimizations related to data structures used by the programs. They are mostly relevant for higher-level languages. Applying them for low-level procedural languages like C is complicated by the following issues:

- The layout of data structures is precisely given in C (by ABI for the particular architecture). This makes it only possible to alter it in cases when we are able to prove that there are no external references to the structure, and that the program does not rely on a particular layout of the data structure.

- The exposed pointer arithmetics makes all analyzes close to impossible. It is not easy to handle even the basic prerequisite for all the optimizations—alias analysis—in a satisfactory way.

Despite of these problems, some of the optimizations have also been studied in the context of procedural languages, since the memory access times are a bottleneck in many applications. For these reasons we provide only a short descriptions of several chosen optimizations, with references to relevant papers:

- Array reshaping changes the layout of arrays (order of indices and their dimensions) to improve the effectiveness of caches. See for example [G00] that implements the array padding (changing dimensions of an array by adding unused elements). Memory layout optimizations can be with advantage used together with loop nest optimizations ([CL95]).

- Linked lists are the basic structures used in the declarative languages. Therefore much of effort is directed to their optimizations. Although the procedural programs often also work with linked lists, usability of the techniques mentioned below is quite limited due to problems with identifying this pattern (see [CAZ99] for overview of such an analysis).

  If we are able to detect usage of linked lists, we may use the knowledge in several ways. We may arrange the mem-

ory to be allocated sequentially, thus improving cache behavior ([LA02] does a similar optimization, but without trying to identify precisely the access pattern). In cases when the list is accessed in a queue-like fashion only, we may also change the representation of the list, for example by putting several consecutive elements of the list to an array. This decreases the amount of memory needed by eliminating the successor (and possibly predecessor) pointers, allows more effective traversal of the lists (in loops controlled by a normal induction variable) and consequently increases efficiency of loop optimizations.

- Declarative languages often support use of temporary data structures (especially linked lists) in an almost transparent fashion, leading to initially quite ineffective code. This makes optimizations like dead store elimination for partially dead data structures necessary; see for example ([L98]).

- For some of these optimizations a memory access profiling might be useful. In the most expensive variant, the full list of all memory accesses tagged with the corresponding references to the source program and perhaps also exact values of indices for array accesses is recorded in the training pass. This data together with a memory cache model provides a quite exact base for determining the parameters for all cache directed optimizations. The optimizations that require exact analysis of the access pattern of course cannot be based just on this empiric data, but they may at least use it to locate the opportunities and to evaluate their usefulness.

  Obviously recording all memory accesses may turn quite expensive, so recording just the relevant information may be necessary. For implementation details in GCC see for example the recent works of the author and Caroline Tice on profiling driven array prefetching.

## Conclusions

Several of the techniques we have presented appear to be implementable GCC (note that at least for some of them this would not be a simple task at all, however) and useful enough so that they might bring measurable speedups, especially

- improvements of the recursion elimination
- data access profiling and data structure reorganization
- call graph profiling
- function cloning and specialization

There are other optimizations that seem to be "cool" and implementable in the GCC framework, although they are only applicable in very special cases. They probably would not improve the performance much by themselves, but implementing them might be interesting from theoretical reasons. In some cases there also seem to be a chance to generalize them and thus improve their applicability. They include

- interprocedural loop optimizations
- loop incrementalization
- linear structures analysis and related optimizations

Of course there also are many optimizations that probably are only useful in context of

declarative languages (deforestation, strictness analysis and unboxing, etc.)

The list of the optimizations can by no means considered complete. I have filtered out the low-level optimizations that seem too specific for the particular compilation model. I also am not deeply involved in the declarative language compilation research, so I probably missed quite a few relevant techniques; I would be grateful to anyone pointing my attention to them.

## Acknowledgments

This research was supported by SuSE Labs.

## References

[W83] D.H.D. Warren, *An abstract Prolog instruction set*, Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[A84] L. Augustsson, *A Compiler for Lazy ML*, Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, August 1984, pp. 218–227.

[J84] Johnsson, T., *Efficient compilation of lazy evaluation*, Proceedings of the SIGPLAN'84 Symposium on Compiler Construction, June 1984, pp. 58–69.

[FW87] J. Fairbairn, S. Wray, *TIM—a simple lazy abstract machine to execute supercombinators*, Functional Programming Languages and Compiler Architecture, LNCS 274, Springer Verlag.

[W90] P. Wadler, *Deforestation: transforming programs to eliminate trees*, Theoretical Computer Science 73 (1990), 231–248.

[HK92] P.G. Harrison, H. Khoshnevisan, *On the synthesis of function inverses*, Acta Informatica, 29(3):211–239, 1992.

[J92] S.P. Jones, *Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine*, Journal of Functional Programming 2(2) (April 1992), pp. 127–202.

[JL92] S.P. Jones, D. Lester, *Implementing functional languages: a tutorial* Published by Prentice Hall, 1992.

[R94] P. van Roy, *Issues in implementing logic languages*, Slides, École de Printemps, Châtillon/Seine, May 1994.

[CL95] M. Cierniak, W. Li, *Unifying Data and Control Transformations for Distributed Shared-Memory Machines*, Proceedings of the ACM SIGPLAN Conference of Programming Design and Implementation (PLDI'95), La Jolla, California, USA, 1995.

[G96] A. Gill, *Cheap deforestation for non-strict functional languages*, PhD thesis, University of Glasgow, Jan 1996.

[P97] G. Puebla, *Advanced Compilation Techniques based on Abstract Interpretation and Program Transformation*, Ph.D. Thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, November 1997.

[L98] Y.A. Liu, *Dependence Analysis for Recursive Data*, Proceedings of the 1998 International Conference on Computer Languages, 1998.

[CAZ99] F. Corbera, R. Asenjo, E.L. Zapata, *New shape analysis techniques for automatic parallelization of C codes*, Proceedings of the 13th International

Conference on Supercomputing, Rhodes, Greece, 1999.

[LS99] Y.A. Liu, S.D. Stoller, *From recursion to iteration: what are the optimizations?*, Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, pp. 73–82.

[FPP00] F. Fioravanti, A. Pettorossi, M. Proietti, *Rules and Strategies for Contextual Specialization of Constraint Logic Programs*, Electronic Notes in Theoretical Computer Science, Vol. 30 (2) (2000)

[G00] C. Grelck, *Array Padding in the Functional Language SAC*, Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 2000, June 24–29, 2000, Las Vegas, Nevada, USA

[DC01] D. Diaz, P. Codognet, *Design and Implementation of the GNU Prolog System*, Journal of Functional and Logic Programming (JFLP), Vol. 2001, No. 6, October 2001.

[HS02] F. Henderson, Z. Somogyi, *Compiling Mercury to high-level C code*, Proceedings of the 2002 International Conference on Compiler Construction Grenoble, France, April 2002.

[JM02] S.P. Jones, S. Marlow, *Secrets of the Glasgow Haskell Compiler inliner*, Journal of Functional Programming 12(4), July 2002, pp393-434.

[LA02] C. Lattner, V. Adve, *Automatic Pool Allocation for Disjoint Data Structures*, ACM SIGPLAN Workshop on Memory System Performance (MSP), Berlin, Germany, June 2002.

[LSLR02] Y. Liu, S. Stoller, N. Li, T. Rothamel, *Optimizing Aggregate Array Computations in Loops*, Technical Report TR 02-xxxx, Computer Science Department, State University of New York at Stony Brook, Stony Brook, New York, July 2002.

[H04] J. Hubicka, *The GCC call graph module: a framework for inter-procedural optimization*, accepted to the 2nd GCC & GNU Toolchain Developers' Summit, June 2004.

# High-Level Loop Optimizations for GCC

*Daniel Berlin*
IBM T.J. Watson Research Center
dberlin@us.ibm.com

*David Edelsohn*
IBM T.J. Watson Research Center
dje@watson.ibm.com

*Sebastian Pop*
Centre de recherche en informatique, École des mines de Paris
sebastian.pop@cri.ensmp.fr

## Abstract

This paper will present a design for loop optimizations using high-level loop transformations. We will describe a loop optimization infrastructure based on improved induction variable, scalar evolution, and data dependence analysis. We also will describe loop transformation opportunities that utilize the information discovered. These transformations increase data locality and eliminate data dependencies that prevent optimization. The transformations also can be used to enable automatic vectorization and automatic parallelization functionality.

The TreeSSA infrastructure in GCC provides an opportunity for high level loop transforms to be implemented. Prior to the Loop Nest Optimization effort described in this paper, GCC has performed no cache reuse, data locality, parallelization, or loop vectorization optimizations. It also had no infrastructure to perform data dependence analysis for array accesses that are necessary to apply these transformations safely. We have implemented data dependence analysis and linear loop transforms on top of TreeSSA, which provides the following features:

1. A data dependence framework for determining whether two data references have a dependence. The core of the dependence analysis is a new, low-complexity algorithm for the recognition of scalar evolutions that tracks induction variables across a def-use graph. It is used to determine the legality of various transformations, including the vectorization transforms being implemented, and the matrix based transformations.

2. A matrix-based transformation method for rearranging loop nests to optimize locality, cache reuse, and remove inner loop dependencies (to help vectorization and parallelization). This method can perform any legal combination of loop interchange, scaling, skewing, and reversal to a loop nest, and provides a simple interface to doing it.

## 1 Introduction

As GNU/Linux tackles high-performance scientific and enterprise computing challenges, GCC (the GNU Compiler Collection)—the GNU/Linux system compiler—is challenged as well. Modern computer processors and systems are implemented with advanced features that require greater compiler assistance to achieve high performance. Many techniques

developed for vector and parallel architectures have found new application to superscalar and VLIW computer architectures, and to systems with large memory latencies, more complicated function unit pipelines, and multiple levels of memory caches.

The TreeSSA optimization infrastructure[11] in GCC provides an enhanced framework for program analysis. Improved data dependence information allows the compiler to transform an algorithm to achieve greater locality and improved resource utilization leading to improved throughput and performance.

The GCC Loop Nest Optimizer joins a powerful loop nest analyzer with a matrix transformation engine to provide an extensible loop transformation optimizer that addresses unimodular and scaling operations. The data dependence analyzer is based on a new algorithm to track induction variables without being limited to specific patterns. The matrix transformation functionality uses a building block design that allows many of the standard toolbox of optimizations to be implemented. A similar matrix toolkit is used by proprietary commercial compilers. The pieces form a clean and maintainable design, avoiding an ad hoc set of optimizers with similar technical requirements.

## 2 Scalar Evolutions

After the *genericization* and *gimplification*, the loop structures of the compiled language are transformed into lower level constructs that are common to the imperative languages: three address assignments, gotos and labels. In order to retrieve the classic representation of loops from the GIMPLE representation[9], the natural loop structures are detected, as described in the Dragon Book [1], then based on the analysis of the instructions contained in the loops bodies, the indexes and the bounds of loops are detected.

We describe in this section the algorithm used for analyzing the properties of the scalar variables updated in a loop. The main extracted properties are the number of iterations of a loop, and a form that allows a fast evaluation of the values of a variable for a given iteration. Based on these two properties, it is possible to extend the copy constant propagation pass after the crossing of a loop, and the elimination of redundant checks. A further analysis extracts a representation of the relations between the reads and the writes to the memory locations referenced by arrays, and the classic data dependence tests.

### 2.1 Representation of the Program

The analyzed program is in *Static Single Assignment* form [10, 5], that ensures the uniqueness of a variable definition, and a fast retrieval of the definition from a use. These properties have lead to the design of an efficient algorithm that extracts the scalar evolutions in a bidirectional, non-iterative traversal of the control-flow graph.

### 2.2 Chains of Recurrences

The information extracted by the analyzer is encoded using the chains of recurrences (chrecs) representation proposed in [3, 6, 17, 14, 13]. This representation permits fast evaluations of a function for a given integer point, using the Newton's interpolation formula. In the following, we present an intuitive description of the chrecs based on their interpretation, then the link between the notation of the chrecs and the semantics of the polynomial functions.

```
r1 = 0
r2 = 1
r3 = 2
loop (ℓ1)
| r1 += r2
| r2 *= r3
end1:
```

Figure 1: Univariate evolution

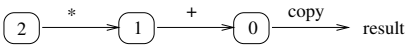



Figure 2: Data-flow interpretation

```
r4 = 9
r5 = 8
r6 = 7
loop (ℓ2)
| loop (ℓ3)
| | r6 += r5
| end3:
| r5 += r4
end2:
```

Figure 3: Multivariate

### 2.2.1 Interpretation of Chrecs

The main property modeled by the chrecs is the effect of the iterative execution of a program on storage points. Each storage point contains an initial value on the entry of a loop. The stored value evolves during the execution of the loop following the operations of the updating statements. The description of the updating expressions is embedded in the chrecs representation, such that it is possible to retrieve a part of the original program from the chrec representation. In other words, only the interesting scalar properties are selected, and the undecidable scalar properties are abstracted into the unknown element. In the following, the chrecs representation is illustrated by intuitive examples based on two interpretation models: using a register based machine, and a data-flow machine.

In the register based machine, the coefficients of a chrec are stored in registers. Then, the value of a register is updated at each iteration of a loop, using the operation specified in the chrec on its own value and the value of the register on its right. The first example illustrates the interpretation of a chrec that vary in a single loop.

**Example 1 (Univariate chrec on register machine)** *Figure 2.2.1 illustrates the interpretation of the chrec* $\{0, +, \{1, *, 2\}_1\}_1$. *The registers* $r1$, $r2$, *and* $r3$ *are initialized with the coefficients of the chrec. Then, the registers are updated in the loop specified in index of the chrec: loop* $1$. *The register* $r2$ *is updated in the loop, and its evolution is described by the chrec* $\{1, *, 2\}_1$. $r1$ *is accumulating the successive values of* $r2$ *starting from its initial value* $0$, *and consequently it is described by the chrec* $\{0, +, \{1, *, 2\}_1\}_1$.

Another intuitive description of the chrecs is given by the data-flow model: the nodes of an oriented graph contain the initial conditions of the chrec, while the oriented edges transfer information from a node to another and perform an operation on the operands. Figure 2 illustrates the data-flow machine that interprets the chrec from Example 1.

Finally, the last example illustrates the interpretation of a chrec that vary in two loops.

**Example 2 (Multivariate chrec on register machine)** *In Figure 2, the register* $r6$ *can be described by the multivariate scalar evolution* $\{7, +, \{8, +, 9\}_2\}_3$. *The value of* $r6$ *is incremented at each iteration of loop* $3$ *by the value contained in* $r5$ *that vary in loop* $2$.

In the register based machine, the value of a chrec at a given integer point is computed by successively evaluating all the intermediate values. The initial values of the chrec are stored in registers that are subsequently updated at each iteration step. One of the goals of the analyzer is to detect these iterative patterns, and then to recognize, when possible, the computed function. The link between the

chrecs and the classic polynomial functions is described in the next subsection.

#### 2.2.2 Semantics of Chrecs

As described in the previous works [3] Newton's interpolation formula is used for fast evaluation of the chrec at a given integer point. The evaluation of the chrec $\{c_0, +, \ldots, +, c_k\}$, at an integer point $x$ is given by the formula

$$\{c_0, +, \ldots, +, c_k\}(x) = \sum_{i=0}^{k} c_i \binom{x}{i}$$

with $c_0, \ldots, c_k$ integer coefficients. In the peculiar case of linear chrecs, this formula gives

$$\{base, +, step\}(x) = base + step \cdot x$$

where $base$ and $step$ are two integer constants. As we will see, it is possible to handle symbolic coefficients, but the above formula for evaluating the chrecs is not always true.

#### 2.2.3 Symbolic Chrecs

We have extended the classic representation of the scalar evolution functions by the use of parameters, that correspond to unanalyzed variables. The main purpose of this extension is to free the analyzer from the ordering constraints that were proposed in the previous versions of the analyzer. The parameters allow the analyzer to postpone the analysis of some scalar variables, such that the analyzer establishes the order in which the information is discovered in a natural way.

However, this extension leads to a more expressive representation, on which the Newton interpolation formula cannot be systematically used for fast evaluation of the chrec, because some of the parameters can stand for a function. In order to guarantee that all the coefficients of the chrec have scalar (non varying) values, the last step of the analysis fully instantiate all the parameters. When the instantiation fails, the remaining parameters are all translated into the unknown element, $\top$.

#### 2.2.4 Peeled Chrecs

We have proposed another extension of the classic chrecs representation in order to model the variables that have an initial value that is overwritten during the first iteration. For representing the peeled chrecs, we have chosen a syntax close to the syntax of the SSA phi nodes because the symbolic version of the peeled chrec is the loop phi node itself. The semantics of the peeled chrecs is as follows:

$$(a, b)_k = \begin{cases} a, & \text{during the first iteration of loop k,} \\ b & \text{otherwise.} \end{cases}$$

where $a$ and $b$ are two chrecs that can be in a symbolic form. The peeled chrecs are built whenever the loop phi node does not define a strongly connected component over the SSA graph. The next section describes in more details the extraction algorithm.

### 2.3 Extraction Algorithm

Figure 4 presents the algorithm that computes the scalar evolutions for all the loop-$\phi$ nodes of the loops. The scalar evolution analyzer is composed of two parts: ANALYZEEVOLUTION returns a symbolic representation of the scalar evolution, and the second part INSTANTIATEEVOLUTION completes the analysis by instantiating the symbolic parameters. The

*Algorithm:* COMPUTEEVOLUTIONS
*Input:* SSA representation of the procedure
*Output:* a chrec for every variable defined by loop-$\phi$ nodes
  For each loop $l$
    For each loop-$\phi$ node $n$ in loop $l$
      INSTANTIATEEVOLUTION(ANALYZEEVOLUTION($l$, $n$), $l$)

*Algorithm:* ANALYZEEVOLUTION($l$, $n$)
*Input:* $l$ the current loop, $n$ the definition of an SSA name
*Output:* chrec for the variable defined by $n$ within $l$
  v ← variable defined by $n$
  $ln$ ← loop of $n$
  If $n$ was analyzed before Then
    $res$ ← evolution of $n$
  Else If $n$ matches "`v = constant`" Then
    $res$ ← `constant`
  Else If $n$ matches "`v = a`" Then
    $res$ ← ANALYZEEVOLUTION($l$, a)
  Else If $n$ matches "`v = a` $\odot$ `b`" (with $\odot \in \{+, -, *\}$) Then
    $res$ ← ANALYZEEVOLUTION($l$, a) $\odot$ ANALYZEEVOLUTION($l$, b)
  Else If $n$ matches "`v = loop-`$\phi$`(a, b)`" Then
    (notice a is defined outside loop $ln$ and b is defined in $ln$)
    Search in depth-first order a path from b to v:
    (*exist*, *update*) ← DEPTHFIRSTSEARCH($n$, definition of b)
    If (not *exist*) (i.e., if such a path does not exist) Then
      $res$ ← $(\mathsf{a}, \mathsf{b})_l$
    Else If $update$ is $\top$ Then
      $res$ ← $\top$
    Else
      $res$ ← $\{\mathsf{a}, +, update\}_l$
  Else If $n$ matches "`v = condition-`$\phi$`(a, b)`" Then
    $eva$ ← INSTANTIATEEVOLUTION(ANALYZEEVOLUTION($l$, a), $ln$)
    $evb$ ← INSTANTIATEEVOLUTION(ANALYZEEVOLUTION($l$, b), $ln$)
    If $eva = evb$ Then
      $res$ ← $eva$
    Else
      $res$ ← $\top$
  Else
    $res$ ← $\top$
  Save the evolution function $res$ for $n$
  Return the evaluation of $res$ in loop $l$

*Algorithm:* DEPTHFIRSTSEARCH($h$, $n$)
*Input:* $h$ the halting loop-$\phi$, $n$ the definition of an SSA name
*Output:* (*exist*, *update*), *exist* is true if $h$ has been reached
  If ($n$ is $h$) Then
    Return (true, 0)
  Else If $n$ is a statement in an outer loop Then
    Return (false, $\bot$),
  Else If $n$ matches "`v = a`" Then
    Return DEPTHFIRSTSEARCH($h$, definition of a)
  Else If $n$ matches "`v = a + b`" Then
    (*exist*, *update*) ← DEPTHFIRSTSEARCH($h$, a)
    If *exist* Then Return (true, *update* + b),
    (*exist*, *update*) ← DEPTHFIRSTSEARCH($h$, b)
    If *exist* Then Return (true, *update* + a)
  Else If $n$ matches "`v = loop-`$\phi$`(a, b)`" Then
    $ln$ ← loop of $n$
    (notice a is defined outside $ln$ and b is defined in $ln$)
    If a is defined outside the loop of $h$ Then
      Return (false, $\bot$)
    $s$ ← APPLY($ln$, ANALYZEEVOLUTION($ln$, $n$),
        NUMBEROFITERATIONS($ln$))
    If $s$ matches "`a + t`" Then
      (*exist*, *update*) ← DEPTHFIRSTSEARCH($h$, a)
      If *exist* Then
        Return (*exist*, *update* + $t$)
  Else If $n$ matches "`v = condition-`$\phi$`(a, b)`" Then
    (*exist*, *update*) ← DEPTHFIRSTSEARCH($h$, a)
    If *exist* Then Return (true, $\top$)
    (*exist*, *update*) ← DEPTHFIRSTSEARCH($h$, b)
    If *exist* Then Return (true, $\top$)
  Return (false, $\bot$)

*Algorithm:* INSTANTIATEEVOLUTION(*chrec*, $l$)
*Input:* *chrec* a symbolic chrec, $l$ the instantiation loop
*Output:* an instantiation of *chrec*
  If *chrec* is a constant c Then Return c
  Else If *chrec* is a variable v Then
    Return ANALYZEEVOLUTION($l$, v)
  Else If *chrec* is of the form $\{e_1, +, e_2\}_{l'}$ Then
    $i_1$ ← INSTANTIATEEVOLUTION($e_1$, $l$)
    $i_2$ ← INSTANTIATEEVOLUTION($e_2$, $l$)
    Return $\{i_1, +, i_2\}_{l'}$
  Else If *chrec* is of the form $(e_1, e_2)_{l'}$ Then
    $i_1$ ← INSTANTIATEEVOLUTION($e_1$, $l$)
    $i_2$ ← INSTANTIATEEVOLUTION($e_2$, $l$)
    Return $(i_1, i_2)_{l'}$
  Else Return $\top$

Figure 4: Algorithm to compute scalar evolutions

main analyzer is allowed to discover only a part of the evolution information. The missing information is stored under a symbolic form, waiting for a full instantiation. The role of the instantiation is to determine an order for assembling the discovered information. After full instantiation, the extracted information corresponds to the classic chains of recurrences. In the rest of the section we analyze in more details the components of this algorithm, and give two illustration examples.

### 2.3.1 Description of the Algorithm

The cornerstone of the algorithm is the search and reconstruction of the symbolic update expression on a path of the SSA graph. Let us

start with the description of the DEPTHFIRSTSEARCH algorithm. Each step is composed of a look-up of an SSA definition, and then followed by a recursive call of the search algorithm on the symbolic operands. The search halts when the starting loop-$\phi$ node is reached. When analyzing an assignment whose right-hand side is a sum, the search algorithm examines the first operand, and if the starting loop-$\phi$ node is not reachable through this path, it examines the second operand. When one of the operands contains a path to the starting loop-$\phi$ node, the other operand of the sum is added to the update expression, and the result is propagated to the lower search steps together with the reconstructed update expression. If the starting loop-$\phi$ node cannot be found by depth-first search, i.e., when DEPTHFIRSTSEARCH returns (false, $\perp$), we know that the definition does not belong to a cycle of the SSA graph: a peeled chrec is returned.

INSTANTIATEEVOLUTION substitutes symbolic parameters in a chrec. It computes their statically known value, i.e., a constant, a periodic function, or an approximation with intervals, possibly triggering other computations of chrecs in the process. The call to INSTANTIATEEVOLUTION is postponed until the end of the depth-first search, ensuring termination of the recursive nesting of depth-first searches, and avoiding early approximations in the computation of update expressions. Combined with the introduction of symbolic parameters in the chrec, postponing the instantiation alleviates the need for a specific ordering of the computation steps. This is a strong advantage with respect to the method by Engelen [14] based on a topological sort of all definitions. Furthermore, it becomes possible to recognize evolutions in every possible SSA graph, although some of them may not yield a closed form.

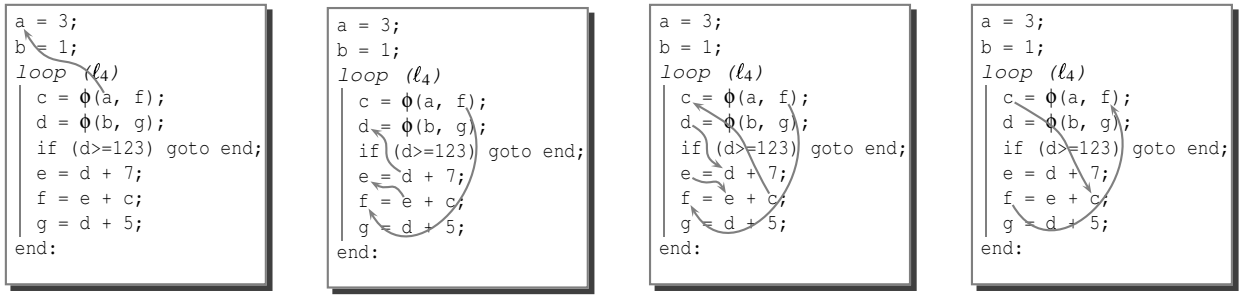
The overall effect of an inner loop may only be computed when the exit value of the variable is a function of the entry value. In such a case, the whole loop is behaving as a macro-increment operation. When the exit condition depends on affine chrec only, function NUMBEROFITERATIONS deduces the number of iterations of the loop. Then we call APPLY to evaluate the overall effect of the inner loop. APPLY implements the efficient evaluation scheme for chrec based on Newton interpolation series (see Section 2.2.2). As a side-effect, the algorithm does indeed compute the loop-trip count for many natural loops in the control-flow graph. Our method recovers information that was lost during the lowering process or syntactically hidden in the source program.

### 2.3.2 Illustration Examples

Let us now illustrate the algorithm on two examples in Figures 5 and 6. In addition to clarifying the depth-first search and instantiation phases of the algorithm, this will exercise the recognition of polynomial and multivariate evolutions.

**First example.** The depth-first search is best understood with the analysis of `c = `$\phi$`(a, f)` in the first example. The SSA edge of the initial value exits the loop, as represented in Figure 5.(1). Here, the initial value is left in a symbolic form, but GCC would replace it by 3 through constant propagation.

To compute the parametric evolution function of `c`, the analyzer starts a depth-first search algorithm, as illustrated in Figure 5.(2). We follow the update edge `c`$\rightarrow$`f` to the definition of `f` in the loop body: assignment `f = e + c`. The depth-first algorithm follows the first operand, `f`$\rightarrow$`e`, reaching the assignment `e = d + 7`, and finally follows the edge `e`$\rightarrow$`d` that leads to a loop-$\phi$ node of the same loop.




(1) Initial condition edge

(2) Search "c"

(3) Found the halting phi

(4) the "returning path"

Figure 5: The first example

Since this is not the loop-$\phi$ node from which the analyzer has started the depth-first search, the search continues on the other operands that were not yet analyzed: back on `e = d + 7`, operand `7` is a scalar and there is nothing more to do, then back on `f = e + c`, the edge `f`→`c` is followed to the starting loop-$\phi$ node, as illustrated in Figure 5.(3).

At this point, the analyzer has found the strongly connected component that corresponds to the path of iterative updates. Following this path in execution order, as illustrated in Figure 5.(4), the analyzer builds the update expression as an aggregation of the operands that are not on the updating path: in this example, the update expression is just `e`. As a result, the analyzer assigns to the definition of `c` the parametric evolution function $\{\mathtt{a}, +, \mathtt{e}\}_1$.

The instantiation of the parametric expression $\{\mathtt{a}, +, \mathtt{e}\}_1$ starts with the substitution of the first operand of the chrec: $\mathtt{a} = 3$, then the analysis of `e` is triggered. First the assignment `e = d + 7` is analyzed, and since the evolution of `d` is not yet known, the edge `e`→`d` is taken to the definition `d = `$\phi$`(b, g)`. Since this is a loop-$\phi$ node, the depth-first search algorithm is used as before and yields the evolution function of `d`, $\{b, +, 5\}_1$, and after instantiation, $\{1, +, 5\}_1$. Finally the evolution of `e = d + 7` is computed: $\{8, +, 5\}_1$. The final result of the instantiation yields the polynomial chrec of `c`: $\{3, +, 8, +, 5\}_1$.

```
h = 3;
loop (ℓ5)
  i = ϕ(h, x);
  loop (ℓ6)
    j = ϕ(i, k);
    k = j + 1;
    t = j - i;
    if (t>=9) goto end6;
  end6:
  x = k + 3;
  if (x>=123) goto end5;
end5:
```

Figure 6: Second example

**Second example.** We will now compute the evolution of `x` in the nested loop example of Figure 6, to illustrate the recognition of multivariate induction variables and the computation of the trip count of a loop. The first step consists in following the SSA edge to the definition of `x`. Consider the right-hand side of the definition: since the evolution of `k` along loop 5 is not yet analyzed, we follow the edge `x`→`k` to its definition in loop 6, then `k`→`j` ending on the definition of a loop-$\phi$ node.

At this point we know that `j` is updated in loop 6. The initial condition `i` is kept under a symbolic form, and the iteration edge `j`→`k`

is followed in the body of loop 6. The depth-first search algorithm starts from right-hand side of the assignment `k = j + 1`: following the edge `k`→`j` we end on the loop-$\phi$ node from which we have started the search, meaning that the search succeeded. Back on the path `j`→`k`→`j`, the analyzer gathers the evolution of `j` along the whole loop, an increment of 1, and ends on the following symbolic chrec: $\{\mathtt{i}, +, 1\}_6$.

From the evolution of `j` in the inner loop, the analyzer determines the overall effect of loop 6 on `j`, that is the evaluation of function $f(n) = n + \mathtt{i}$ for the number of iterations of loop 6. Fortunately, the exit condition is the simple expression `t>=9`, and the chrec for `t` (or `j - i`) is $\{0, +, 1\}_6$, an affine (non-symbolic) expression. It comes that 10 iterations of loop 6 will be executed for each iterations of loop 5. Calling APPLY(6, $\{\mathtt{i}, +, 1\}_6$, 10) yields the overall effect `j = i + 10`.

The analyzer does not yet know the evolution function of `i`, and consequently it follows the SSA edge to its definition: `i =` $\phi$`(h, x)`. Since this is a loop-$\phi$ node, the analyzer must determine its evolution in loop 5. We ignore the edge to the initial condition, and walk back the update edge, searching for a path from `i` to itself.

First, edge `i`→`x` leads to the statement `x = k + 3`, then following the SSA edge `x`→`k`, we end on a statement of the loop 6. Again, edge `k`→`j` is followed, ending on the definition of `j` that we have already analyzed: $\{\mathtt{i}, +, 1\}_6$. The depth-first search selects the edge `j`→`i`, associated with the overall effect statement `j = i + 10` that summarizes the evolution of the variable in the inner loop. We finally reached the starting loop-$\phi$ node `i`. From this point, the path is walked back gathering the stride of the loop: 10 from the assignment `j = i + 10`, then 1 from the assignment `k = j + 1`, and 3 from the last assignment on the return path. We have computed the symbolic chrec of `i`: $\{\mathtt{h}, +, 14\}_5$.

The last step consists in the propagation of this evolution function from the loop-$\phi$ node of `i` to the original node of the computation: the definition of `x`. Back from `i` to `j`, we can partially instantiate its evolution: a symbolic chrec for `j` is $\{\{\mathtt{h}, +, 14\}_5, +, 1\}_6$. Then back to `k = j + 1` we get a symbolic chrec for `k`: $\{\{\mathtt{h} + 1, +, 14\}_5, +, 1\}_6$; and finally back to `x = k + 3`, we get a symbolic chrec for `x`: $\{\mathtt{h} + 14, +, 14\}_5$. A final instantiation of `h` yields the closed form of `x` and all other variables.

As we have seen, the analyzer computes the evolution functions on demand, and caches the discovered informations for later queries occurring in different analyzes or optimizations that make use of the scalar evolution information. In the next section, we describe the applications that use the informations extracted by the analyzer.

### 2.4 Applications

Scalar optimizations have been proposed in the early days of the optimizing compilers, and have evolved in speed and in accuracy with the design of new intermediate representations, such as the SSA. In this section we describe the extensions to the classic scalar optimization algorithms that are now enabled by the extra information on scalar evolutions. Finally, we give a short description of the classic data dependence tests.

#### 2.4.1 Condition Elimination

In order to determine the number of iterations in a loop, the algorithm computes the first iteration that does not satisfy the condition that

keeps the execution inside the loop. This same algorithm can be used on other condition expressions that don't keep the loop exit, such that the algorithm determines the number of iterations that fall in the then or in the else clauses. Based on the total number of iterations in the loop it is then possible to determine whether a branch is always taken during the execution of the loop, in which case the condition can be eliminated together with the dead branch.

Another approach for the condition elimination consists in using symbolic techniques for proving that two evolutions satisfy some comparison test for all the iterations of the loop. In the case of an equality condition, the algorithm is close to the value numbering technique, and is described in the next subsection.

#### 2.4.2 Value Numbering

The value numbering is a technique based on a compile-time classification of the values taken at runtime by an expressions. The compiler determines the inclusion property of an expression into a class based on the results of an analysis: in the classic algorithms, the analysis is a propagation of symbolic AST trees [10, 12].

Using the information extracted by the scalar evolution, the classification can be performed not only on constants and symbols, but also on evolution functions, or on the scalar values determined after crossing the loop.

#### 2.4.3 Extension of the Constant Propagation

The field of action of the classic conditional constant propagation (CCP) is limited to code that does not contain loop structures. When the scalar evolution analyzer is asked for the evolution function of a variable after crossing a loop with a static count, it computes a scalar value, that can be further propagated in the rest of the program. This removes the restriction of the classic CCP, where constants are only propagated from their definition to the dominance frontier.

#### 2.4.4 Data Dependence Analysis

Several approaches have been proposed for computing the relations between the reads and the writes to the memory locations referenced by arrays. The compiler literature [4, 15, 10] describes loop normalizations, then the extraction of access functions by pattern matching techniques, while more recent works [16], rely on the discovery of monotonicity properties of the accessed data. An important part of the efficiency of these approaches resides in the algorithm used for determining the memory access patterns, while the subscript intersection techniques remain in the same range of complexity.

Our data dependence analyzer is based on the classic methods described in [4, 2]. These techniques are well understood and quite efficient with respect to the accuracy and the complexity of the analysis. However, our data dependence analyzer can be extended to support the newer developments on monotonicity properties proposed by Peng Wu *et al*. [16], since the scalar evolution analyzer is able to extract not only chrecs with integer coefficients, but also evolution envelopes, that occur whenever a loop contains updating expressions in a condition clause. In the following we shortly describe the classic data dependence analyzer, and show how to extend it for handling the monotonicity informations exposed by the scalar analyzer.

A preliminary test, that avoids unnecessary further computations, classifies the relation between two array accesses as *non dependent*

when their base name differ. Thus, the remaining dependence tests consider only tuples of accesses to the same base name array.

The first test separately analyzes each tuple of access functions in each dimension of the analyzed array. This tuple is in general called a subscript. A basic test classifies a subscript following the number of loops in which it is varying. The three classes of subscripts, constants, univariate, or multivariate, have different specific dependence tests that avoids the use of the multivariate generic solver.

The iterations for which a subscript access the same element, or conflicting iterations, are computed using a classic Diophantine[1] equation solver. The resulting description is a tuple of functions that is encoded yet again using the chrecs representation. Banerjee presents a formal description [4] of the classic data dependence tests that we just sketch in this paper. The basic idea is to find a first solution (or the first conflicting iteration) to the Diophantine equation, then to deduce all the subsequent solutions from this initial one: this is represented as a linear function under the form of a chrec as base plus step. The gcd test provides an easy way to prove that the initial solution does not exist, and consequently it proves the *non dependence* property and stops the algorithm before the resolution of the Diophantine equation. The most costly part of this dependence test is effectively the resolution of the Diophantine equation, and more precisely the determination of the initial solution.

Once the conflicting iterations are known, the analyzer is able to abstract this information into a less precise representation: the distance per subscript information. When the conflicting iterations have a same evolution step, the difference of their base gives the distance at which the conflicts occur. When the steps of the conflicting iterations are not equal, the dependence relation is not captured by the distance description.

In a second step, the analyzer refines the dependence relations using the information on several subscripts. The subscript coupling technique allows the disambiguation of more non dependent relations in the case of multidimensional arrays. The classic per loop distances are computed based on the per subscript distance information. When a loop carries two different distances for two different subscripts, the relation is classified to be *non dependent*.

As we have seen, the current implementation of the dependence analyzer is based on the classic dependence tests. For this purpose, only the well formed linear access functions were selected for performing the dependence analysis. Among the rejected access functions are all those whose evolution is dependent on an element that was left under a symbolic form, or contain intervals. For all these cases, the conservative result of the analyzer is the *unknown dependence* relation. In the case of evolution envelopes, it is possible to detect independent data accesses based on the monotonicity properties, as proposed by Peng Wu *et al*. [16].

# 3 Matrix Transformations

## 3.1 Purpose

The reason for using matrix based transformations as opposed to separate loop transformations in conjunction are many. First, one can composite transformations in a much simpler way, which makes it very powerful. While any of the transformations described could be written as a sequence of simple loop transforms, determining the order in which to apply them to achieve the desired transformation is

[1] A Diophantine equation is an equation with integer coefficients.

non-trivial. However, with a matrix transform, one can generate the desired transformation directly. In addition, determining the legality of a given transformation is a simple matter of multiplication. The algorithm used also allows for completion of partial transforms.

### 3.2 Algorithm

The code generation algorithm implemented for GCC is based on Wei Li's Lambda Loop Transformation Toolkit [8]. It uses integer lattices as the model of loop nests and uses non-singular matrices as the model of the transforms. The implemented algorithm supports any loop whose bounds can be expressed as a system of linear expressions, where each linear expression can include loop invariants in the expression. This algorithm is in use by several commercial compilers (Intel, HP), including those known to perform these transformations quite well. This was a consideration in choosing it. Using this algorithm, we can perform any combination of the following transformations, simply by specifying the applicable transformation matrix.

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

```
DO I=1,3
  DO J=1,3
    A(I, 2*J) = J
  END DO
END DO
```

Figure 7: Original loop

$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

```
DO U=1,3
  DO V=2,6,2
    A(U, V) = V/2
  END DO
END DO
```

Figure 8: Loop scaling

The loops produced by applying these transforms to the loop in 7 can be seen in Figures 8, 9, 10, and 11 respectively.

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

```
DO U=1,3
  DO V=1,3
    A(V, 2*U) = U
  END DO
END DO
```

Figure 9: Interchanged loop

$$\begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

```
DO U=1,3
  DO V=U + 1,U + 3
    A(U, 2*(V-U)) = 2*(V-U)
  END DO
END DO
```

Figure 10: Skewed loop

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

```
DO U=1,3
  DO V=-3,-1
    A(U, -2*V) = -V
  END DO
END DO
```

Figure 11: Reversed loop

This set of operations includes every unimodular operation (interchange, reversal, and skewing) plus scaling. The addition of scaling to the applicable transforms means that any non-singular transformation matrix can be applied to a loop, because they can all be reduced to some combination of the above. Scaling is useful in the context of loop tiling, and distributed memory code generation.

Legality testing is performed simply by multiplying the dependence vectors of the loop by the transformation matrix, and verifying that the resulting dependence vectors are lexicographically positive. This will guarantee that the data dependencies are respected in the loop nest generated.

The completion procedures allows completion

of transformation matrices that contain the desired transformation for some portion of the loop, in a way that respects loop dependencies for the entire loop.

Consider the following loop:

```
DO I=4,8
  DO J=3,8
    A(I, J) = A(I-3, J-2) + 1
  END DO
END DO
```

The dependence matrix for this loop is

$$D = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

The outer loop can be made parallel if and only if it does not carry any dependences, i.e., the first entry of every dependence vector is 0. In its current form, this is obviously not true. We can make it parallel if we can find a transformation $T$ such that every entry in the first row of $TD$ is 0. We can easily satisfy that with the partial transform $\begin{pmatrix} 2 & -3 \end{pmatrix}$. However, this is not a complete transformation matrix because it does not specify what to do with the inner loop. The completion algorithm will complete this partial transform in a way that maintains the legality of the transform, i.e., respects dependences.

The full completion procedure is specified in [8]. It works by generating vectors that are independent of the existing row vectors in the partial transformation and within 90 degrees of each dependence vector.

### 3.3 Implementation

The GCC implementation of linear loop transforms is decomposed into several pieces: a matrix math engine, a transformation engine, and converters.

The matrix math engine implements various vector and matrix math routines necessary to perform the transformations (inversion, computation of Hermite form, multiplication, etc).

The transformation engine implements legality testing, rewriting of loop bounds, rewriting of loop bodies, and completion of partial transforms.

To transform a loop using GCC, we first need to convert it to a form usable by the code generation algorithm. There is a simple function which takes a GCC loop structure and produces a loopnest structure usable by the transformation engine. This loopnest structure consists of a system of linear equations representing the bounds of each loop.

Next, we perform legality testing. We have provided a function that takes the loopnest structure and a transformation matrix, and returns true if it is legal. This mainly is useful for transformations that were not produced by the completion algorithm, because that component only produces legal transforms.

Third, The loop bounds of the loopnest structure are rewritten using the aforementioned code generation algorithm.

Finally, we transform the loopnest structure back into real GIMPLE/Tree-SSA code. The subroutine accepts a loopnest structure and rewrites the actual loop nest code to match it. This involves two steps: first the new iteration variables, bounds, and exit condition are generated. Next, the body of the loop is transformed to eliminate uses of the old iteration variables. This procedure is straightforward: given a vector of source iteration variables $S_i$ and a vector of the target iteration variables $S_j$, and the transformation matrix $T$, the function computes the source iteration variables in terms of the target iteration variables using the equation $S_i = T^{-1}S_j$. This calculation is performed for each statement in the loop, and the

old uses are replaced with the new equations.

As a side note, all of these functions work independently of one another. In other words, as long as one supplies the function that rewrites loopnest structures into GCC code, one can reuse the components for other transformations.

### 3.4 Applications

Matrix based loop transforms can be used to improve effectiveness of parallelization and vectorization by removing inner loop dependencies that inhibit their substitution. They can also be used to perform spatial and temporal locality optimizations that optimize cache reuse [7].

These types of optimizations have the potential to significantly improve both application and benchmark scores. Memory locality optimizations are observed to produce speedup factors from 2 to 50 relative to the unmodified algorithm, depending on the application.

As an example of such a speedup, we'll take a well known SPEC® CPU2000 benchmark, SWIM[2].

SWIM spends most of its time in a single loop. By simply interchanging this loop, the performance can be improved sevenfold, as shown in Figure 12.

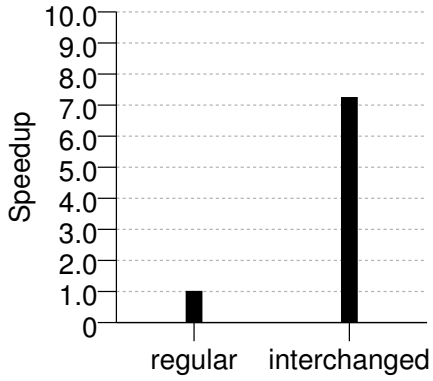



Figure 12: Effect of interchanging loop on SWIM

### 3.5 Future plans

Determination of a good transformation matrix for optimizing temporal and spatial locality is work in progress. There are many potential algorithms from which to choose. The authors are investigating research literature and other compiler implementations in order to choose a good algorithm to implement in GCC. An optimal transform matrix can be calculated in polynomial time for most loops encountered. The matrix transform method can be extended to perform loop alignment transforms, statement-based iteration space transforms, and other useful operations.

[2] `http://www.spec.org/`

## 4 Optimizations

### 4.1 Loop Optimizations

The new data dependence and matrix transformation functionality allows GCC to implement loop nest optimizations that can significantly improve application performance. These optimizations include loop interchange, unroll and jam, loop fusion, loop fission, loop reversal, and loop skewing.

Loop interchange exchanges the order of loops to better match use of loop operands to system characteristics, e.g., improved memory hierarchy access patterns or exposing loop iterations without dependencies to allow vectorization. When the transformation is safe to perform, the optimal ordering of loops depends on the target system. Depending on the intended effect,

interchange can swap the loop with the greatest dependencies to an inner position within the loop nest or to an outer position within the nest. The effectiveness of the optimization is limited by alias and data dependence information.

The Unroll and jam transformation unrolls iterations of an outer loop and then fuses copies of the inner loop to achieve greater value reuse and to hide function unit latency. The optimal unrolling factor is a balance between scheduling and register pressure. The optimization is related to loop interchange and unrolling, so it similarly requires accurate alias and data dependence information.

Loop fusion combines loops to increase computation granularity and create asynchronous parallelism by merging independent computations with the same bounds into a single loop. This allows dependent computations with independent iterations to execute in parallel. Loop fusion requires appropriate alias and data dependence information, and also requires *countable loops*.

```
DO I=1,N
   A(I) = F(B(I))
END DO
Q = ...
DO J=2,N
   C(I) = A(I-1) + Q*B(I)
END DO

      ⇓

Q = ...
A(1)=F(B(1))
DO I=2,N
   A(I) = F(B(I))
   C(I) = A(I-1) + Q*B(I)
END DO
```

Figure 13: Example of Loop Fusion

Loop fission or distribution is the opposite of loop fusion: breaking multiple computations into independent loops. It can enable other optimizations, such as loop interchange and blocking. Another benefit is reduction of register pressure and isolation of vectorizable operations, e.g., exposing the opportunity to invoke a specialized implementation of an operator for vectors or using a vector/SIMD instruction. Vectorization is a balance between vector speedup and memory locality. Again, alias information, data dependence, and *countable loops* are prerequisites.

```
DO I=1,N
   S = B(I) / SQRT(C(I))
   A(I) = LOG(S)*C(I)
END DO

      ⇓

CALL VRSQRT(A,C,N)
DO I=1,N
   A(I) = B(I)*A(I)
END DO
CALL VLOG(A,A,N)
DO I=1,N
   A(I) = A(I)*C(I)
END DO
```

Figure 14: Example of Loop Fission

Loop reversal inverts the direction of iteration and loop skewing rearranges the iteration space to create new dependence patterns. Both optimizations can expose existing parallelism and aid other transformations.

### 4.1.1 Future Plans

After addressing the optimizations that can be implemented with initial loop transformation infrastructure, the functionality will be

expanded to other well-known loop optimizations, such as loop tiling, interleaving, outer unrolling, and support for triangular and trapezoidal access patterns.

The goal function for high-level loop transformations is dependent on the target system. Communicating the system characteristics to the GCC loop optimizer is an ongoing area of investigation.

GCC's high-level loop optimization framework will not implement all, or even most, loop transformations in the first release—it is a work in progress, but an effective starting point from which to grow. Future enhancements to the framework will expand the functionality in two directions: implementing additional optimizations and reducing the restrictions on existing optimizations. The transformations first must be safe to enable for any application with well-defined numerical behavior. The optimizations will be enhanced to recognize more and different types of loops that can benefit from these techniques and improve application performance.

#### 4.1.2 Helping the Compiler

The programmer can assist the compiler in its optimization effort while ensuring that the source code is easy to understand and maintain. This primarily involves simplifying memory analysis, loop structure, and program structure to aid the compiler.

Limiting the use of global variables and pointers allow the compiler to compute more thorough alias information, allowing the safety of transformations to be determined. Replacing pointers by arrays and array indexing is one such example.

Simplified loop structure permits more extensive analysis of the loops and allows easier modification of loops. Some loop transformation optimizations require *perfect loop nesting*, meaning no other code is executed in the containing loop, and most loop optimizations are limited to *countable loops*. A countable loop has a single entry point, a single exit point, and an iteration count that can be determined before the loop begins. A loop index should be a local variable whose address is not taken and avoids any aliasing ambiguity.

Programmers are encouraged to nest loops where possible and restructure loops to avoid branches within, into, or out of loops. Additionally, the programmer manually can perform loop fission to generate separate loops with simple bounds instead of a single loop with complicated bounds and conditionally-executed code within the loop.

### 4.2 Interacting with the Compiler: towards an OpenMP implementation

The OpenMP[3] standard can be seen as an extension to the C, C++, and Fortran programming languages, that provides a syntax to express parallel constructs. Because the OpenMP does not specify the compiler implementation, implementations range from the simple source to source preprocessors such as `OdinMP`[4] and `Omni`[5] to the optimizing compilers like `ORC`[6], that exploit the extra information provided by the programmer for better optimizing loop nests. Based on these implementations of the OpenMP norm, we give some reflections on a possible implementation of OpenMP in GCC.

[3] `http://www.openmp.org/`
[4] `http://odinmp.imit.kth.se/`
[5] `http://phase.hpcc.jp/Omni/`
[6] `http://ipf-orc.sourceforge.net/`

### 4.2.1 Source to Source Implementations

The source to source implementations of OpenMP include a parser that constructs an abstract syntax tree (AST) of the program, then a pretty printer that generates a source code from the AST. The AST is rewritten using the information contained in the OpenMP directives. The transformations involved in the rewriting of the AST are principally insertions of calls to a thread library, the creation of new functions, and restructuring of loop bounds and steps. The main benefit of this approach is that it requires a reduced compiler infrastructure for translating the OpenMP directives.

For implementing this source to source approach in GCC, two main components have to be designed:

- *a directive parser*, that is an extension of the parser for generating AST nodes for each directive, and
- *a directive rewriter*, that transforms the code in function of the directives.

In order to keep the code generation part generic for all the front-ends, a specific `OMP_EXPR` node could contain the information about the directives, until reaching the GENERIC, or the GIMPLE levels, the GIMPLE level having the benefit of being simpler, and more flexible for restructuring the code.

In the source to source model, the rewrite of the directives directly generates calls to a threading library, and the rest of the compiler does not have to handle the `OMP_EXPR` nodes. This kind of transformation tends to obfuscate the code by inserting calls to functions in place of the loop bodies, rendering the loop optimizations ineffective. In order to avoid this drawback we have to make the optimizers aware about the parallel constructs used by the programmer.

### 4.2.2 An Optimizing Compiler Approach

In the C, C++, and Fortran programming languages, the parallelism is expressed mainly using calls to libraries that implement threading or message passing interfaces. The compiler is not involved in the process of optimizing parallel constructs because the parallel structures are masked by the calls to the parallel library. In other programming languages, such as Ada and Java, parallel constructs are part of the language specification, and allow the compiler to manage the parallel behavior of the program. OpenMP directives fill a missing part of the C, C++, and Fortran programming languages with respect to the interaction of the programmer with the compiler for concurrent programming. It is in this extent that the OpenMP norm is interesting from the point of view of an optimizing compiler.

In order to allow the optimizers to deal with the parallel constructs in a generic way, the compiler has to provide a set of primitives for the parallel constructs. For the moment, the GENERIC level does not contain parallel primitives, and consequently the front-end languages have to lower their parallel constructs before generating GENERIC trees. In this respect, the OpenMP directives should not be different than other languages parallel constructs, and should not have a specific `OMP_EXPR` that allow these constructs to be propagated to the GIMPLE level for their expansion as described in section 4.2.1. The support of OpenMP in this context is to *genericize* the directives to their equivalent constructs in GENERIC and let the optimizers work on this representation. Using this approach would allow the compiler to choose the right degree of parallelism based on a description of the underlying architecture.

In the previous discussions on the GCC mailing lists, there were some good questions on whether we want to support the OpenMP standard in GCC, but rather than asking again this question, the authors would like to ask another question: do we want the generic optimizer to deal with concurrency aspects, and the programmer to be able to interact with the optimizer on parallel constructs?

## 5 Conclusions

The Loop Nest Optimizer provides an effective and modular framework for implementing high-level loop optimizations in GCC. Initial loop optimizations are built on a new loop data dependence analysis and matrix transformation engine infrastructure.

This work allows GCC to expand into a number of areas of optimization for high performance computing. The loop optimizations improve performance directly and provide a base on which to develop auto-vectorization and auto-parallelization facilities.

## 6 Acknowledgements

This work has been supported by IBM Research and the IBM Linux Technology Center. We wish to thank Zdenek Dvorak *SUSE* and Devang Patel and Andre Pinski *Apple Computer* for their collaboration, Bob Blainey *IBM Toronto* for insightful discussions, and the developers of Tree-SSA who created the opportunity for new enhancements to GCC.

Sebastian Pop would like to thank *École des mines de Paris (ENSMP)* who supports his PhD thesis, Georges-André Silber *(CRI-ENSMP)*, Albert Cohen *(Alchemy-INRIA)*, and Philippe Clauss *(ICPS-LSIIT)* to whom he is grateful for their guidance and collaboration.

## References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence Based Approach*. Morgan Kaufman, San Francisco, 2002.

[3] O. Bachmann, P. S. Wang, and E. V. Zima. Chains of recurrences-a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249. ACM Press, 1994.

[4] U. Banerjee. *Loop transformations for restructuring compilers*. Kluwer Academic Publishers, Boston, 1994.

[5] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

[6] V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional chains of recurrences. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation*, pages 199–206. ACM Press, 1998.

[7] W. Li. Compiler optimizations for cache locality and coherence. Technical Report TR504, Department of Computer Science, University of Rochester, 1994.

[8] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *1992 Workshop*

*on Languages and Compilers for Parallel Computing*, number 757, pages 391–405, New Haven, Conn., 1992. Berlin: Springer Verlag.

[9] J. Merrill. Generic and gimple: A new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers' Summit*, pages 171–179, 2003.

[10] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.

[11] D. Novillo. Tree ssa: A new otimization infrastructure for gcc. In *Proceedings of the 2003 GCC Developers' Summit*, pages 181–193, 2003.

[12] Y. N. Srikant and P. Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. CRC Press, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA, 2002.

[13] R. van Engelen. Symbolic evaluation of chains of recurrences for loop optimization. Technical Report TR-000102, Computer Science Department, Florida State University, 2000.

[14] R. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction, ETAPS 2001, LNCS 2027*, pages 118–132, 2001.

[15] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, Reading, Mass., 1996.

[16] P. Wu, A. Cohen, D. Padua, and J. Hoeflinger. Monotonic evolution: an alternative to induction variable substitution for dependence testing. In *Proceedings of the 15th ACM International Conference on Supercomputing (ICS-01)*, pages 78–91, New York, June 17–21 2001. ACM Press.

[17] E. V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 international symposium on Symbolic and algebraic computation*, page 345. ACM Press, 2001.

# Swing Modulo Scheduling for GCC

*Mostafa Hagog*
IBM Research Lab in Haifa
mustafa@il.ibm.com

*Ayal Zaks*
IBM Research Lab in Haifa
zaks@il.ibm.com

## Abstract

Software pipelining is a technique that improves the scheduling of instructions in loops by overlapping instructions from different iterations. Modulo scheduling is an approach for constructing software pipelines that focuses on minimizing the cycle count of the loops and thereby optimize performance. In this paper we describe our implementation of Swing Modulo Scheduling in GCC, which is a Modulo Scheduling technique that also focuses on reducing register pressure. Several key issues are discussed, including the use and adaptation of GCC's machine-model infrastructure for scheduling (DFA) and data-dependence graph construction. We also present directions for future enhancements.

## 1 Introduction

Software pipelining is an instruction scheduling technique that exploits instruction level parallelism found in loops by overlapping successive iterations of the loop and executing them in parallel. The key idea is to find a pattern of operations (named the kernel code) that when iterated repeatedly, produces the effect that an iteration is initiated before previous ones have completed [3]. Modulo scheduling is a technique for implementing software pipelining. It does so by first estimating a lower bound on the number of cycles it takes to execute the loop. This number is called the *Initiation Interval* — II, and the bound is called a *Minimum II* — MII (see example in Figure 1). Then it tries to place the instructions of the loop in II cycles, while taking into account the machine resource constraints and the instruction dependencies. In case the loop couldn't be scheduled in II cycles it tries with larger II until it succeeds.

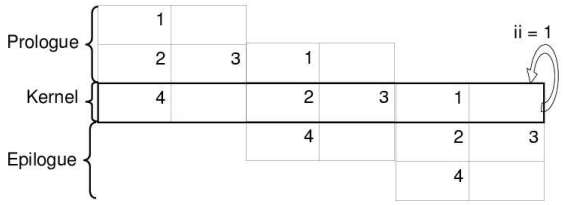



Figure 1: Example software pipelined loop of 4 instructions and the resulting kernel, prologue and epilogue.

Swing Modulo Scheduling (SMS) is a heuristic approach that aims to reduce register pressure [2]. It does so by first ordering the instructions in an alternating up-and-down order according to the data dependencies, hence its name (see section 2.2). Then the scheduling algorithm (section 2.3) traverses the nodes in the given order, trying to schedule dependent instructions as close as possible and thus to

shorten live ranges of registers.

## 2 Implementation in GCC

Swing Modulo Scheduling (SMS) [2, 3] was implemented as a new pass in GCC that immediately precedes the first scheduling pass. An alternative is to perform SMS after register allocation, but that would require register renaming and spilling in order to remove anti-dependencies and free additional registers for the loop. The new pass traverses the current function and performs SMS on loops. It generates a new schedule for the instructions of the loop according to the SMS algorithm, which is "near optimal" in utilizing hardware resources and register pressure. It also handles long live ranges and generates prologue and epilogue code as we describe in this section.

The loops handled by SMS obey the following constraints: (1) The number of iterations of the loop is known before entering the loop (i.e. is loop-invariant). This is required because when we exit the kernel, the last few iterations are in-flight and need to be completed in the epilogue. Therefore we must exit the kernel a few iterations before the last (or support speculative partial execution of a few iterations past the last). (2) A single basic block loop. For architectures that support predicated instructions, multiple basic block loops could be supported.

For each candidate loop the modulo scheduler builds a data-dependence graph (DDG), whose nodes represent the instructions and edges represent intra- and inter-loop dependences. The modulo scheduler then performs the following steps when handling a loop:

1. Calculate a MII.
2. Determine a node ordering.
3. Schedule the kernel.
4. Perform modulo variable expansion.
5. Generate prolog and epilog code.
6. Generate a loop precondition if required.

After a loop is successfully modulo-sceduled it is marked to prevent subsequent rescheduling by the standard instruction scheduling passes. Only the kernel is marked; the prolog and epilog are subject to subsequent scheduling.

Subsection 3.1 describes the DDG. In the remainder of this section we elaborate each of the above steps.

### 2.1 Calculating a MII

The minimum initiation interval ("MII") is a lower-bound on the number of cycles required by any feasible schedule of the kernel of a loop. A schedule is feasible if it meets all dependence constraints with their associated latencies, and avoids all potential resource conflicts. Two separate bounds are usually computed—one based on recurrence dependence cycles ("recMII") and the other based on the resources available in the machine and the resources required by each instruction ("resMII") [6]:

$$\text{MII} = \max\{\text{recMII}, \text{resMII}\}.$$

In general, if the computed MII is not an integer, loop unrolling can be applied to possibly improve the scheduling of the loop. The purpose of computing MII is to avoid trying II's that are too small, thereby speeding-up the modulo scheduling process. It is not a correctness issue, and being a lower bound does not affect the resulting schedule.

The "recMII" lower bound is defined as the maximum, taken over all cycles $C$ in the dependence graph, of the sum of latencies along

$C$ divided by the sum of distances along $C$:

$$\text{recMII} = \max_{C \in DDG} \frac{\sum_{e \in C} \text{latency}(e)}{\sum_{e \in C} \text{distance}(e)}.$$

Computing the maximum above can be done in $\Theta(N^3)$ (worst and best) time, where $N$ is the number of nodes in the dependence graph [6]. We chose to implement a less accurate yet generally more efficient computation of a dependence-recurrence based lower bound, focusing on simple cycles $S$ that contain a single back-arc $b(S)$ (more complicated cycles are ignored, resulting in a possibly smaller lower bound):

$$\text{recMII'} = \max_{S \in DDG} \frac{\sum_{e \in S} \text{latency}(e)}{\text{distance}(b(S))}.$$

(Note that for such simple cycles $S$, $\text{distance}(e) = 0$ for all edges $e \in S$ except $b(S)$.) This maximum is computed by finding for each back-arc $b(S) = (h, t)$ the longest path (in terms of total latency) from $t$ to $h$, excluding back-arcs (i.e. in a DAG). This scheme should be more efficient because the number of back-arcs is anticipated to be relatively small, and is expected to suffice because we anticipate most recurrence cycles to be simple.

The "resMII" is currently computed by considering issue constraints only: the total number of instructions is divided by the ISSUE_RATE parameter. This bound should be improved by considering additional resources utilized by the instructions.

In addition to the MII lower-bound, we also compute an upper-bound on the II, called *MaxII*. This upper-bound is used to limit the search for an II to effective values only, and also to reduce compile-time. We set MaxII= $\sum_{e \in DDG} \text{latency}(e)$ (the standard instruction scheduler should achieve such an II), and provide a factor for tuning it (see Section 5).

### 2.2 Determining a Node Ordering

The goal of the "swinging" order is to schedule an instruction after scheduling its predecessor or successor instructions and as close to them as possible in order to shorten live ranges and thereby reduce register pressure. Alternative ordering heuristics could be supported in the future. (See figure 7 [1] for the swing ordering algorithm).

The node ordering algorithm takes as input a data dependence graph, and produces as output a sorted list of the nodes of the graph, specifying the order in which to list-schedule the instructions. The algorithm works in two steps. First, we construct a partial order of the nodes by partitioning the DDG into subsets $S_1, S2, \ldots$ (each subset will later be ordered internally) as follows:

1. Find the SCC (Strongly Connected Component)/Recurrence of the data-dependence graph having the largest recMII—this is the first set of nodes $S_1$.
2. Find the SCC with the next largest recMII, put its nodes into the next set $S_2$.
3. Find all nodes that are on directed paths from any previous set to the next set $S_2$ and add them to the next set $S_2$.
4. If there are additional SCCs in the dependence graph goto step 2. If there are no additional SCCs, create a new (last) set of all the remaining nodes.

The second step orders the nodes within each $S_i$ set using a directed-acyclic subgraph of the DDG obtained by disregarding back-arcs of $S_i$:

1. Calculate several timing bounds and properties for each node in the dependence

graph (earliest/latest times for scheduling according to predecessors/successors—see subsection 4.1 [1]).

2. Calculate the order in which the instructions will be processed by the scheduling algorithm using the above bounds.

### 2.3 Scheduling the Kernel

The nodes are scheduled for the kernel of the loop according to the precomputed order. Figure 2 shows the pseudo code of the scheduling algorithm, and works as follows. For each node we calculate a scheduling window—a range of cycles in which we can schedule the node according to already scheduled nodes. Previously scheduled predecessors (PSP) increase the lower bound of the scheduling window, while previously scheduled successors (PSS) decrease the upper bound of the scheduling window. The cycles within the scheduling window are not bounded a-priori, and can be positive or negative. The scheduling window itself contains a range of at-most II cycles. After computing the scheduling window, we try to schedule the node at some cycle within the window, while avoiding resource conflicts. If we succeed we mark the node and its (absolute) schedule time. If we could not schedule the given node within the scheduling window we increment II, and start over again. If II reaches an upper bound we quit, and leave the loop without transforming it.

If we succeed in scheduling all nodes in II cycles, the register pressure should be checked to determine if registers will be spilled (due to overly aggressive overlap of instructions), and if so increment II and start over again. This step has not been implemented yet.

During the process of scheduling the kernel we maintain a *partial schedule*, that holds the scheduled instructions in II *rows*, as follows: when scheduling an instruction in cycle $T$ (inside its scheduling window), it is inserted into row $(T \bmod II)$ of the partial schedule. Once all instructions are scheduled successfully, the partial schedule supplies the order of instructions in the kernel.

A modulo scheduler (targeting e.g. a superscalar machine) has to consider the order of instructions within a row, when dealing with the start and end cycles of the scheduling window. When calculating the start cycle for instruction $i$, one or more predecessor instructions $p$ will have a tight bound $SchedTime_p + Latency_{p,i} - distance_{p,i} \times ii =$ start (see Figure 2). If $p$ was itself scheduled in the start row, $i$ has to appear after $i$ in order to comply with the direction of the dependence. An analogous argument holds for successor instructions that have a tight bound on the end cycle. Notice that there are no restrictions on rows strictly between start and end. In most cases (e.g. targets with hardware interlocks) the scheduler is allowed to relax such tight bounds that involve positive latencies, and the above restriction can be limited to zero latency dependences only.

### 2.4 Modulo Variable Expansion

After all instructions have been scheduled in the kernel, some values defined in one iteration and used in some future iteration must be stored in order not to be overwritten. This happens when a life range exceeds II cycles—the defining instruction will execute more than once before the using instruction accesses the value. This problem can be solved using modulo variable expansion, which can be implemented by generating register copy instructions as follows (certain platforms provide such support in hardware, using rotating-register capabilities):

1. Calculate the number of copies needed for a given register defined at cycle $T$_def and

```
ii = MII; bump_ii = true;
ps = create_ps (ii, G, DFA_HISTORY);
while (bump_ii && ii < maxii){
 bump_ii = false;sched_nodes = φ;
           step = 1;
 for (i=0, u=order[i];
      i<|G|; u=order[++i]) do {
  /*Compute sched window for u.*/
  PSP = u_preds ∩ sched_nodes;
  PSS = u_succs ∩ sched_nodes;
  if (PSP ≠ φ ∧ PSS = φ){
   start=max(SchedTime_v + Latency_v,u
             −distance_v,u × ii)∀v ∈ PSP
   end = start + ii;
  }
  else if (PSP = φ ∧ PSS ≠ φ){
   start=min(SchedTime_v − Latency_u,v
             +distance_u,v × ii)∀v ∈ PSS
    end = start - ii; step = -1;
  }
  else if (PSP ≠ φ ∧ PSS ≠ φ){
   estart=max(SchedTime_v + Latency_v,u
             −distance_v,u × ii)∀v ∈ PSP
   lstart=min(SchedTime_v − Latency_u,v
             +distance_u,v × ii)∀v ∈ PSS
   start = max(start, estart);
   end = min(estart+ii, lstart+1);
  }
  else /* PSP = φ ∧ PSS = φ */
   start = ASAP_u; end = start + ii;

  /* Try scheduling u in window.  */
  for (c = start; c != end; c += step)
   if (ps_add_node (ps, u, c)){
    SchedTime_u = c;
    sched_nodes = sched_nodes ∪ {u};
    success = 1;
   }
  if (!success){
   ii++; bump_ii = true;
   reset_partial_schedule (ps, ii);
  }
 }/* Continue with next node.  */
 if (!bump_ii
     &&check_register_pressure(ps)){
  ii++; bump_ii = true;
  reset_partial_schedule (ps, ii);
 }
}/* While bump_ii.  */
where:  ASAP_u is the earliest time u
        could be scheduled in[2]
```

Figure 2: Algorithm for Scheduling the Kernel

used at cycle $T$_use, according to the following equation:

$$\left\lfloor \frac{T_{\text{use}} - T_{\text{def}}}{\text{II}} \right\rfloor + \text{adjustment} \quad (1)$$

where "adjustment" = -1 if the use appears before the def on the same row in the partial schedule, and zero otherwise. The total number of copies needed for a given register def is given by the last use.

2. Generate the register copy instructions needed, in reverse order preceeding the def:

$$r_n \leftarrow r_{n-1}; r_{n-1} \leftarrow r_{n-2}; \ldots r_1 \leftarrow r_{def}$$

and attach each use to the appropriate $r_m$ copy.

### 2.5 Generating Prolog and Epilog

The kernel of a modulo-scheduled loop contains instances of instructions from different iterations. Thus a prolog and an epilog (unless all moves are speculative) are needed to keep the code correct.

When generating the prolog and epilog, special care should be taken if the loop bound is not known. One possibility is to add an exit branch out of each iteration of the prolog, targeting a different epilog. This is complicated and increases the code size (see [1]. Another approach is to keep an original copy of the loop to be executed if the loop-count is too small to reach the kernel, and otherwise execute a branch-less prolog followed by the kernel and a single epilog. We implemented the latter because it is simpler and has smaller impact on code size.

# 3 Infrastructure Requirements for Implementing SMS in GCC

The modulo scheduler, being a scheduling optimization, needs to work with a low level representation close to the final machine code. In GCC that is RTL. The SMS algorithm requires several building blocks from the RTL representation:

1. Identifying and representing RTL level loops—we use the CFG representation.
2. Building data dependence graph (for loops) with loop carried dependencies—we implemented a Data Dependence Graph (DDG).
3. An ordered linked list of instructions (exists in the RTL). Union, intersection, and subtraction operations on sets of instructions—we use the `sbitmap` representation.
4. Machine resource model support, mainly for checking if a given instruction will cause resource conflicts if scheduled at a given cycle/slot of a partial schedule.
5. Instruction latency model—we use the `insn_cost` function.

We now describe the DDG and Machine model support.

### 3.1 Data Dependence Graph (DDG) Generation

The current representation of data dependencies in GCC does not meet the requirements for implementing modulo scheduling; it lacks inter-loop dependencies and it is not easy to use. We decided to implement a DDG, which provides additional capabilities (i.e. loop carried dependencies) and modulo-scheduling oriented API.

The data dependence graph is built in several steps. First, we construct the intra-loop dependencies using the standard LOG_LINKS/INSN_DEPEND structures by calling the sched_analyze function of haifa-sched.c module; a dependence arc with distance zero is then added to the DDG for each INSN_DEPEND link. We then calculate inter-loop register dependencies of distance 1 using the df.c module as follows:

1. The latency between two nodes is calculated using the `insn_cost` function of the scheduler.
2. For each downwards reaching definition, if there is an upwards reaching use of the same register (this information is supplied by the df analysis) a TRUE dependence arc is added between the def and the use.
3. For each downwards reaching definition find its first definition and connect them by an OUTPUT dependence, if they are distinct. Avoid creating self OUTPUT dependence arcs.
4. For each downwards reaching use find its first def, if this is not the def feeding it (intra-loop) add an ANTI inter-loop dependence. Avoid creating inter-loop ANTI register dependences—modulo variable expansion will handle such cases (see 2.4). FLOW dependence exists in the opposite direction;

Finally, we calculate the inter-loop memory dependencies. Currently, we are over conservative due to limitation of alias analysis. This issue is expected to be addressed in the future. The current implementation adds the following dependence arcs, all with distance 1 (unless the nodes are already connected with a dependence arc of distance 0):

1. For every two memory writes add an inter-loop OUTPUT dependence.

2. For every memory write followed by a memory read (across the back-arc) add a TRUE memory dependence.

3. For every memory read followed by a memory write across the back-arc add an ANTI memory dependence.

The following general functionality is provided by the DDG to support the node-ordering algorithm of SMS:

- Identify cycles (strongly connected components) in the data dependence graph, and sort them according to their recMII.

- Find the set of all predecessor/successor nodes for a given set of nodes in the data dependence graph.

- Find all nodes that lie on some directed path between two strongly connected subgraphs.

### 3.2 Machine Resource Model Support

During the process of modulo scheduling, we need to check if a given instruction will cause resource conflicts if scheduled at a given cycle/slot of a partial schedule. The DFA-based resource model in GCC [4] works by checking a sequence of instructions, in order. This approach is suitable for cycle scheduling algorithms, in which instructions are always appended at end of the current schedule. In order for SMS to use this linear approach, we generate a trace of instructions cycle by cycle, centered at the candidate instruction, and feeding it to the DFA [5]. Figure 3 describes the algorithm that checks if there are conflicts in a given partial schedule around a given cycle. Several functions are made available to manipulate the partial schedule, the most important one is *ps_add_node_check_conflicts* described in Figure 4; it updates the partial schedule (tentatively) with a new instruction at a given cycle, and feeds the new partial schedule to the DFA. If it succeeds it updates the partial schedule and returns success, if not it resets the partial schedule and returns failure. The major drawback of the above mechanism is the increase in compile time; there are plans to address this concern in the future.

```
/* Checks if PS has resource
   conflicts according to DFA,
   from FROM cycle to TO cycle.  */
ps_has_conflicts (ps, from, to){
  state_reset (state);
  for (c = from; c <= to; c++) {
    /* Holds the remaining issue
       slots in the current row.  */
    issue_more = issue_rate;
    /* Walk DFA through CYCLE C.  */
    for (I = ps->rows[c % ps->ii)];
         I; I = I->next) {
      /* Check if there is room for the
         current insn I.*/
      if (! issue_more
          || state_dead_lock_p (state))
        return true;
      /* Check conflicts in DFA.*/
      if (state_transition (state, I))
        return true;
      if (DFA.variable_issue)
        issue_more=DFAissue(state, I);
      else issue_more--;
    }
    advance_one_cycle ();
  }
  return false;
}
```

Figure 3: Feeding a partial schedule to DFA.

## 4 Current status and future enhancements

An example of a loop and its generated code, when compiled with gcc and SMS enabled (`-fmodulo-sched`) is shown in Figure 5. The kernel combines the fmadds of the current iteration with the two lfsx's of the next iteration. As a result, the two lfsx's appear in

```
/* Checks if a given node causes
   resource conflicts when added to
   PS at cycle C.  If not add it.  */
ps_add_node_check_conflicts (ps, n, c)
{
  ps_n = add_node_to_ps (ps, n, c);
  from = c - ps->history;
  to = c + ps->history
  has_conflicts
    = ps_has_conflicts(ps, from, to);

  /* Try different slots in row.  */
  while (has_conflicts)
    if (!ps_insn_advance_column(ps,
                                ps_n))
      break;
    else has_conflicts
      = ps_has_conflicts(ps,
                         from, to);

  if (! has_conflicts)
    return ps_n;
  remove_node_from_ps(ps, ps_n);
  return NULL;
}
```

Figure 4: Add new node to partial schedule

the prolog and the fmadds appears in the epilog. This could help hide the latencies of the loads. The count of the loop is decreased to 99, and no register-copies are needed because every life range is smaller than II,

Following are milestones for implementing SMS in GCC.

First stage (Approved for mainline)

1. Implement the required infrastructure: DDG (section 3.1), special interface with DFA (section 3.2).
2. Implement the SMS scheduling algorithm as described in [3, 2].
3. Support only distance 1 and register carried dependences (including accumulation).

```
float dot_product (float *a,
                   float *b){
  int i; float c=0;
  for (i=0; i < 100; i++)
     c += a[i]*b[i];
  return c;
}
            (a)
L5:
        slwi r0,r2,2
        addi r2,r2,1
        lfsx f13,r4,r0
        lfsx f0,r3,r0
        fmadds f1,f0,f13,f1
        bdnz L5
        blr
            (b)

Prolog: addi r2,r2,1
        lfsx f0,r3,r0
        lfsx f13,r4,r0
        li r0,99
        mtctr r0
L5:
        slwi r0,r2,2
        addi r2,r2,1
        fmadds f1,f0,f13,f1
        lfsx f13,r4,r0
        lfsx f0,r3,r0
        bdnz L5
Epilog: fmadds f1,f0,f13,f1
        blr
           (c)
```

Figure 5: (a) An example C loop, (b) Assembly code without SMS, (c) Assembly code with SMS (`-fmodulo-sched`), on a PowerPC G5.

4. Support for live ranges that exceed II cycles by register copies.
5. Support unknown loop bound using loop preconditioning.
6. Prolog and epilog code generation as described in Section 2.5.
7. Preliminary register pressure measurements—gathering statistics.

Second stage

1. Support dependences with distances greater than 1.
2. Improve the interface to DFA to decrease compile time.
3. Support for live ranges that exceed II cycles by unroll & rename.
4. Improve register pressure heuristics/measurements.
5. Improve computation of resMII and possibly recMII lower bounds.
6. Unroll the loop if tight MII bound is a fraction.

Future enhancements [tentative list]

1. Consider changes to DFA to make SMS less time consuming when checking resource conflicts.
2. Consider spilling during SMS if register pressure rises too much.
3. Support speculative moves.
4. Support predicated instructions and if-conversion.
5. Support for live ranges that exceed II cycles by rotating registers (for appropriate architectures).

## 5 Compilation Flags for Tuning

We added the following four options for tuning SMS:

sms-max-ii-factor. This parameter is used to tune the SMS_MAX_II threshold, which affects the upper bound for II (maxII). The default value for this parameter is 100. Decreasing this value will allow modulo scheduling to transform only the loops where a relatively small II can be achieved.

sms-dfa-history. The number of cycles considered when checking conflicts using the DFA interface. The default value is 0, which means that only potential conflicts between instructions scheduled in the same cycle are considered. Increasing this value may result in higher II (possibly less loops will be modulo scheduled), longer compile-time, but potentially less hazards.

sms-loop-average-count-threshold. A threshold on the average loop count considered by the modulo scheduler; defaults to 0. Increasing this value will result in applying modulo scheduling to additional loops, that iterate on average fewer times.

max-sms-loop-number. Maximum number of loops to perform modulo scheduling, mainly for debugging (search for first faulty loop). The default is -1 which means to consider all relevant loops.

## 6 Conclusions

In this paper we described our implementation of Swing Modulo Scheduling in GCC. An example of its effects is given in Section 4. The major challanges involved using the DFA-based machine model of GCC, and building a data-dependence graph for loops including inter-loop dependences. The current straightforward usage of the machine model is time-consuming and should be improved, which involves changes to the machine model. The inter-loop dependencies of the DDG should be built more accurately, in-order to allow more aggressive movements by the modulo scheduler. The DDG is general and can be used by other optimizations as well. Additional opportunities for improving and tuning the modulo scheduler exist, including register pressure and loop unrolling considerations.

## 7 Acknowledgments

One of the key issues in implementing SMS in GCC is the ability to use DFA resource modeling. We would like to thank Vladimir Makarov for helping us understand the DFA interface, the fruitful discussions that led to extending the DFA interface, and his assistance in discussing and reviewing the implementation of SMS in GCC.

## References

[1] Josep Llosa Stefan M. Freudenberger. Reduced code size modulo scheduling in the absence of hardware support. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO-35)*, November 2002.

[2] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: A lifetime sensitive approach. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96)*, pages 80–87, Boston, Massachusetts, USA, October 1996.

[3] J. Llosa, A. Gonzalez, E. Ayguade, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. on Comps.*, 50:3, March 2001.

[4] Vladimir N. Makarov. The finite state automaton based pipeline hazard recognizer and instruction scheduler in gcc. In *Proceedings of the GCC Developers Summit*, May 2003.

[5] Vladimir N. Makarov. Personal communication. 2004.

[6] B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, November 1994.

# The GCC call graph module

**a framework for inter-procedural optimization**

*Jan Hubička*
SUSE ČR
jh@suse.cz

## Abstract

The implementation of call graph based optimizations in GCC required several design changes to the interfaces in between front-ends and back-end parts of the compiler. We describe in detail the new interfaces, optimizations we implemented (in-lining and basic inter-procedural propagation) and the call graph datastructure itself. We compare memory consumption, compilation time and code quality of function at a time and unit at a time compilation scheme. We also outline future plans for the more advanced inter-procedural optimizations and whole program optimization.

## 1 Introduction and motivation

The implementation of function inlining in gcc used to be a major source of dissatisfaction among users of the compiler. Even though inlining had been redesigned from scratch in GCC 3.0, both inliners had serious problems.

The old inlining implementation (based on the low-level RTL intermediate language) could not remove several ugly artefacts in the code, such as in-memory structures used to pass arguments. It also consumed unnecessarily large amount of memory to store function bodies in RTL form. Memory consumption was further increased by storing functions after inlining of callees instead of before.

The new tree-based implementation of inlining in GCC 3.x solved all of these problems but unfortunately brought several new issues. For very complex C++ programs, the new inlining decision heuristics inlined too many functions causing extreme memory consumption, large compile times, and impractically bloated applications. On the other hand the default inline limits were way too low for C programs such as the Linux kernel, causing many functions to not be inlined at all despite the programmer having manually marked them inline. As a result compiler became almost unusable for some C++ programmers working on template heavy code (such as POOMA library) and Linux kernel developers adopted the paradigm of using the `always_inline` attribute to override the default inlining heuristics everywhere.

In addition to these problems, GCC traditionally was unable to perform “backward inlining” (inline functions used before defined), causing noticeable loss in some benchmarks such as SPEC2000 when compared to other compilers.

It seemed impossible to tune the inlining heuristics using the available set of parameters, and thus we started to look for a more involved solution. While looking at the problem from a high level, it seems to be really easy to sim-

ply inline all "small" functions as long as doing so does not cause "extreme bloat." Defining which functions are "small" can be done easily by limiting number of instructions in it's body, while defining code bloat can be done with two parameters: first one limits growth of single function body (since compiler algorithms are generally not linear, and for really large functions, produce both poor code and long compilation times) and the second one limits growth of overall binary size. Unfortunately without whole program optimization (still mostly out of reach of the current GCC framework), it is impossible to realize the last argument, but one still can limit the overall growth of single compilation unit and get similar results.

Because implementation of such a global parameters for function inlining was very difficult with the original organization of the compiler we took a more difficult path and first developed an infrastructure to assist inter-procedural optimization, to be used later when focusing on the inlining issues.

In this paper we describe the infrastructure and the new optimizations implemented while working on this project. The rest of this paper is organized as follows. In Section 2 we briefly describe some problems we had to deal with and solutions we chose for them; in Section 3 we describe the basic data structures we use; in Section 4 we describe the interface to the front-end; in Section 5, the implementation of inlining; and Section 6 contains some experimental evaluation of the new algorithms.

## 2 Overall design and the implementation challenges

GCC compiled the majority of functions immediately after parsing their bodies (only a few functions, such as static inline functions, were special-cased and deferred until it was obvious that the out of line copy is needed) making implementation of inter-procedural optimizations impossible. It was necessary to reorganize the compilation process in a way so all functions are parsed first, then analyzed and compiled last. We will refer to this scheme of compilation as *unit-at-a-time* as opposed to function-at-a-time used by GCC originally.

The main problem that arised was that the original GCC design made it very difficult to change the compilation order. The back-end has been organized as a library that allowed the front-end to compile a specified function. Each of the front-ends implemented its own (in some cases remarkably complex) logic on compiling and/or deferring a function and expected the compilation to happen immediately after passing it to back-end (for instance, the C++ front-end looked back into the symbols actually output to the assembly file to figure out which functions were referenced and had be compiled).

Instead of implementing unit-at-a-time logic into each individual front-end, it seemed easier to reorganize the interface in between the front-ends and back-ends to allow implementation of the generic compilation driver taking care of all the decisions. Since reorganizing all the front-ends at once was a difficult task, the new API has been made optional, and we first implemented unit-at-a-time for the C front-end only and later started work on reorganizing the others.

At the moment, only the C, Objective C, C++, Java, and F90 front-ends have been updated to the new API, and with exception of C, each conversion was a nontrivial task. C++ needed to look back into assembly files to discover what templates needs to be instantiated; Objective C gathered information about method API during compiling the function body, and later producing functions using that informa-

tion; and F90 and Java use trees slightly different from the C++ family, and broke some expectations in the new code.

Switching to unit-at-a-time by default just seemed too radical. The main concerns that were pointed out in the discussion about the change were about peak memory usage growth: in function-at-a-time mode, the function bodies can be released early once the processing of given function finished, while unit-at-a-time mode needs to store into memory all functions at once. If the amount of memory occupied by the function bodies gets too large, it may result in slow down of the compilation.

As a result of this discussion, we decided to allow coexistence of both schemes and added the command line option `-funit-at-a-time` to choose particular one. To date, optimization levels `-O0` and `-O1` by default use function-at-a-time compilation, while `-O2` and `-O3` use unit-at-a-time. Once the front-end is converted into the new API, both supported compilation schemes (unit-at-a-time and function-at-a-time) appear almost identical to the front-end, and all the logic is hidden in the new compilation driver implemented in `cgraphunit.c`.

The compilation process is now organized as follows:

1. Parsing phase: This step is fully controlled by the front-end. It is up to the front-end to decide when a given function is "finalized" and pass it to the compilation driver. After that point the front-end is not allowed to make any modifications on the function body or declaration, and it is fully up to the compilation driver to decide when (and if) the function will be compiled.

   It is probably important to note that there is one exception the rule disallowing any changes to the functions passed to the back-end. The C front-end, GCC allows the function to be first defined as `extern inline` and later be re-defined with a completely different body as an ordinary function. In this special case, we allow the finalization to be called twice; we simply remove all traces of the old body from the data structures and mark the function as uninlinable, then, when this situation is detected.

   At this stage, early analysis of finalized functions is done as well. Certain warnings (such as about unused function parameters) are output here, since it is the last time we'll see unneeded functions. It is also decided whether the function is an "entry point"—i.e., whether it is reachable from unknown code by some way (such as via external linkage).

   The difference between function-at-a-time and unit-at-a-time mode also lies in the finalization code. In unit-at-a-time mode, the function is just stored into the data-structure and left for later analysis, while in function-at-a-time mode all functions are fully analyzed immediately, the control flow graph is incrementally built, and most functions are compiled—the only exceptions being static inline, extern inline, comdat,[1] and nested functions. These are just stored into the call-graph and compiled only when they turn out to be necessary (i.e., when symbol is output into the assembly file).

   A similar mechanism is implemented for file-scope variables. In unit-at-a-time, all variables are stored into variable pool data-structure, while in function-at-a-time mode, all variables are output to the assembly file immediately.

   In function-at-a-time mode compilation

[1] functions that may appear in multiple units and are linked into a single function.

terminates once parsing is finished, while in unit-at-a-time it goes into following stages:

2. Analysis phase: The call-graph is built and local optimization information is gathered at this stage. To reduce the amount of work done, the call-graph is built incrementally and only functions reachable from the entry points are analyzed. Since we do not handle any dependency edges on data-structures, the reachable data-structures are immediately output into the assembly file and further functions/data structures referenced by them are added into the work lists via a callback from back-end function, outputting a symbol reference into the assembly file.

   The local analysis used to drive inter-procedural optimizations is also supposed to happen here. At the moment, the size of function body is estimated for later use in inlining.

3. Optimization phase: Several optimizations are performed on the call-graph itself in sequence. At the moment following optimizations are done:

   (a) Reclaiming of memory occupied by the unused (i.e., unanalyzed) functions and data-structures.

   (b) Local function discovery: A *local function* is a function that is not an entry point and whose address has never been taken. We mark these functions by special flag, since it is possible to perform optimizations interfering with the target ABI on such functions. For instance on i386 we now use register-passing conventions, but there are considerably more possibilities for target-specific optimization here. (In PIC compilation, one can, for instance, avoid recomputing of global offset table pointers in the prologues of local functions, and propagate the computation into callers.)

   (c) Construction of inlining plan: We make all the inlining decisions in advance and store them in call graph as a so-called "inlining plan." See Section 5 for details.

   (d) Another pass of unreachable function removal: in some cases, a function might be reachable only via a call in an extern inline function that was never inlined. Since the body of the extern inline function is never output, it is possible to remove all such functions, too. This scenario is very common for C++ programs.

   Note that it is very desirable not to touch the function bodies at this stage. In real whole program optimization, the functions are parsed and stored into "object files" containing intermediate representation of the program. The intra-procedural optimization phase executed in linker then should not need to load everything into memory at once and instead use the data files as a database reading the call-graph information first and using the function bodies just later in the compilation phase.

4. Expansion: We proceed in reverse DFS order on functions that are still present in the call-graph, applying inter-procedural optimizations such as inlining to the functions, and finally leaving them to the back-end to do the actual optimization and compilation.

   Function reordering allows more reliable propagation of information from the callee code generation into the caller. For instance, it is possible to generate a better call sequence when the callee's preferred stack frame boundary is known.

Such function ordering would permit implementation of more interesting optimizations too (for instance simple inter-procedural register allocation). On the other hand, it makes it almost impossible to avoid compilation of some function when its call has been optimized out. At the moment we make no attempts to solve this issue; however, in the future we may want to do early optimization during the analysis stage to catch most of these cases.

It also would be also desirable to defer output of global variables to this stage and output only the variables that are still referred by functions after the optimization. Implementing this feature is easy and we hope to do so in the near future.

# 3 Data-structures

Most of the code in the compilation driver actually manipulates only two data structures, that is, the call-graph and the variable pool.

## 3.1 The call-graph

The *call-graph* consist of nodes and edges represented via linked lists. Each function (external or not) corresponds to the unique node and each direct call has corresponding edge from caller to the callee.

The mapping from declarations to call-graph nodes is done using an hash table based on the declarations' `DECL_UID`, so it is essential that the frontend use single declaration ID for each function or variable. The call-graph nodes are created lazily using the `cgraph_node` function, when an unknown declaration is called.

When the call-graph is built, there is one edge for each direct call. The indirect calls are not represented at a moment. We simply mark each function with address taken as externally visible function. Optimizers then have to expect conservatively that each indirect call and/or call of unknown function might in turn call some of the entry points. The entry points are merged via flag `needed` in the call-graph node.

Finally there is a work list used to maintain nodes that are reachable from the entry points and thus needs to be analyzed or output into the file.

## 3.2 Data-structures for inter-procedural information

Call-graph is place to store data needed for inter-procedural optimization. All data-structures are divided into three components: `local_info` that is produced while analyzing the function, `global_info` that is result of global walking of the call-graph on the end of compilation and `rtl_info` used by RTL back-end to propagate data from already compiled functions to their callers.

The division has been made to make it possible to reduce memory usage in the future. Each of the field has different lifetimes and thus they don't necessarily need to be allocated all the time. At the moment the data-structures are small and thus all allocated at once with the call graph nodes, but the `cgraph_global_info`, `cgraph_local_info`, `cgraph_rtl_info` accessor functions shall be used to access the data. These functions already contain sanity checks that enforce the lifetimes of the individual data structures.

In the contrast, there is structure `function` allocated for each parsed function body traditionally used to store related information by many other parts of the compiler. This structure has no such organization and it consumes up to 25% of overall memory for some C++ programs. We hope to improve the situation

by reorganizing `struct function` similar way and moving to the call-graph nodes some of the data currently held in `struct function`, removing redundancies on where the information shall be stored.

### 3.3 The varpool data structure

In order to allow elimination of unused static data within the backend, we modified the interface to the output data-structures too. The varpool module is used to maintain variables in similar manner as call-graph is used for functions. At the moment it is implemented as a simple hash table containing entries for all global data-structures, and a worklist maintaining a list of variables that need to be output into assembly file. No dependencies or references are represented explicitly.

## 4 Front-end API

An important part of the new compilation driver design is the API to front-end. We tried hard to make it as easy to use as possible, however practice has shown that it is not always trivial to update existing front-ends to the new philosophy. Hopefully the API will still be natural to use in the new code.

All functions the front-end programmer shall be interested in are:

**cgraph_finalize_function** shall be called once front-end has parsed whole body of function and it is certain that the function body nor the declaration will change.

(As mentioned above, there is one exception needed for implementing GCC's `extern inline` functions, but it should not be used by new code.)

**cgraph_varpool_finalize_variable** has the same behavior but is used for file scope variables.

**cgraph_finalize_compilation_unit** shall be called called once parsing of compilation unit is finalized and trees representing it will no longer be changed by the front-end.

In unit-at-a-time mode, call-graph construction and local function analysis takes place here. Bodies of unreachable functions are released to conserve memory usage.

The compilation unit in this point of view should be compilation unit as defined by the language—for instance the C front-end allows multiple compilation units to be parsed at once and it should call this function each time parsing is done, in order to save memory. This is not what happens currently because the C front-end does global static variable renaming pass at the very end of compilation. As a result, unnecessary and duplicate function bodies are maintained in memory up to very end of the parsing process.

Modifying the C front-end to use this scheme is not an easy task. Merging of C compilation units together involve a lot of C language specific behavior and we need to consider whether it is feasible to implement that logic in the generic pass or through a some simple set of front-end hooks.

**cgraph_optimize** performs inter-procedural analysis and compile functions in unit-at-a-time mode (in function-at-a-time this function does nothing except for producing debug dumps). Front-end shall call this function at the very end of compilation, after releasing all those internal data-structures that are not passed to the back-end.

**cgraph_mark_needed_node** can be used when a function is referenced by some hidden way (for instance if it is marked by attribute `used`, which usually means that it is called in inline assembly code). The call-graph data structure is updated in a way that function is marked as entry point and thus it is never optimized as local function and always compiled.

**cgraph_varpool_mark_needed_node** has a similar meaning as function `cgraph_mark_needed_node`, but is used for variables.

To overcome problems in the front-end specific representation of trees, we had to implement two callbacks that allow a front-end to define front-end specific expansion of trees into RTL. We plan to eliminate these completely once the work on tree-ssa branch is finished.

**analyze_expr callback** This function should lower tree nodes not understood by generic code into understandable ones or, alternatively, should mark referenced call-graph and varpool nodes.

**expand_function callback** is used to expand the function into RTL form in front-end specific way. The front-end should not make any assumptions about when this function can be called. Existence of this hook is also used as a check on whether front-end supports unit-at-a-time API.

# 5 Inlining Heuristics

Only non-trivial inter-procedural optimization implemented at a moment is inlining we describe in this section. The inliner implementation can be used as an example how other inter-procedural optimizers can be implemented on the on the top of the new infrastructure, so we will describe it in greater detail.

### 5.0.1 Inlining plans

The function inlining information is decided in advance (in the optimization phase) and maintained in the call-graph in the so called inlining plan until the function is optimized. Once a function body is physically inlined into another, the callgraph data-structure is updated to reflect new program structure. This organization is critical to make it possible to save parsed function bodies into disk and make all inter-procedural optimizations without actually touhing the bodies and having them to resist in memory all at once.

The inlining decisions are reflected in the call-graph as follows: When the heuristics decide to inline given call-graph edge, the calle's node is cloned to represent the new function copy that will be later produced by inliner (so each inlined call of given function gets unique clone node and all the clones are linked together via linked list). Each edge has an "`inline_failed`" field. When the field is set to NULL, the call will be inlined. When it is non-NULL it contains an reason why inlining wasn't performed, that might be eventually output by the inliner when `-Winline` is specified.

We originally didn't clone the nodes and simply had a flag in each edge specifying whether the given call shall be inlined. This was found soon to have many limitations. For example, it is impossible to represent inline plans that are not *transitive* (i.e., once call of function $B$ in offline copy of function $A$ is inlined, each inline copy of function $A$ must have the function $B$ inlined as well). Non-transitive inlining plans are needed in order to let the programmer claim that all direct and indirect callees shall be inlined recursively; experience has shown that this kind of control is useful in template-heavy C++ numeric code.

Reorganizing the code to new scheme also

turned out to simplify significantly the estimates of overall code size growth caused by inlining, and allowed to release function body as soon as all of its inline copies are produced.

### 5.0.2 Profitability estimates

To make good inlining decisions, the profitability of inlining a given call must be estimated. Ideally, one might take into account the expected time spent in callee and compute how large relative speedup will elimination of the call overhead is. It is also desirable to take into account the new optimization possibilities and weight it with the expected code size growth. See for instance [1] for more discussion on the topic.

With current very high level and partly front-end specific intermediate representation it is difficult to do such a complex analysis and the profitability analysis actually represent the weakest spot of our implementation. At a moment we simply compute estimated function body size in front-end specific way by walking the tree representation and summing cost of the nodes. The majority of nodes has a cost of 1 with exception of a few nodes that are known to have zero cost (such as lexical scope regions or `__builtin_constant_p` calls) and a few others that are known to be expensive (such as division or function call) and are assigned a cost of 10. This implementation is still a noticeable improvement compared to previous implementations that were merely counting number of statements in the source and completely ignored the different complexities of individual constructs.

The cost of inlining given call is estimated as cost of increasing the callers body cost by callees cost minus 10 (eliminating the call). Our objective is to inline as many function calls before reaching given growth limits.

Toggether with developers from Apple we are working towards a better implementation of this analysis based on tree-ssa representation. This work is being done tree-profiling branch and will take into account the runtime call frequencies computed from the profile, allowing the compiler to perform a realistic estimate the costs of individual calls. We also plan to implement a partial specialization pass on functions that will notice situations where function body can be significantly simplified when some of its arguments are known. This project is however still far from being finished.

### 5.0.3 Limiting parameters

As discussed earlier, we provide set of parameters to avoid too extreme amount of inlining. The final set of parameters are just slightly more complicated than ones outlined in the introduction section:

**max-inline-insns-single** sets the maximum number of instructions (counted in GCC's internal representation) in a single function that the tree inliner will consider for inlining. This only affects functions declared inline and methods implemented in a class declaration (C++). The default value is 500.

**max-inline-insns-auto** sets limit on estimated size of inline candidates when `-finline-functions` (included in `-O3`) is used. The default value is 120.

**large-function-insns** is a limit that specifies which functions are considered to be "large": for functions greater than this limit, inlining is constrained by `--param large-function-growth`. This parameter is useful primarily to avoid

extreme compilation time caused by non-linear algorithms used by the back-end. The default value is 3000.

**large-function-growth** specifies maximal growth of large function caused by inlining in percent. The default value is 200.

**inline-unit-growth** specifies maximal overall growth of the compilation unit caused by inlining. This parameter is ignored when `-funit-at-a-time` is not used. The default value is 150.

#### 5.0.4 Global inlining heuristics

Given the rules established by these five parameters, inlining decisions are made in three passes. In the first pass all function calls marked with the `always_inline` attribute are inlined, so that other decisions cannot interfere with it.

In the second pass inlining of small functions is performed; all function candidates are put into a priority heap ordered by the estimated costs of inlining the function into all its callers and then they are inlined in priority order, updating the costs of other enqueued candidates until the heap is empty or the overall unit growth parameters reached.

This algorithm (often described as knapsack style, see [2]) seem to perform better than simple top-down and bottom-up heuristics resulting in more function calls to be inlined without breaking the same inline limits discussed above.

In the third pass all functions that are still called just once are inlined unless the callee body become too large.

Finally the fourth pass does so-called "recursive inlining." When the function contains recursive calls and its body is called, the calls are inlined up to recursion depth computed in a way so function reach size specified by parameter. This optimization has similar effect as loop unrolling.

#### 5.0.5 Incremental inlining heuristics

The global inlining heuristics can not be used in function-at-a-time mode and thus there is an alternative implementation of simple bottom up inlining heuristics. Most of the code (checking of limits and updating call-graph) is shared in between the implementations and thus the implementation is pretty straight forward.

The major problem of this heuristics appears to be in fact that the overall compilation unit growth argument is ignored. In some extreme C++ test cases (such as those based on POOMA library) the compiler now compiles faster at `-O2` compilation level compared to `-O1`.

## 6 Experimental Results

Evaluating the effectiveness of new infrastructure is difficult task. The benefits (and losses) vary greatly together with the coding style of the tested application. Very good results can be measured in the template heavy C++ code, such as the DLV application or POOMA library that we use as a benchmark suite. The table 1 summarizes the results of DLV benchmark suite evolving over various GCC releases and it is easy to notice the degradation in performance in GCC 3.0, as well as a reduction of code size caused by decreasing inline limits to avoid compile time problems as mentioned earlier. This problem remained apparent until GCC 3.3 despite quite serious attempts to tune the heuristics. GCC 3.4 behaves quite well in both function-at-a-time and unit-at-a-

time heuristics, but the code size has increased noticeably. For this particular benchmark it is possible to reduce the inlining limits somewhat and get code sizes smaller than GCC 2.95 without considerable performance regressions; reducing the limits, however, hurts performance in other benchmarks signalizing that the profitability analysis needs more work. Unfortunately it is no longer possible to present GCC 3.4 numbers with the old heuristics, but the initial tests did already show benefits similar as ones compared to GCC 3.3 so we believe that majority of the improvements actually come from inlining in this particular case.

The author evaluated number of template heavy test cases while working on new implementation, and the benefits can be virtually infinite scaling with complexity of the code. For test case based on POOMA library, compilation times went down from 25 minutes to 1 minute with noticeable improvements in execution time too.

On the other hand, the C and Fortran benchmarks shows a much more moderate improvement. Table 3 shows benchmarks made on AMD Opteron chip in 32bit and 64bit mode. While majority of the tests improve, the benefits are less noticeable. The good news, however, are that the unit-at-a-time reduce code size almost consistently on the `-O2` level of optimization. On the other hand the `-O3` scores demonstrate that backward inlining can cause code size growth without major changes in the performance.

By comparing the 64-bit and 32-bit scores, one also can notice the benefits of register passing conventions.

One area where author was hoping for considerable improvement is performance of desktop applications. It is difficult to present the benchmarks of the GUI application but the simple test of compiling x86-64 KDE and Mozilla source gave savings of 7.4% and 6.6% respectively in the overall size of stripped binaries, and a partial i386 Open-Office build gave 22% savings. These savings ought to bring a noticeable improvement in execution time and reduction of memory usage too. In addition the performance of code shall be improved similar way as in the DLV application benchmark presented here.

It remains to discuss the memory usage of the compiler. Again it is not difficult to present extreme improvements (for example, compiling the POOMA library only requires 2% of the memory) as well as extreme regressions: a huge compilation unit consisting of small but uninlinable functions will result in arbitrarily high unit-at-a-time peak memory usage, without increasing peak usage in function-at-a-time mode.

Real world application however show that compilation units usually require less memory, both because they are not very large and also because the lifetime data structures used by the front-end in unit-at-a-time mode does not overlap with the lifetime of data structures used by backend; in addition, unneeded functions and data-structures are released early.

Table 2 shows peak GGC memory usage while compiling some of relatively large source files. The numbers were obtained by compiling with `--param ggc-min-expand=0 --param ggc-min-heapsize=2048 -Q` and examining the GGC debug output for largest memory usage after the collection. The `generate.ii` is a large test case of template heavy code, while `combine.c` is one of largest source files of GCC. The graph of memory usage in unit-at-a-time of the C++ testcase is almost flat demonstrating that the pass releasing unneeded function bodies release enough memory so the back-end no longer increase the peak. For the C test case

there are small regression at `-O1` and `-O2` but author would hope that these won't prevent unit-at-a-time from being enabled by default in the future.

## 7 Contributors

The project would be impossible without following contributions: Steven Bosscher reorganized f90 front-end, reviewed early implementations of the inlining code and made a number of cleanups. Richard Günther provided a lot of feedback about POOMA library issues. He also implemented patch for "`leafify`" function attribute that brought major motivation for reorganization of the inlining plans representation. Richard Henderson reviewed most of the call-graph code. Gerald Pfeifer provided the DLV benchmark that has turned out to be extremely useful to tune the heuristics and gave a lot of useful feedback. Jeff Sturm revamped the Java front-end to cgraph code. Mark Mitchell helped to choose feasible way on how to reorganize C++ compiler, reviewed the changes and helped to solve some of issues. Zack Weinberg reorganized the code to not use hash tables based on assembler names.

A number of SUSE developers (mainly Andreas Jaeger, Andi Kleen and Michael Matz) helped to test GCC on SUSE distribution build and analyzed/fixed many of compatibility issues and implementation defects so the implementation was ready for production use before the offical GCC 3.4 release.

Author would also like to thank to Paolo Bonzini and John W. Lockhart who helped to proofread the paper.

## References

[1] *Towards Better Inlining Decisions Using Inlining Trials* (1994), Jeffrey Dean, Craig Chambers

[2] *A Comparative Study of Static and Profile-Based Heuristics for Inlining* (2000), Matthew Arnold, Stephen Fink, Vivek Sarkar, Peter F. Sweeney

Table 1: Speedup in the DLV Benchmark relative GCC 2.95
Execution times in second and relative comparisons to GCC 2.95, smaller is better.

| benchmark | GCC 2.95 | 3.0.4 | | 3.3.2 | | 3.4 -fno-unit… | | 3.4 -funit-at… | |
|---|---|---|---|---|---|---|---|---|---|
| STRATCOMP1-ALL | 2.45s | 24.92s | 1017.00% | 4.68s | 191.00% | 8.31s | 339.00% | 2.58s | 105.00% |
| STRATCOMP-770.2-Q | 0.49s | 0.57s | 116.00% | 1.22s | 248.00% | 0.47s | 95.00% | 0.45s | 91.00% |
| 2QBF1 | 10.92s | 13.96s | 127.00% | 28.68s | 262.00% | 11.06s | 101.00% | 9.33s | 85.00% |
| PRIMEIMPL2 | 7.52s | 8.75s | 116.00% | 43.60s | 579.00% | 6.27s | 83.00% | 6.00s | 79.00% |
| 3COL-SIMPLEX1 | 4.68s | 4.97s | 106.00% | 11.13s | 237.00% | 4.56s | 97.00% | 4.34s | 92.00% |
| 3COL-RANDOM1 | 6.66s | 8.15s | 122.00% | 38.14s | 572.00% | 5.95s | 89.00% | 5.86s | 87.00% |
| HP-RANDOM1 | 4.93s | 5.72s | 116.00% | 18.44s | 374.00% | 5.23s | 106.00% | 4.44s | 90.00% |
| HAMCYCLE-FREE | 0.80s | 1.12s | 140.00% | 4.96s | 620.00% | 1.03s | 128.00% | 0.72s | 90.00% |
| DECOMP2 | 8.44s | 9.59s | 113.00% | 33.91s | 401.00% | 8.53s | 101.00% | 7.87s | 93.00% |
| BW-P5-nopush | 4.45s | 4.85s | 108.00% | 12.90s | 289.00% | 4.25s | 95.00% | 4.19s | 94.00% |
| BW-P5-pushbin | 3.79s | 4.05s | 106.00% | 12.61s | 332.00% | 3.44s | 90.00% | 3.40s | 89.00% |
| BW-P5-nopushbin | 1.21s | 1.31s | 108.00% | 4.07s | 336.00% | 1.13s | 93.00% | 1.09s | 90.00% |
| HANOI-Towers | 2.05s | 2.19s | 106.00% | 6.21s | 302.00% | 1.94s | 94.00% | 1.82s | 88.00% |
| RAMSEY | 5.34s | 5.69s | 106.00% | 16.69s | 312.00% | 4.83s | 90.00% | 4.58s | 85.00% |
| CRISTAL | 5.30s | 5.91s | 111.00% | 12.67s | 239.00% | 5.14s | 96.00% | 4.75s | 89.00% |
| 21-QUEENS | 6.35s | 7.31s | 115.00% | 40.15s | 632.00% | 5.09s | 80.00% | 4.86s | 76.00% |
| MSTDir[V=13,A=40] | 12.58s | 14.46s | 114.00% | 41.77s | 332.00% | 9.14s | 72.00% | 8.60s | 68.00% |
| MSTDir[V=15,A=40] | 12.62s | 14.49s | 114.00% | 41.44s | 328.00% | 9.15s | 72.00% | 8.53s | 67.00% |
| STUndir[V=13,A=40] | 6.47s | 7.57s | 117.00% | 25.48s | 393.00% | 4.96s | 76.00% | 4.61s | 71.00% |
| TIMETABLING | 7.08s | 7.37s | 104.00% | 18.21s | 257.00% | 6.30s | 88.00% | 5.90s | 83.00% |
| compilation time | 2m42s | 2m53s | 106.7% | 2m47s | 103% | 2m9s | 79.6% | 2m28s | 91.3% |
| Code size | 1251k | 622k | 49.7% | 1562k | 124.8% | 1808k | 144.5% | 1628k | 130.1% |

| test | optimization level | function-at-a-time | unit-at-a-time | savings |
|---|---|---|---|---|
| generate.ii | -O0 | 33563K | 32606K | 2.9% |
| generate.ii | -O1 | 33462K | 32606K | 2.9% |
| generate.ii | -O2 | 43296K | 33239K | 30% |
| generate.ii | -O3 | >55077K | 33411K | >64% |
| combine.c | -O0 | 3655K | 3625K | 1.1% |
| combine.c | -O1 | 3199K | 3531K | -11% |
| combine.c | -O2 | 3450K | 3609K | -4.0% |
| combine.c | -O3 | 6245K | 4086K | 52% |

Table 2: Peak GGC memory usage

Table 3: 64-bit SPECint 2000 -fnon-unit-at-a-time compared to -funit-at-a-time

Performance (relative speedup in percent, bigger is better):

| options | gzip | vpr | gcc | mcf | crafty | parser | eon | perl | gap | vortex | bzip2 | twolf | avg |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `-O2` | -0.89 | 1.22 | 0.72 | 0.00 | 0.42 | 0.35 | 3.51 | 4.84 | -1.19 | 3.27 | 0.12 | -4.36 | 0.24 |
| `-O2 -m32` | -0.71 | 4.02 | 0.21 | -0.19 | -1.60 | 0.15 | 10.39 | 1.64 | -1.82 | -0.19 | 0.14 | -0.61 | 0.86 |
| `-O3` | -0.52 | 4.08 | 0.93 | 0.00 | 0.36 | 0.34 | 5.27 | 0.00 | 0.50 | -0.50 | -0.38 | -4.27 | 0.11 |
| `-O3 -m32` | -0.50 | 7.77 | -1.93 | 0.00 | -1.89 | -0.71 | 6.36 | 0.96 | 0.26 | 1.52 | -0.28 | -1.53 | 0.61 |
| `-O3` + profile | -1.78 | 3.91 | 0.19 | 0.00 | -0.37 | -0.35 | 3.84 | 3.91 | -6.37 | -1.61 | 0.49 | -0.74 | 0.00 |
| `-O3 -m32` + profile | -0.96 | 10.04 | 0.52 | 0.18 | 0.10 | 0.42 | 10.16 | 2.78 | -0.89 | -0.63 | 0.95 | 2.12 | 2.04 |

File size (relative increase of the size of stripped binaries in percent):

| options | gzip | vpr | gcc | mcf | crafty | parser | eon | perl | gap | vortex | bzip2 | twolf | total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| `-O2` | -20.42 | -5.62 | -2.08 | 0.00 | -0.02 | 0.00 | -8.58 | -1.08 | -0.10 | -1.41 | 0.00 | 0.46 | -2.63 |
| `-O2 -m32` | -19.93 | -2.66 | -2.47 | 0.00 | 0.10 | -0.03 | -7.98 | -0.89 | -0.09 | -0.87 | 0.00 | -0.05 | -2.44 |
| `-O3` | -13.79 | -1.47 | 5.14 | 0.00 | 3.68 | 4.17 | -3.89 | 4.45 | 2.22 | 1.13 | 12.36 | 5.23 | 2.60 |
| `-O3 -m32` | -12.72 | 3.62 | 5.48 | 0.00 | 4.33 | 5.28 | -3.66 | 4.87 | 2.81 | 1.01 | 18.79 | 7.48 | 3.24 |
| `-O3` + profile | -14.41 | -1.33 | 5.18 | 0.00 | 2.35 | 4.12 | -3.60 | 4.95 | 2.58 | 0.72 | 13.23 | 4.83 | 2.62 |
| `-O4 -m32` + profile | -12.30 | 3.66 | 5.66 | 0.00 | 4.34 | 5.43 | -3.68 | 5.21 | 2.99 | 1.02 | 18.29 | 5.79 | 3.29 |

Performance (relative speedup in percent, bigger is better):

| options | wupwise | swim | mgrid | applu | mesa | art | equake | ammp | apsi | total |
|---|---|---|---|---|---|---|---|---|---|---|
| `-O2` | 0.00 | 0.14 | 0.00 | 0.00 | -0.70 | 0.32 | -0.13 | 0.00 | 0.00 | 0.00 |
| `-O2 -m32` | -0.13 | 0.00 | 0.00 | 0.00 | -1.36 | 1.48 | 0.72 | 0.00 | 0.00 | 0.17 |
| `-O3` | 0.00 | 0.00 | 0.00 | 0.17 | -3.51 | 0.63 | 4.87 | 0.00 | 0.00 | 0.14 |
| `-O3 -m32` | 1.36 | 0.29 | -0.18 | 0.00 | 4.67 | 1.89 | 3.75 | 0.00 | 0.00 | 1.02 |
| `-O3` + profile feedback | 0.11 | 0.43 | 0.00 | 0.00 | 3.35 | 1.92 | 1.74 | 0.00 | 0.00 | 0.86 |
| `-O3 -m32` + profile feedback | 0.00 | 0.00 | 0.18 | 0.00 | 7.36 | 2.80 | 3.01 | 0.00 | 0.00 | 1.19 |

File size (relative increase of the size of stripped binaries in percent):

| options | wupwise | swim | mgrid | applu | mesa | art | equake | ammp | apsi | total |
|---|---|---|---|---|---|---|---|---|---|---|
| `-O2` | 0.00 | 0.00 | 0.00 | 0.00 | -1.73 | 0.00 | 0.00 | 0.00 | 0.00 | -0.47 |
| `-O2 -m32` | 0.00 | 0.00 | 0.00 | 0.00 | -1.32 | 0.00 | 0.24 | 0.00 | 0.00 | -0.35 |
| `-O3` | 0.00 | 0.00 | 0.00 | 0.00 | -0.23 | 0.00 | 0.85 | 0.00 | 0.00 | -0.06 |
| `-O3 -m32` | 0.00 | 0.00 | 0.00 | 0.00 | -0.24 | 1.35 | 5.57 | 0.00 | 0.00 | 0.02 |
| `-O3` + profile feedback | 0.00 | 0.00 | 0.00 | 0.00 | -0.24 | 0.00 | 0.00 | 0.00 | 0.00 | -0.07 |
| `-O3 -m32` + profile feedback | 0.00 | 0.00 | 0.00 | 0.00 | -0.21 | 1.43 | 5.16 | 0.00 | 0.00 | 0.03 |

# Code Factoring in GCC

*Gábor Lóki, Ákos Kiss, Judit Jász, and Árpád Beszédes*
Department of Software Engineering
Institute of Informatics
University of Szeged, Hungary
{loki,akiss,jasy,beszedes}@inf.u-szeged.hu

## Abstract

Though compilers usually focus on optimizing for performance, the size of the generated code has only received attention recently. On general desktop systems the code size is not the biggest concern, but on devices with a limited storage capacity compilers should strive for as small a code as possible. GCC already contains some very useful algorithms for optimizing code size, but code factoring – a very powerful approach to reducing code size – has not been implemented yet in GCC. In this paper we will provide an overview of the possibilities of using code factoring in GCC. Two code factoring algorithms have been implemented so far. These algorithms, using CSiBE as a benchmark, produced a maximum of 27% in code size reduction and an average of 3%.

## 1 Introduction

In the recent years handheld devices such as PDAs, telephones and smartphones are becoming more important. With these systems the amount of runtime memory and storage capacity is often very limited but at the same time the need for more sophisticated software is increasing. Hence powerful size reducing methods are required to cram new features into the applications.

Although GCC already contains size reducing algorithms, further optimization techniques are needed since GCC is already used for compiling for handheld devices. The official compiler for the increasingly popular Symbian OS-based mobile phones is GCC [8], some PDAs like the iPAQs already have Linux ports [9] (where, needless to say, the default compiler is GCC) and Linux-based mobile phones are also available.

In this paper we will provide an overview on code factoring, a class of powerful optimization techniques for code size reduction, and present a new, enhanced algorithm for procedural abstraction. These algorithms have been implemented in GCC and have resulted in 3% code size reductions on average, while achieving a 27% reduction in the best cases, based on the CSiBE benchmark [5].

The rest of the paper is organized as follows. In Section 2 we introduce code factoring and present a new enhancement for procedural abstraction. In Section 3 we discuss some details of the implementation of the algorithms in GCC, while in Section 4 we give our experimental results. Finally, in Section 5 we present our conclusions and future plans.

## 2 Code Factoring

Code factoring is the name of a class of useful optimization techniques developed explicitly for code size reduction [1, 2, 3, 4]. These approaches aim to reduce size by restructuring the code. The following subsections will discuss two code factoring algorithms, one of which works with individual instructions, while the other handles longer instructions sequences.

### 2.1 Local Factoring

The optimization strategy of local factoring (also known as local code motion, code hoisting and code sinking) is to move identical instructions from basic blocks to their common predecessor or successor, if they have any. The semantics of the program have to be preserved of course, thus only those instructions which neither invalidate any existing dependences nor introduce new ones may be moved. Figure 1a shows a control-flow graph (CFG) with basic blocks containing identical instructions. To obtain the best size reduction some of the instructions are moved upwards to the common predecessor, while some are moved downwards to the common successor. Figure 1b shows the result of the transformation.

Let us now consider some more complicated cases. While not frequent, it may occur that multiple basic blocks have more than one predecessors, all of which are common. In this case, if the basic blocks in question have identical instructions and the number of predecessors is less than the number of the examined blocks, then the instructions shall be moved to all the predecessors. Figure 2 depicts this case. A similar situation is when basic blocks have more than one common successors (see Figure 3.) Furthermore, in the case of sinking even those instructions that are not present in all of the blocks may be moved by creating a new

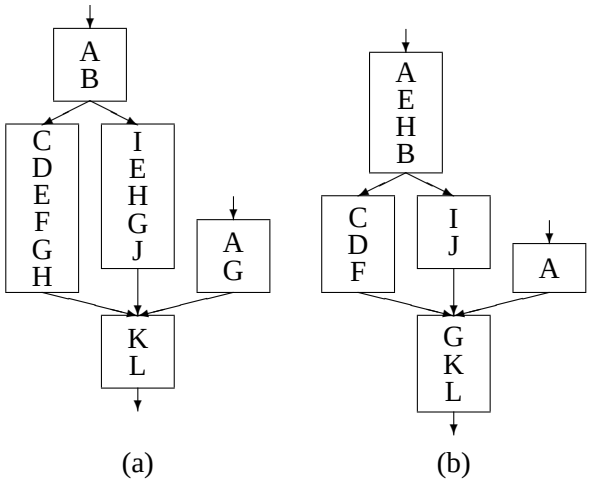



Figure 1: Local code factoring. CFG (a) before and (b) after the transformation. Identical letters denote identical instructions.

successor block for them. Figure 4 shows an example CFG for this case.

Except for this last case, which involves the creation of a new basic block, local factoring has an additional benefit of being good for performance also.

### 2.2 Procedural Abstraction

Procedural abstraction is a size optimization method which, unlike local factoring, works with whole single-entry single-exit code fragments (instruction sequences smaller than a basic block, whole blocks or even larger units) instead of single instructions. The main idea of this technique is to find identical regions of code, which can be turned into procedures, and then replace all occurrences with calls to the newly created subroutine.

The existing solutions [2, 4] can only deal with such code fragments that are either identical or equivalent in some sense or can be transformed somehow (e.g. by means of register renaming) to an equivalent form. However, these approaches fail to find an optimal solution for

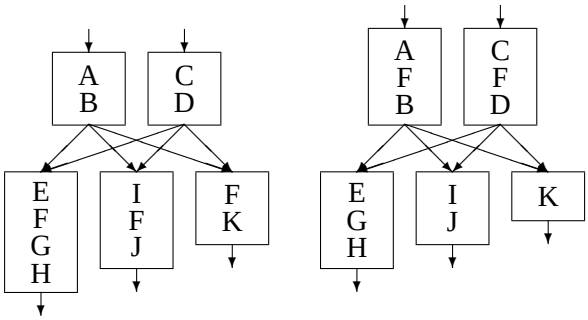



Figure 2: Basic blocks with multiple common predecessors (a) before and (b) after local factoring.

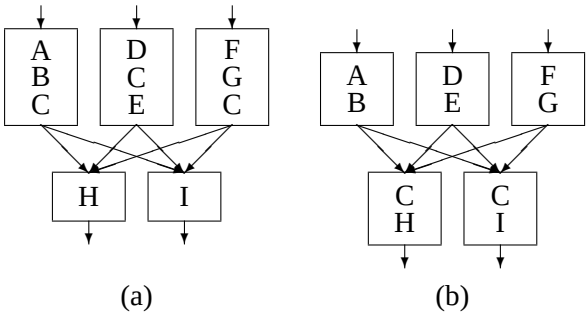



Figure 3: Basic blocks with multiple common successors (a) before and (b) after local factoring.

those cases where an instruction sequence is equivalent to another one, while a third one is only identical with its suffix (as shown in Figure 5a). The current solutions either choose to abstract the longest possible sequence into a function and leave the shorter one unabstracted (Figure 5b) or turn the instructions common in all sequences into a function and create another new function from the remaining common part of the longer sequences, thus introducing the overhead of the inserted extra call/return code (Figure 5c).

In this paper we propose to create multiple-entry functions in the cases described above to allow the abstraction of instruction sequences

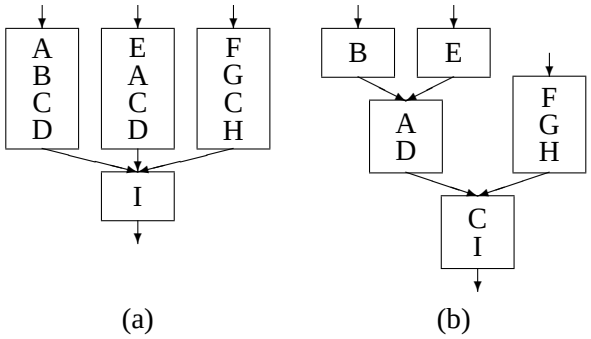



Figure 4: Basic blocks with multiple common successors but only partially common instructions (a) before and (b) after local factoring.

of differing lengths without the overhead of superfluous call/return code. The longest possible sequence shall be chosen as the body of the new function and entry points need to be defined according to the length of the matching sequences. Each matching sequence has to be replaced with a call to the appropriate entry point of the new function. Figure 5d shows the optimal solution for the problem depicted in Figure 5a.

Needless to say, procedural abstraction introduces some performance overhead with the execution of the inserted call and return code. Moreover, the size overhead of the inserted code must also be taken into account. The abstraction shall only be carried out if the gain resulting from the elimination of duplicates exceeds the loss arising from the insertion of extra instructions.

## 3 Implementation details

GCC already contains some algorithms similar to those discussed in Section 2, but they usually reduce code size only if the transformation does not introduce a (significant) performance overhead. Furthermore, they are usually of less potential than the previously described ones. The *cross-jumping* algorithm merges identical

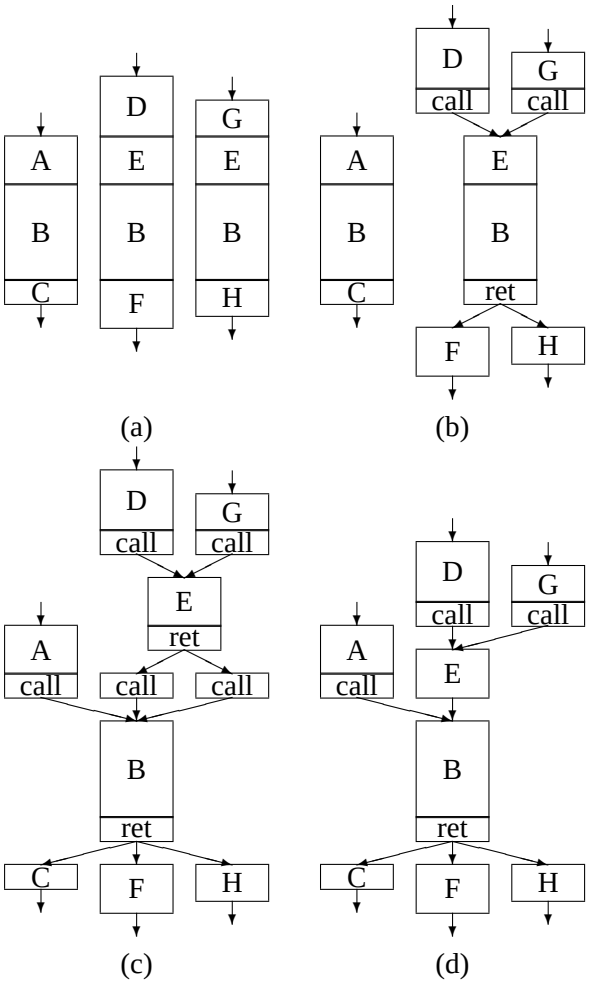



Figure 5: Abstraction of (a) instruction sequences of differing lengths to procedures using different strategies (b,c,d). Identical letters denote identical sequences.

tails of basic blocks, but this approach can only deal with a very limited subset of the generic problems of procedural abstraction. Another algorithm, called *if conversion*, has a similar effect on the code as local factoring when followed by a *combine* phase. As contrast to local factoring, *if conversion* is bound to conditional jumps.

Both of the new algorithms have been implemented as new RTL optimization phases in GCC (a snapshot taken from mainline on 2004-03-10 12:00:00 UTC). Using the RTL representation algorithms can optimize only one function at a time. Although procedural abstraction is inherently an interprocedural optimization technique, it can be adapted to intraprocedural operation. Instead of creating a new function from the identical code fragments, one representative instance of them has to be retained in the body of the processed function and all the other occurrences will be replaced by code transferring control to the retained instance. To preserve the semantics of the original program, however, the point where control has to be returned after the execution of the retained instance must be remembered somehow, thus the subroutine call/return mechanism has to be mimed. In the current implementation we use labels to mark the return addresses, registers to store references to them and jumps on registers to transfer control back to the "callers."

Unfortunately, the current implementation of the enhanced procedural abstraction algorithm suffers from the problem of increasing the compilation time by a factor of 2–4 on average. This stems from the complex problem of finding the optimal candidates for abstraction. However, we hope that by applying more efficient algorithms we will be able to bring down the compilation time factor to a manageable level.

For the sake of simplicity, local factoring has been split into two parts and implemented in GCC as two individual algorithms. One of the algorithms implements the hoisting of instructions, i.e. moving them upwards to their predecessor blocks, while the other one is responsible for the sinking of the instructions, that is move them downwards to their successor basic blocks. A central problem for both algorithms is to decide whether an instruction may be moved freely out from its block. An instruction cannot be moved across instructions, which use parameters defined by the instruc-

tion or define parameters used or defined by the instruction. GCC provides methods for gathering the required definition/use information for the whole processed function. However, from a local factoring point of view, these methods are too expensive since only a small portion of the computed information is used. Therefore the implementation contains a "slim" version of the definition/use calculation code. Being sensitive to the compilation time in the implementation, we also made it possible to parameterize the maximum number of instructions the algorithms should analyze starting from the top or bottom of the basic blocks when looking for candidates of motion.

The implementation of the two algorithms are publicly available. They have been sent in form of patches to the appropriate mailing list [6, 7].

## 4 Results

On examining code size we found the code factoring algorithms had impressive effects. We evaluated the discussed algorithms with the help of CSiBE, the GCC Code Size Benchmark Environment, version 1.1.1, and found that a 3% code-size reduction can be achieved on average, but in some cases they are able to produce reduction ratios as high as 27%. Table 1 details the average code size reduction achieved by each algorithms on some relevant targets. The table also shows the combined effect of the techniques. The figures are relative to the unmodified GCC optimizing for size, i.e. optimizing with `-Os`. Table 2 shows the best figures for each algorithm.

| Target | Local Factoring | Procedural Abstraction | Combined |
|---|---|---|---|
| arm-elf | 0.148% | 2.785% | 3.120% |
| i386-elf | 0.701% | 1.356% | 2.052% |
| i686-linux | 0.696% | 1.448% | 2.143% |
| m68k-elf | 0.092% | 2.312% | 2.401% |

Table 1: Average code size reduction achieved by code factoring algorithms.

| Target | Local Factoring | Procedural Abstraction | Combined |
|---|---|---|---|
| arm-elf | 3.794% | 27.230% | 27.342% |
| i386-elf | 14.621% | 13.210% | 16.795% |
| i686-linux | 11.592% | 13.261% | 17.389% |
| m68k-elf | 1.468% | 23.174% | 23.174% |

Table 2: Maximum code size reduction achieved by code factoring algorithms.

## 5 Conclusion and future plans

In this paper we gave an overview of two code factoring algorithms and provided an enhancement to procedural abstraction, which provides superior results compared to the existing solutions. We implemented the discussed algorithms in GCC and achieved a 3% code-size reduction on average, based on the CSiBE benchmark. In the best cases the optimizations yielded reduction ratios as high as 27%.

From the nature of procedural abstraction it follows that it can optimize larger inputs better than small ones. To be able to utilize the full potential of the algorithm the current implementation has to be modified so that it can work interprocedurally, which means a unit-at-a-time in GCC terminology instead of working intraprocedurally, i.e. transforming only one function at a time. This may necessitate rewriting the implementation so it can work on the GIMPLE representation, as some feedback already suggested. We are also aware of the algorithm complexity problem and have been striving to improve the implementation in order to reduce the compilation time by applying more efficient algorithms.

We are already investigating the possibility of making the local factoring implementation

work on GIMPLE also, even if the algorithm cannot be extended to work interprocedurally, since GIMPLE is now preferred over RTL. Our preliminary results are very promising.

When we have finished with our ongoing research, we also plan to consider the adaptation and implementation of other algorithms in GCC such as the procedural abstraction of single-entry single-exit regions larger than a basic block or the compaction of matching single-entry multiple-exit regions.

## References

[1] Wen-Ke Chen, Bengu Li, and Rajiv Gupta. Code compaction of matching single-entry multiple-exit regions. In *Proc. 10th Annual International Static Analysis Symposium*, pages 401–417, June 2003.

[2] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 139–149, 1999.

[3] Bjorn de Sutter, Bruno de Bus, Koen de Bosschere, and Saumya Debray. Combining global code and data compaction. In *Proc. ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001.

[4] Saumya K. Debray, William Evans, Robert Muth, and Bjorn de Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[5] Department of Software Engineering, University of Szeged. GCC code-size benchmark environment (CSiBE). `http://www.inf.u-szeged.hu/CSiBE`.

[6] Department of Software Engineering, University of Szeged. [patch] Local factoring algorithms. `http://gcc.gnu.org/ml/gcc-patches/2004-03/msg01907.html`.

[7] Department of Software Engineering, University of Szeged. [patch] Sequence abstraction. `http://gcc.gnu.org/ml/gcc-patches/2004-03/msg01921.html`.

[8] Symbian Ltd. Symbian OS. `http://www.symbian.com`.

[9] The Familiar Project. Familiar distribution. `http://familiar.handhelds.org`.

# Fighting register pressure in GCC

*Vladimir N. Makarov*
Red Hat
vmakarov@redhat.com

## Abstract

The necessity of decreasing register pressure in compilers is discussed. Various approaches to decreasing register pressure in compilers are given, including different algorithms of register live range splitting, register rematerialization, and register pressure sensitive instruction scheduling before register allocation.

Some of the mentioned algorithms were tried and rejected. Implementation of the rest, including region based register live range splitting and rematerialization driven by the register allocator, is in progress and probably will be part of GCC. The effects of discussed optimizations will be reported. The possible directions of improving the register allocation in GCC will be given.

## Introduction

Modern computers have several levels of storage. The faster the storage, the smaller its size. This is the consequence of a trade-off between the computer speed and its price. The fastest storage units are registers (or *hard registers*). They are not enough to store the values of operations and directly referred variables for any serious program.

It is very hard to force any optimization in a compiler (especially in a portable one) to use the hard registers effectively. Therefore most of compiler optimizations is written as if there is infinite number of virtual registers called *pseudo-registers*. The optimizations use them to store intermediate values and values of small variables. Although there is an untraditional approach to use only memory to store the values. For both approaches we need a special pass (or optimization) to map pseudo-registers onto hard registers and memory; for the second approach we need to map memory into hard-registers instead of memory because most instructions work with hard-registers. This pass is called register allocation.

A good register allocator becomes a very significant component of an optimized compiler nowadays because the gap between access times to registers and to first level memory (cache) widens for the high-end processors. Many optimizations (especially interprocedural and SSA-based ones) tend to create lots of pseudo-registers. The number of hard-registers is the same because it is a part of architecture. Even processors with new architectures containing more hard-registers need a good register allocator (although in less degree) because the programs run on these computers tend to be more complicated too.

The register allocator is one or more compiler components that could be considered as ones solving two major tasks (mostly in an integrated way). The first and most interesting one is to decrease register pressure to the level defined by the number of hard registers by different transformations. And the second one is to assign hard registers to pseudo-registers effec-

tively.

So what is register pressure? There are two commonly used definitions. The wide one is the number of hard registers needed to store values of the pseudo-registers at given program point. Another one is the number of living pseudo-registers.

There are a lot of known transformations that decrease register pressure. Some of these transformations generate code which could and should be corrected later. Some transformations are easily and naturally integrated with other transformations, such as the ones decreasing register pressure, assigning hard registers, and fixing the pitfalls of the previous transformations (such as register coalescing in a colouring based register allocator). Some of them are hard to integrate in one pass.

Currently GCC has two register allocators. The new one was written about two years ago and is described in details in [Matz03]. It is based on the Chaitin, Briggs, and Appel approaches to register allocation [Chaitin81, Briggs94, Appel96].

The old register allocator (I will call it *the original register allocator*) has been existing since the very first version of GCC. It was written by Richard Stallman. Some of its important components stayed practically unchanged since the first version. Richard Stallman took the register allocator design from a portable Pastel (an extension of the programming language Pascal) compiler written in Livermore Laboratories [Stallman04]. The design of the Pastel register allocator (which actually was a second version for the Pastel compiler) is very similar to the GCC one [Killian04]—they both have the same separation on a pass assigning hard registers to pseudo-registers and a pass which actually changes the code following the assignment and, if it is not possible, generates additional instructions to reload the registers.

Despite its lack of many modern optimizations present in the new register allocator, the original register allocator can easily compete with the new one in terms of compiler speed, quality of the generated code and size of code. This was a major reason for me to start work on improving the original register allocator. After thorough investigation, I found that the method of assigning hard registers is very similar to the *priority based colouring* register allocator [Chow84, Chow90], although it is more similar to the modifications described in [Sorkin96]. It was confirmed later.

Chow's approach is a real competitor to the Chaitin/Briggs approach. Some advantages of Chow's approach are acknowledged even by Preston Briggs [Briggs89]. Chow's algorithm is used in SGI Pro64 [Pro64] compiler and derived compilers like Open64 [Open64] and ORC [ORC]. For example, as Briggs' optimistic colouring, Chow's algorithm easily finds hard-registers for the diamond conflict graph (see Figure 1).

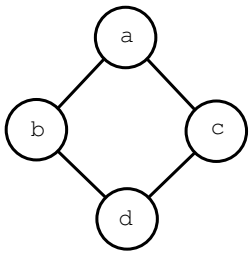



Figure 1: Diamond graph

All that was mentioned above was a major motivation to start work on improvement of the original register allocator. This article is focused on improving the original GCC register allocator. The first section describes the original GCC register allocator. The second section describes the method for decreasing the register pressure for the original register allocator based on register live range splitting. The third section describes decreasing regis-

ter pressure based on the live range shrinking approach. The fourth section describes other possible improvements to the original register allocator. The fifth section gives conclusions from my work.

# 1 The original register allocator in GCC

The original register allocator contains a lot of passes. Figure 2 describes the major passes and their order.

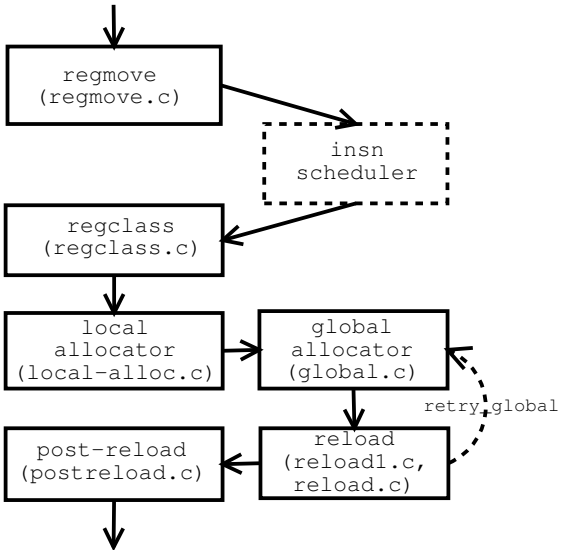



Figure 2: The original register allocator

**The regmove** pass is usually not considered to be a part of the original register allocator. I included it because the pass solves one task (register coalescing) peculiar to register allocators. The pass removes some register moves if the registers have the same value and it can be found in a basic block scope. Although the major task of regmove is to generate move instructions to satisfy two operand instruction constraints when the destination and source registers should be the same. The reload pass can solve this task too but in a less effective manner.

If register coalescing and global value numbering (mentioned in Section 4) are a part of GCC, we could try to remove register coalescing from this pass.

**The instruction scheduler** is not a part of the original register allocator. It is present just to show GCC's major passes starting with the regmove pass. Although the instruction scheduler could solve the task of decreasing register pressure (see section "register pressure sensitive instruction scheduling").

**Regclass.** GCC has a very powerful model for describing the target processor's register file. In this model there is the notion of register class. The register class is a set of hard registers. You can describe as many register classes as possible. Of course, they should reflect the target processor's register file. For example, some instructions can accept only a subset of all registers. In this case you should define a register class for the subset. Any relations are possible between different register classes: they can intersect or one register class can be a subset of another register class (there are reserved register classes like `NO_REGS` which does not contain any register or `ALL_REGS` which contains all registers).

The pass regclass (file `regclass.c`) mainly finds the *preferred* and *alternative* register classes for each pseudo-register. The preferred class is the smallest class containing the union of all register classes which result in the minimal cost of their usage for the given pseudo-register. The alternative class is the smallest class containing the union of all register classes, the usage of which is still more profitable than memory (the class `NO_REGS` is used

for the alternative if there are no such registers besides the ones in the preferred class).

It is interesting to note that the pass also implicitly does code selection. Regclass works in two passes. On the first pass, it defines the preferred and alternative classes without taking the possible classes of other operands into account. For example, an instruction with two operand pseudo-registers exists in two variants; one accepting classes *A* and *B*, and other one accepting *C* and *D*. On the first pass, the algorithm does not see that the variant with classes *A* and *D* will be more costly because it will require the generation of additional move instructions. On the second pass, the algorithm will take it into account. As a result the preferred or alternative class of a pseudo-register could change. This means two passes are not enough to find the preferred and alternative classes accurately; but it is a good approximation.

The file regclass.c also contains functions to scan the pseudo-registers to find general information about them (like the number of references and sets of pseudo-registers, the first and last instructions referencing the pseudo-registers etc.).

**The local allocator** assigns hard-registers only to pseudo-registers living inside one basic block. The result of the work is stored in the global array `reg_renumber` whose element values indexed by pseudo-register numbers are hard-registers assigned to the corresponding pseudo-registers.

Besides assigning hard-registers, the local allocator does some register coalescing too: if two or more pseudo-registers shuffled by move instructions do not conflict, they always get the same hard-registers. The global allocator also tries to do this in a less general way. The local allocator also performs a simple copy and constant propagation. It is implemented in the function `update_reg_equiv`.

Actually all hard-registers could be assigned in the global allocator. Such division between the local and global allocator has historical roots. In my opinion it is reasonable to remove the local allocator in the future because faster allocation of local pseudo-registers does not compensate the cost of an additional pass. If all assigning hard-registers is done in the global register allocator (but we still call `update_equiv_regs`), GCC is in average 0.5% faster on SPEC2000 benchmarks on Pentium 4.

**The global allocator** assigns hard-registers to pseudo-registers living in more one basic block. It could change an assignment made by the local allocator if it finds that usage of the hard-register for a global pseudo-register is more profitable than one for the local pseudo-register.

The global allocator forms a bit-vector for each pseudo-register containing hard registers conflicting with the pseudo-registers, builds a conflict graph for pseudo-registers and sorts all pseudo-registers according to the following priority:

$$\frac{\log_2 Nrefs \cdot Freq}{Live\_Length} \cdot Size$$

Here $Nrefs$ is number of the pseudo-register occurrences, $Freq$ is the frequency of its usage, $Live\_Length$ is the length of the pseudo-register's live range in instructions, and $Size$ is its size in hard-registers.

Afterwards the global allocator tries to assign hard-registers to the pseudo-registers

with higher priority first. If the current pseudo-register got a hard-register, the hard-register is added to the hard-register conflict bit-vectors of all pseudo-registers conflicting with the given pseudo-register. This algorithm is very similar to assigning hard-registers in Chow's priority-based colouring [Chow84, Chow90].

The global allocator tries to coalesce pseudo-registers with hard-registers met in a move instruction by assigning the hard-register to the pseudo-register. It is made through a preference technique: the hard-register will be preferred by the pseudo-register if there is a copy instruction with them. In brief, the global allocator is looking for a hard-register to assign to a pseudo-register in the following order:

1. a callee saved hard-register which is in the pseudo-register's preferred class and which is preferred by the pseudo-register while not being preferred by another conflicting pseudo-register.
2. a callee saved hard-register which is in the pseudo-register's preferred class and which is preferred by the pseudo-register.
3. a callee saved hard-register which is in the pseudo-register's preferred class.
4. as in 1-3 but a caller saved hard-register (if it is profitable) instead of callee-saved one.
5. as in 1-4 but the hard-register is in the pseudo-register's alternative class.

**The reload** is a very complicated pass. Its major goal is to transform RTL into a form where all instruction constraints for its operands are satisfied. The pseudo-registers are transformed here into either hard-registers, memory, or constants. The reload pass follows the assignment made by the global and local register allocators. But it can change the assignment if needed.

For example, if the pseudo-register got hard-register *A* in the global allocator but an instruction referring to the pseudo-register requires a hard-register of another class, the reload will generate a move of *A* into the hard-register *B* of the needed classes. Sometimes, a direct move is not possible; we need to use an intermediate hard-register *C* of the third class or even memory. If the hard-registers *B* and *C* are occupied by other pseudo-registers, we expel the pseudo-registers from the hard-registers. The reload will ask the global allocator through function `retry_global` to assign another hard-register to the expelled pseudo-register. If it fails, the expelled pseudo-register will finally be placed on the program stack.

To choose the best register shuffling and load/store memory, the reload uses the costs of moving register of one class into register of another class, loading or storing a register of the given class. To choose the best pseudo-register for expelling, the reload uses the frequency of the pseudo-register's usage.

Besides this major task, the reload also does elimination of virtual hard-registers (like the argument pointer) and real hard-registers (like the frame pointer), assigning stack slots for spilled hard-registers and pseudo-registers which finally have not gotten hard-registers, copy propagation etc.

The complexity of the reload is a consequence of the very powerful model of tar-

get processor's register file, permitting to describe practically any weird processor.

**Postreload.** The reload pass does most of its work in a local scope; it generates redundant moves, loads, stores etc. The postreload pass removes such redundant instructions in basic block context.

## 2 Live Range splitting

Live range splitting is based on idea that if we split the live range of a pseudo-register in several parts, the pseudo-register in each live range part will conflict with fewer other pseudo-registers; less hard-registers will be needed for all the pseudo-registers. Figure 3 illustrates this. Pseudo-register *A* conflicts with two pseudo-registers *B* and *C*, but in part 1 and 2 of its live range the pseudo-register conflicts only with one other pseudo-register.

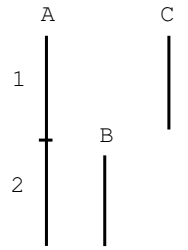



Figure 3: Live range splitting for pseudo-register A.

Live range splitting might require the generation of additional instructions; e.g. instructions storing/loading pseudo-register value into/from memory, moving the pseudo-register into/from a new pseudo-register, or just recalculation of the pseudo-register value. Cost of such additional instructions can outweigh the benefits of reducing the register pressure. So any live range splitting algorithm should take this problem into account.

### 2.1 Register renaming

Register renaming could be considered as no cost live range splitting because no additional instructions need to be generated. We can change a pseudo-register into several ones if there are multiple independent parts of the pseudo-register's usage. The following is a high level example when register renaming could be used.

```
for (i = 0; i < n; i++) { ... }
for (i = 0; i < k; i++) { ... }
```

After register renaming (the pseudo-register renamed is the variable `i`), the corresponding code could look like

```
for (i = 0; i < n; i++) { ... }
for (i_1 = 0; i_1 < k; i_1++) { ... }
```

This optimization was written independently by Jan Hubicka from SUSE and me. Jan's variant is in GCC mainline now. Earlier it was activated by using `-fweb` (independent part of a pseudo-register is traditionally called web in colouring based register allocator). After solving the problem of generating correct debugging information it is default for `-O2` now. Tables 1 and 2 contain SPEC2000 results for Pentium 4 with and without register renaming. Although the results are not impressive for SPECInt2000 (mainly because of perlbmk), this optimization is a "must be" for any optimizing compiler. In most benchmarks it could considerably increase the performance. The results look much better for SPECfp2000. The reduced register pressure means less instructions for spilling and restoring registers and shorter instructions because hard registers instead of memory are used in more instructions. As the result code size for Pentium 4 is 0.3% and 0.6% less in average for SPECint2000 and SPECfp2000 correspondingly.

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 164.gzip | 747 | 750 | +0.40% |
| 175.vpr | 531 | 530 | -0.19% |
| 176.gcc | 891 | 897 | +0.90% |
| 181.mcf | 539 | 539 | +0.00% |
| 186.crafty | 800 | 798 | -0.25% |
| 197.parser | 649 | 648 | -0.15% |
| 252.eon | 663 | 682 | +2.87% |
| 253.perlbmk | 1019 | 939 | -7.85% |
| 254.gap | 831 | 838 | +0.84% |
| 255.vortex | 973 | 961 | -1.23% |
| 256.bzip2 | 621 | 628 | +1.11% |
| 300.twolf | 665 | 671 | +0.90% |
| SPECint2000 | 728 | 726 | -0.27% |

Table 1: SPECint2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with register renaming.

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 168.wupwise | 895 | 898 | +0.33% |
| 171.swim | 617 | 621 | +0.64% |
| 172.mgrid | 598 | 597 | -0.17% |
| 173.applu | 636 | 637 | +0.16% |
| 177.mesa | 654 | 656 | +0.31% |
| 179.art | 245 | 250 | +2.04% |
| 183.equake | 984 | 988 | +0.40% |
| 200.sixtrack | 352 | 406 | +15.34% |
| 301.apsi | 406 | 405 | -0.25% |
| SPECfp2000 | 552 | 563 | +1.99% |

Table 2: SPECfp2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with register renaming.

Register renaming also improves instruction scheduling by removing some anti-dependencies. So it could be useful even for architectures with many hard registers like IA-64. Table 3 contains SPECfp2000 results for Itanium 2 with and without register renaming. The code size for SPECfp2000 was also 0.24% less.

Andrew Macleod from RedHat also implemented this optimization in the transformation of SSA into normal form (this is made very easy on this pass because the independent parts

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 168.wupwise | 383 | 385 | +0.52% |
| 171.swim | 388 | 395 | +1.80% |
| 172.mgrid | 229 | 230 | +0.44% |
| 173.applu | 293 | 297 | +1.37% |
| 177.mesa | 660 | 658 | -0.30% |
| 179.art | 1605 | 1583 | -1.37% |
| 183.equake | 315 | 315 | 0.00% |
| 200.sixtrack | 157 | 161 | +2.55% |
| 301.apsi | 266 | 267 | +0.38% |
| SPECfp2000 | 373 | 375 | +0.53% |

Table 3: SPECfp2000 for Itanium2 GCC with `-O2` without and with register renaming.

of a variable usage are present naturally in SSA). He reported about 2% improvement for SPECint2000 for Pentium 4. When tree-SSA branch becomes GCC mainline, Jan's implementation probably should probably go away because register renaming is made easier and faster during the translation of SSA into normal form.

### 2.2 Live range splitting

The idea of this approach is to store a pseudo-register living through a region but not used in the region right before entering the region and reload its value right after leaving the region. It decreases register pressure in the region by one.

I have implemented practically the same algorithm described in [Morgan98]. Morgan's algorithm works as a separate compiler pass. It starts work on the topmost loops with the register pressure higher than the number of available hard-registers. It searches for pseudo-registers living through the loop but not being used there. It chooses a pseudo-register living through a maximal number of loops (and basic blocks) which are neighbors of the loop being processed. Then it spills the pseudo-register before the loop(s) and restore the pseudo-register after the loop(s). After processing the

loops the algorithm recursively processes sub-loops. When all sub-loops are processed, the algorithm tries to decrease register pressure inside basic blocks. Figure 4 illustrates how the algorithm works.

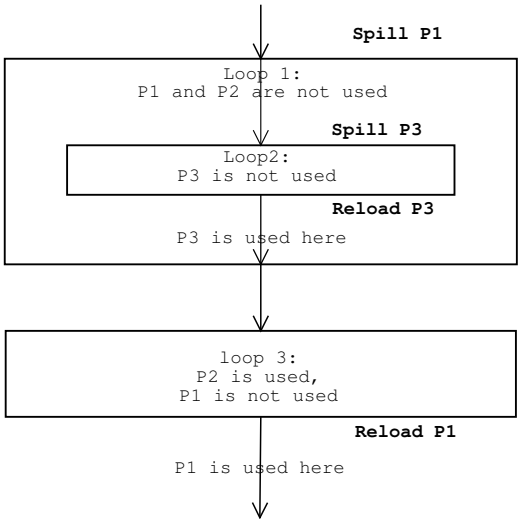



Figure 4: Illustration of Morgan's algorithm of live range splitting.

The current implementation is different from Morgan's in the following:

- Although our implementation also works on loops, it could be easily modified to work on any nested regions instead.
- Instead of spilling the pseudo-register into memory before the loop(s) and reloading it we create a new pseudo-register living only in the loop(s) and inserting instructions shuffling the two pseudo-registers. If both pseudo-registers get memory or hard-registers (it really can happen in the reload pass), the move instructions are coalesced (see the section on coalescing later in this article). If one pseudo-register gets a hard-register and another one gets memory, the move instructions will be transformed into memory store and load instructions.
- GCC has a complicated description model for registers. A hard-register can belong to more one register class. A pseudo-register can get a hard-register from two different classes (see the description of the original register allocator above). To calculate register pressure we consider a pseudo-register belonging to the smallest register class containing the two pseudo-register classes (preferred and alternative ones).
- We do not decrease register pressure inside the basic blocks. We found that on most benchmarks this is not profitable.

The current SPECInt95 results for the optimization usage for Pentium 4 are given in Table 4. The improvement can be even more for some benchmarks. For example, Fast Fourier Transform became 6% faster for Pentium 4 with this optimization, a linear-space Local similarity algorithm [Huang91] became 14% faster, and fftbench [Ladd03] became more 30% faster.

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 099.go | 68.6 | 67.8 | -1.17% |
| 124.m88ksim | 72.3 | 71.8 | -0.69% |
| 126.gcc | 75.2 | 74.8 | -0.53% |
| 129.compress | 55.5 | 56.4 | +1.62% |
| 130.li | 78.3 | 78.0 | -0.38% |
| 132.ijpeg | 72.5 | 72.6 | +0.14% |
| 134.perl | 68.5 | 79.8 | +16.50% |
| 147.vortex | 68.1 | 68.6 | +0.73% |
| SPECint95 | 69.6 | 70.9 | +1.87% |

Table 4: SPECInt95 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with live range splitting.

I see the following possible improvements to the implementation:

- Better utilization of profile information to choose loops with many iterations.

- Forming regions based on the profile information different from the loops for the algorithm of live range splitting.
- Choosing pseudo-registers whose live range splitting does not result in critical edge splitting. As a consequence, no additional branch instructions will be generated. It could be important for live range splitting around loops with few iterations.
- More accurate evaluation of register pressure for register classes to which a living pseudo-register belongs.

### 2.3 Rematerialization

Instead of reloading a pseudo-register's value we could just recalculate it again if it is more profitable. Such approach is called register rematerialization. Preston Briggs believed that it is a more promising approach than live range splitting. It requires that all the pseudo registers used as operands are live and got hard registers because otherwise we will need to reload the operand value too. Reloading the operand's value usually costs the same as reloading the pseudo-register in question.

My current implementation of the register rematerialization works between global register allocation and reload passes. To rematerialize a pseudo-register we insert an existing instruction setting up the pseudo-register's value. To know what instructions could be inserted we define the partial availability of instruction patterns according to the following equations.

$$P\_PavIn_i = \bigcup_{j \in Pred(i)} P\_PavOut_j$$
$$P\_PavOut_i = (P\_PavIn_i - P\_Kill_i) \bigcup P\_Gen_i$$

Here $P\_Kill_i$ is a set of patterns whose defined and used locations (registers or memory) are redefined or clobbered in basic block $i$ or whose clobbered registers are live at the end of basic block. $P\_Gen_i$ is a set of patterns in basic block whose defined or used locations are not killed in the basic block after the pattern's occurrence and whose clobbered registers are not live at the end of the basic block.

After we calculated partial availability of patterns, we use it as an initial value to calculate availability of patterns according to the following equation.

$$P\_AvIn_i = \bigcap_{j \in Pred(i)} P\_AvOut_j$$
$$P\_AvOut_i = (P\_AvIn_i - P\_Kill_i) \bigcup P\_Gen_i$$

The algorithm itself looks like

```
foreach insn I defining the
        only pseudo-register D do
  if D got a hard-register then
    foreach pseudo-register operand Op
            of I do
      if Op got memory then
         Pat := a pattern with a minimal
                cost available right before
                I and whose the only
                destination pseudo-register
                is Op and whose all other
                operand pseudo-registers
                got hard-registers;
        if there is Pat and its cost is less
           than cost of loading Op then
          insert insn before I with pattern
              Pat changing Op on D;
          change Op in I on D;
          break;
        fi
      fi
    done
  fi
done
```

e.g. if pseudo-register *A* got memory and pseudo-registers *B*, *C* and *D* got hard-registers, the algorithm will work as follows

```
A <- op1 (B, C)          ...
...                  ->  ...
D <- op2 (A, E)          D <- op1 (B, C)
                         D <- op2 (D, E)
```

If the second instruction in the example is

move, the algorithm together with the dead code elimination will work as

```
A <- op1 (B, C)          ...
...                  ->  ...
D <- A                   D <- op1 (B, C)
```

Table 5 contains results of the optimization for SPECint2000 for Pentium 4.

| Benchmark | Base | Peak | Change |
|---|---|---|---|
| 164.gzip | 838 | 839 | +0.12% |
| 175.vpr | 602 | 598 | -0.66% |
| 176.gcc | 1137 | 1146 | +0.79% |
| 181.mcf | 715 | 715 | 0.00% |
| 186.crafty | 874 | 875 | +0.11% |
| 197.parser | 734 | 734 | 0.00% |
| 252.eon | 764 | 763 | -0.13% |
| 253.perlbmk | 1145 | 1164 | +1.66% |
| 254.gap | 954 | 951 | -0.31% |
| 255.vortex | 1079 | 1080 | +0.09% |
| 256.bzip2 | 743 | 745 | +0.27% |
| 300.twolf | 757 | 767 | +1.32% |
| Est. SPECint2000 | 845 | 848 | +0.36% |

Table 5: SPECint2000 for Pentium 4 with `-O2 -mtune=pentium4` without and with register rematerialization.

Register rematerialization could be done in a separate pass before the register allocation [Simpson96]. In brief, Simpson's algorithm looks like

```
foreach basic block BB do
  while the register pressure is too high
        in BB do
    P := a pattern available and live at
         the end of BB with a result
         pseudo-register is not used in BB
         and its operands are live
         at the end of BB;
    if there is no such P then
      break;
    fi
    put insns with pattern P on edges
        exiting from BB where P are live;
    move the insns to the bottom of CFG as
         far as possible along the paths
         where P is still available, and P
         and its operands are live;
    update the register pressure in BB and
           basic blocks we moved the insns
           through;
  done
done
```

The liveness of a pattern in a CFG point means that a result register of the pattern is used in another point achieved from the given point. Figure 5 illustrates how the algorithm works.

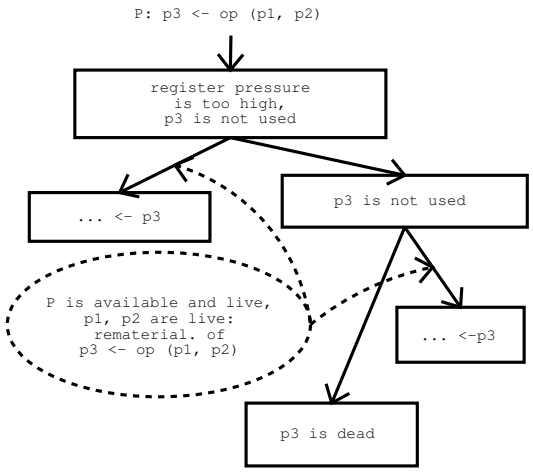



Figure 5: Illustration of Simpson's algorithm of rematerialization.

I have implemented Simpson's approach in GCC. It gave about 1.3% improvement for SPECint2000 for Pentium 4 on the tree-ssa branch. And after deciding to implement Morgan's live range splitting, I rejected Simpson's implementation because I believe that Mor-

gan's live range splitting together with register rematerialization after global register allocation will work better. I see the following reasons for this:

- It is difficult to know which operand pseudo-registers will get hard-registers in the end. Adding instruction rematerializing pseudo-register's value might result in generation of additional load instructions in the reload pass if the operand pseudo-registers do not get hard-registers.

- Morgan's approach to live range splitting works in more cases than Simpson's. The instructions shuffling pseudo-registers generated in Morgan's algorithm are removed by coalescing and, if it is not possible, rematerialized.

- Rematerialization could be done in more cases. The single criterion is a profitability not just high register pressure as in Simpson's approach.

# 3 Live range shrinking

The live range shrinking approach is to move the definitions of pseudo-register as close as possible to their usages. It decreases the number of conflicts for the pseudo-register and consequently may decrease register pressure. There are few articles devoted this approach (one of them is [Balakrishnan01]). The reason for this is in its constraints for modern pipelined processors. Solving this problem without taking instruction scheduling into account could worsen code in many cases. So live range shrinking mainly became a part of register pressure sensitive instruction schedulers.

## 3.1 Register Pressure Sensitive Instruction Scheduling

GCC uses a classical two pass instruction scheduling approach: instruction scheduling both before and after the register allocator. It works well for RISC processors with a large enough number of registers.

For processors with few registers, however, instruction scheduling before register allocation creates such high pressure that it actually worsens the code. Therefore it is switched off for x86 and sh4.

One year ago Dale Johannesen from Apple added a new heuristic right after the critical path length heuristic. This heuristic prefers instructions with smaller contribution to register pressure. He reported about 2% improvement for SPECint2000 for PowerPC.

Sanjiv Gupta implemented machine-dependent register pressure-sensitive instruction scheduling for SH4. He reported a big improvement for some benchmarks (Table 6) when the first instruction scheduler with the register-pressure heuristic was switched on. Unfortunately, he did not compare instruction scheduling with and without the heuristic (probably the results would be even better because earlier the first instruction scheduling pass without any register-pressure heuristic was switched off).

Sanjiv's implementation is very similar to the Hsu and Goodman approach [GooHsu88] to register pressure sensitive instruction scheduling: when the pressure becomes high, it uses register pressure heuristic as major one instead of the critical path length heuristic. I have implemented Hsu's approach in a machine independent way. My goal was to improve x86 code by switching on the first instruction scheduling pass. Although GCC with the register pressure sensitive approach in the first pass generated a more 1% better code for

| Benchmark | Base | Peak | Change |
|---|---|---|---|
| Gsm compression | 31.83 | 26.16 | +17% |
| Gsm decompression | 17.72 | 16.94 | +4.4% |
| cjpeg -dct int | 2.30 | 2.34 | -1.7% |
| cjpeg -dct float | 2.12 | 2.19 | -3% |
| djpeg -dct int | 1.53 | 1.45 | +5% |
| djpeg -dct float | 1.69 | 1.42 | +15% |
| gzip | 225 | 222 | +1% |
| gunzip | 17.30 | 16.69 | +3.5% |
| Mpg123 | 1.29 | 1.26 | +2% |

Table 6: Benchmarks for SH4 GCC with `-O2` without the 1st instruction scheduling and with the 1st register pressure sensitive instruction scheduling.

SPECfp95 than with the standard first pass, the results are disappointing in comparison with GCC without any first instruction scheduling. Table 7 contains SPEC95 results for the programs compiled without the first instruction scheduling pass (default in GCC for x86) and with Hsu's approach in the first instruction scheduler. I used Athlon MP because GCC still has no pipeline description for Pentium 4.

The most interesting result is for *fpppp*: the code became practically 3 times slower (SPECfp95 results would be very close without *fpppp*). The hot point of *fpppp* is the function with one huge basic block. The register pressure reaches several hundred there for x86 GCC. It looks to me like the basic block was optimized manually to minimize the register pressure. Any rearrangement of the instructions results in a higher register pressure, especially for x87 floating point top stack register. So in my opinion, to make a successful register pressure sensitive instruction scheduler for x86, we need a more sophisticated approach than Hsu's on-the-fly approach. These approaches should be based on the evaluation of all data flow graphs like a parallel interference graph [Norris93] or a register reuse graph [Berson98].

| Benchmarks | Base | Peak | Change |
|---|---|---|---|
| 099.go | 68.9 | 68.2 | -0.73% |
| 124.m88ksim | 52.8 | 51.8 | -1.89% |
| 126.gcc | 57.5 | 57.1 | -0.70% |
| 129.compress | 28.0 | 27.9 | -0.36% |
| 130.li | 58.9 | 59.0 | +0.17% |
| 132.ijpeg | 53.1 | 50.6 | -2.82% |
| 134.perl | 79.2 | 76.3 | -3.66% |
| 147.vortex | 50.3 | 50.5 | +0.40% |
| SPECint95 | 54.0 | 53.3 | -1.30% |
| 101.tomcatv | 74.1 | 75.7 | +2.16% |
| 102.swim | 139 | 139 | 0.00% |
| 103.su2cor | 22.4 | 21.8 | -2.68% |
| 104.hydro2d | 24.5 | 24.3 | -0.82% |
| 107.mgrid | 47.7 | 50.6 | +6.08% |
| 110.applu | 28.2 | 27.4 | -2.84% |
| 125.turb3d | 53.2 | 51.4 | -3.38% |
| 141.apsi | 32.5 | 33.3 | +2.46% |
| 145.fpppp | 148 | 54.0 | -63.51% |
| 146.wave5 | 77.3 | 73.7 | -4.66% |
| SPECfp95 | 52.2 | 47.0 | -9.96% |

Table 7: SPEC95 results for Athlon MP with `-O2 -mtune=athlon` without the first instruction scheduler and with Hsu's register pressure sensitive first instruction scheduling.

# 4 Other improvements of the GCC original register allocator

## 4.1 Coalescing

Live range splitting tends to create unnecessary move instructions. As I mentioned above, we generate additional pseudo-registers and instructions shuffling them instead of the traditional approach generating instructions spilling registers to memory and restoring them. Even if the live range splitting optimization is not run, there are still unnecessary move instruction generated by the previous optimizations. To remove them, pseudo-register coalescing is run after the global register allocator. If the pseudo-registers in a move instruction do not conflict we could use one pseudo-register and remove the move instructions. It is done if both pseudo-registers got hard registers or

both pseudo-registers were placed in memory (it means that the move would have been transformed into instructions moving the memory). The following example describes the two situations (the number in the parentheses is the hard register number given to the pseudo-register):

```
p256 (1) <- p128 (2)
     or
p256 (Memory) <- p128 (Memory)
```

Sometimes, removing a pseudo-register move instruction when one pseudo-register gets a hard-register in the global register allocator and another one gets memory could be profitable too. The resulting pseudo-register will be placed in memory after coalescing the two pseudo-registers. Profitability is defined by the execution frequency of the move instruction and the reference frequency of the pseudo-register which got a hard-register. A typical situation when it is profitable is given on figure 6. The pseudo-register *p128* got the hard register number 2 and *p256* was placed in memory.

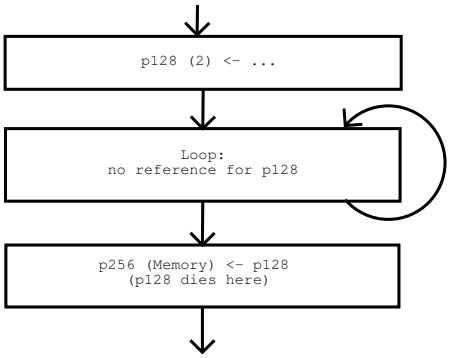



Figure 6: Coalescing memory and register.

Even if there is no move instruction between two pseudo-registers which are placed in memory (usually on the program stack), we can coalesce them. What is the sense of such an optimization? Although the optimization does not remove instructions, it decreases the size of the used stack (it is very important for the Linux kernel which usually has strict constraints for the size of the program stack). For example, the average decrease of function stack frames is about 4% with this optimization for Linpack x86 code. The optimization also improves data locality and code locality for some architectures like x86 because in many cases smaller displacements in instruction are used (we are using the first found stack slot approach). Table 8 shows the text segment's size decrease for the SPECfp2000 benchmarks for Pentium 4. The improved code and data locality considerably improves the code. Table 9 shows the SPECfp2000 performance results for code without and with the optimization for Pentium 4.

| Benchmarks | Base | Peak | Change |
|---|---|---|---|
| 168.wupwise | 25128 | 24648 | -1.910% |
| 171.swim | 7078 | 7014 | -0.904% |
| 173.applu | 58741 | 58453 | -0.490% |
| 177.mesa | 443993 | 439369 | -1.041% |
| 179.art | 12011 | 12011 | 0.000% |
| 183.equake | 17026 | 17026 | 0.000% |
| 200.sixtrack | 844452 | 815060 | -3.481% |
| 301.apsi | 106317 | 103341 | -2.799% |
| Average | | | -1.33% |

Table 8: SPECfp2000 benchmark code sizes for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with coalescing the program stack slots.

The patch improves code and data locality, therefore GCC becomes a bit faster. User time for x86 bootstrapping decreased from 14m0.150s to 13m58.890s. The better code and data locality improves SPECFP2000 benchmark results too (about 2.4%).

### 4.2 Register migration

When the reload pass needs a hard register for a reload, it expels a living pseudo-register from the hard register assigned to it by the local or global register allocator. Then it tries to reas-

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 168.wupwise | 890 | 887 | -0.34% |
| 171.swim | 604 | 609 | +0.83% |
| 173.applu | 624 | 627 | +0.48% |
| 177.mesa | 629 | 639 | +1.59% |
| 179.art | 244 | 248 | +1.64% |
| 183.equake | 964 | 963 | -0.01% |
| 200.sixtrack | 337 | 385 | +14.24% |
| 301.apsi | 401 | 407 | +1.97% |
| SPECint2000 | 388 | 399 | +2.83% |

Table 9: SPECfp2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with coalescing the program stack slots.

sign a free hard register to the pseudo-register (function `retry_global_alloc`). Usually it fails especially when the processor has few registers or there is a high register pressure in the function. So finally the pseudo-register is placed in memory. Figure 7 shows an example of such a situation (the pseudo-register *p128* is expelled from hard register *A* because it is needed for an instruction which is in the live range of *p128*).

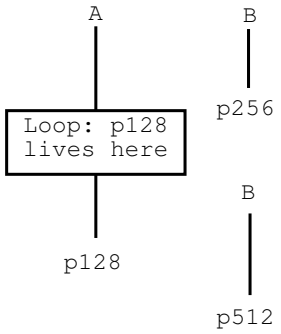



Figure 7: Case for the register migration.

Sometimes it is more profitable to use another hard register (*B* in the example) instead of memory for the pseudo-register. It might be possible by expelling another rarely used pseudo-register (*p256* and *p512* in the example) from their hard registers. In their own turn the expelled pseudo-registers can also migrate.

The optimization works well with processors with irregular register files (which means generation of more reloads because of strict instruction constraints for input/output registers).

Tables 10 and 11 contain SPEC2000 results for Pentium 4 for benchmarks whose codes are different when the optimization is used. We see that the code is smaller and the results are better. Practically the single important degradation is perlbmk (but it can be fixed by the register rematerialization and live range splitting mentioned above). Significant improvement for GCC is more important than perlbmk degradation because it is more difficult to improve GCC than perlbmk; 50% of all time of perlbmk is spent in one very specific function. It is regular expression matching. The SPEC95 perlbmk was a more fare benchmark because it tested the interpreter itself, not regular expression matching.

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 175.vpr | 594 | 596 | +0.34% |
| 176.gcc | 1123 | 1133 | +0.89% |
| 186.crafty | 869 | 877 | +0.92% |
| 197.parser | 730 | 729 | -0.14% |
| 252.eon | 765 | 764 | -0.13% |
| 253.perlbmk | 1159 | 1133 | -2.24% |
| 254.gap | 943 | 944 | +0.11% |
| 255.vortex | 1052 | 1056 | +0.38% |
| 256.bzip2 | 737 | 735 | -0.27% |
| 300.twolf | 753 | 763 | +1.33% |
| 173.applu | 771 | 772 | +0.13% |
| 177.mesa | 720 | 726 | +0.83% |
| 200.sixtrack | 394 | 392 | -0.51% |
| 301.apsi | 486 | 489 | +0.62% |

Table 10: SPEC2000 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the register migration.

This optimization makes GCC a bit faster too (the compiler bootstrap test on Pentium 4 is 0.13% faster with the optimization). As for architectures with more regular register files, I found that three SPECfp95 test codes for

| Benchmarks | Base | Peak | Change |
|---|---|---|---|
| 175.vpr | 128917 | 128949 | 0.025% |
| 176.gcc | 1241720 | 1241440 | -0.022% |
| 186.crafty | 204846 | 204878 | 0.016% |
| 197.parser | 85436 | 85420 | -0.019% |
| 252.eon | 480338 | 480354 | 0.003% |
| 253.perlbmk | 473971 | 473667 | -0.064% |
| 254.gap | 421816 | 421592 | -0.053% |
| 255.vortex | 568904 | 569128 | 0.039% |
| 256.bzip2 | 28133 | 28117 | -0.057% |
| 300.twolf | 181055 | 181055 | 0.000% |
| Average | | | -0.013% |
| 173.applu | 58741 | 58741 | 0.000% |
| 177.mesa | 443993 | 443049 | -0.213% |
| 200.sixtrack | 844452 | 843892 | -0.066% |
| 301.apsi | 106317 | 106317 | 0.000% |
| Average | | | -0.070% |

Table 11: SPEC2000 benchmark code sizes for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the register migration.

PowerPC were different (applu, turb3d, and wave5). Test applu was sped up about 1% (two others had the same result).

### 4.3 More accurate information about register conflicts

The original register allocator used standard live information to build a conflict graph. This live information is based on the most widely used definition of pseudo-register liveness: Register *R* lives at point *p* if there is a path from *p* to some use of *R* along which *R* is not redefined. The live information is described by the following data flow equations:

$$LiveIn_i = (LiveOut_i - Def_i) \bigcup Use_i$$
$$LiveOut_i = \bigcup_{j \in Succ(i)} LiveIn_j$$

$LiveIn_i$ and $LiveOut_i$ are sets of registers correspondingly living at the start and at the end of basic block $i$. $Use_i$ is the set of registers used in basic block $i$ and not redefined after the usage in the basic block. $Def_i$ is the set of registers defined or clobbered in basic block $i$.

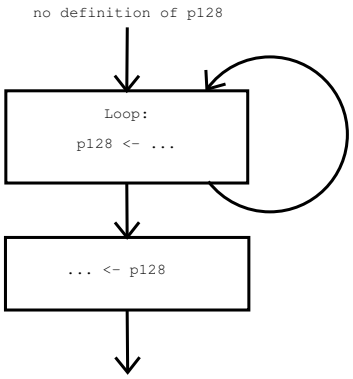



Figure 8: A typical case when accurate life information is different from the standard one.

This information is actually inaccurate because according to it a pseudo-register may live before the first assignment to it. Figure 8 demonstrates such situation. The first assignment to pseudo-register *p128* happens in the loop. According to GCC life analysis, *p128* will live in any basic block where there is a path from the basic block to the loop. Such inaccurate live information results in bigger evaluated register pressure and worse register allocation because *p128* conflicts with all pseudo-registers in the basic blocks preceding the loop.

To make the live information more accurate (RealLive sets) in building conflict graphs we could use the partial availability according to the following equations:

$$PavIn_i = \bigcup_{j \in Pred(i)} PavOut_j$$
$$PavOut_i = (PavIn_i - Kill_i) \bigcup Gen_i$$

$$RealLiveIn_i = LiveIn_i \bigcap PavIn_i$$
$$RealLiveOut_i = LiveOut_i \bigcap PavOut_i$$

$PavIn_i$ and $PavOut_i$ are sets of registers correspondingly partially available at the start and at the end of basic block $i$. $Kill_i$ is the set of registers killed (clobbered) in basic block

$i$. $Gen_i$ is the set of registers defined in basic block $i$ and not killed after their definition in the basic block.

It seems that there are few cases where Real-Live and Live sets are different. In reality there are a lot of benchmarks whose code is different when the accurate live information is used. Tables 12 and 13 contains SPEC95 results for tests which have a different code when more accurate information is used.

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 126.gcc | 80.8 | 81.4 | +0.74% |
| 130.li | 86.4 | 86.6 | +0.23% |
| 132.ijpeg | 79.5 | 80.0 | +0.63% |
| 134.perl | 86.8 | 87.9 | +1.27% |
| 141.apsi | 57.6 | 58.0 | +0.69% |
| 146.wave5 | 95.6 | 95.8 | +0.21% |

Table 12: SPEC95 for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the accurate life information.

| Benchmarks | Base | Peak | Change |
|---|---|---|---|
| 126.gcc | 1102160 | 1101830 | -0.030% |
| 130.li | 44047 | 44031 | -0.036% |
| 132.ijpeg | 120904 | 120808 | -0.079% |
| 134.perl | 233331 | 233315 | -0.007% |
| 141.apsi | 103221 | 103205 | -0.016% |
| 146.wave5 | 96668 | 96668 | 0.000% |
| Average | | | -0.028% |

Table 13: SPEC95 benchmark code sizes for Pentium 4 GCC with `-O2 -mtune=pentium4` without and with the accurate life information.

Another way to decrease the number of conflicts and as a consequence improve the register allocation is to consider the values of pseudo-registers. Pseudo-registers may get the same hard-registers if they hold the same value in every point where they live simultaneously. Global value numbering [Simpson96] could be used for this. I have tried a simplified version of GVN where all operators except copies are different. I believed that most cases belong to this category. GVN even in such form is still an expensive optimization and a bit complicated because reaching definitions [Muchnick97] have to be used for this (usually GVN is fulfilled in SSA). There are few tests where GVN results in different code (e.g. *eon* and *perlbmk* SPECint2000 tests for x86. Eon had the same performance, perlbmk was about 0.2% faster). So I think the usage of such optimization in GCC is not reasonable.

### 4.4 Better utilization of profiling information

The original register allocator mainly utilizes profiling information in its work. But there are some instances where it is not true. One such place is the calculation of profitability of usage of caller-saved hard registers for pseudo-registers crossing function calls. Currently it is based on number of the crossed calls and number of the pseudo-register usages. Usage of the frequencies of the crossed calls and the pseudo-register usages instead of the numbers can improve the generated code especially when the execution profile is used. Tables 14 and 15 contain SPECfp2000 results for Pentium 4 when the profile is used.

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 168.wupwise | 996 | 1006 | +1.00% |
| 171.swim | 921 | 928 | +0.75% |
| 172.mgrid | 702 | 703 | +0.14% |
| 173.applu | 766 | 771 | +0.65% |
| 177.mesa | 734 | 739 | +0.68% |
| 179.art | 381 | 384 | +0.78% |
| 183.equake | 1217 | 1226 | +0.74% |
| 200.sixtrack | 454 | 456 | +0.44% |
| 301.apsi | 450 | 479 | +6.44% |
| SPECfp2000 | 688 | 696 | +1.16% |

Table 14: SPECfp2000 for Pentium 4 GCC with `-O2` without and with caller-saved register profitability based on frequency. The profile information is used.

| Benchmarks | Base | Peak | Change |
|---|---|---|---|
| 168.wupwise | 25384 | 25320 | -0.252% |
| 171.swim | 7174 | 7174 | 0.000% |
| 172.mgrid | 10015 | 10111 | 0.959% |
| 173.applu | 59405 | 59509 | 0.175% |
| 177.mesa | 433609 | 434105 | 0.114% |
| 183.equake | 16386 | 16418 | 0.195% |
| 179.art | 12123 | 12235 | 0.924% |
| 200.sixtrack | 835724 | 838972 | 0.389% |
| 301.apsi | 104573 | 104837 | 0.252% |
| Average | | | 0.31% |

Table 15: SPECfp2000 benchmark code sizes for Pentium 4 GCC with `-O2` without and with caller-saved register profitability based on frequency. The profile information is used.

The results could be better even without the profile information. Tables 16 and 17 contain analogous results without the profile for Athlon.

| Benchmarks | Base ratio | Peak ratio | Change |
|---|---|---|---|
| 168.wupwise | 533 | 551 | +3.38% |
| 171.swim | 428 | 441 | +3.03% |
| 172.mgrid | 404 | 404 | 0.0% |
| 173.applu | 344 | 341 | -0.87% |
| 177.mesa | 623 | 632 | +1.44% |
| 179.art | 165 | 163 | -1.21% |
| 183.equake | 404 | 403 | -0.25% |
| 200.sixtrack | 369 | 368 | -0.27% |
| 301.apsi | 282 | 287 | +1.77% |
| SPECint2000 | 372 | 375 | +0.81% |

Table 16: SPECfp2000 for Athlon GCC with `-O2 -mtune=athlon` without and with caller-saved register profitability based on frequency. Profile information is not used.

### 4.5 Global common subexpression elimination

As I wrote, the post-reload pass of the original register allocator removes redundant instructions (mostly loads and stores) generated by the reload pass. It uses the CSE (common subexpression elimination) library for this. This

| Benchmarks | Base | Peak | Change |
|---|---|---|---|
| 168.wupwise | 24872 | 24792 | -0.322% |
| 171.swim | 7142 | 7142 | 0.000% |
| 172.mgrid | 9791 | 9807 | 0.163% |
| 173.applu | 58197 | 58317 | 0.206% |
| 177.mesa | 456005 | 458773 | 0.607% |
| 179.art | 13254 | 13494 | 1.811% |
| 183.equake | 16724 | 16788 | 0.383% |
| 200.sixtrack | 830268 | 831468 | 0.145% |
| 301.apsi | 103981 | 103773 | -0.200% |
| Average | | | 0.31% |

Table 17: SPECfp2000 benchmark code sizes for Athlon GCC with `-O2 -mtune=athlon` without and with caller-saved register profitability based on frequency. Profile information is not used.

permits to remove redundancy only in basic blocks.

I was going to implement global redundancy elimination as the next logical step. Fortunately, it was already done independently by Mostafa Hagog from IBM. For PowerPC G5 he reported 1.4% improvement for SPECint2000 (with stunning 15% improvement for perlbmk) and 0.5% degradation for SPECfp2000 (see table 18).

## 5 Conclusions

As I wrote, the priority-based colouring register allocator can compete with the Chaitin/Briggs register allocators. Therefore I believe we should work on the original register allocator as much as on the new register allocator. It is good to have two register allocators to choose the better one, depending on architecture used.

There are a lot of ways to improve the original register allocator's code. The most interesting one is live range splitting integrated with the register allocator. This is the single important part which is missed in the original GCC

| Benchmarks | Base | Peak | The improvement |
|---|---|---|---|
| 164.gzip | 775 | 803 | 3.6% |
| 175.vpr | 513 | 504 | -1.8% |
| 181.mcf | 500 | 500 | 0.0% |
| 186.crafty | 868 | 872 | 0.5% |
| 197.parser | 679 | 681 | 0.3% |
| 252.eon | 828 | 819 | -1.1% |
| 253.perlbmk | 730 | 844 | 15.6% |
| 254.gap | 811 | 790 | -2.6% |
| 255.vortex | 952 | 964 | 1.3% |
| 256.bzip2 | 619 | 622 | 0.5% |
| 300.twolf | 605 | 606 | 0.2% |
| Est. SPECint | 702.2 | 712.0 | 1.4% |
| 168.wupwise | 895 | 895 | 0.0% |
| 171.swim | 249 | 249 | 0.0% |
| 172.mgrid | 643 | 643 | 0.0% |
| 173.applu | 647 | 660 | 2.0% |
| 177.mesa | 904 | 905 | 0.1% |
| 178.galgel | 696 | 697 | 0.1% |
| 179.art | 624 | 590 | -5.4% |
| 183.equake | 996 | 994 | -0.2% |
| 187.facerec | 1142 | 1143 | 0.1% |
| 188.ammp | 398 | 398 | 0.0% |
| 189.lucas | 530 | 530 | 0.0% |
| 191.fma3d | 970 | 969 | -0.1% |
| 200.sixtrack | 578 | 562 | -2.8% |
| 301.apsi | 554 | 554 | 0.0% |
| Est. SPECfp | 656 | 653 | -0.5% |

Table 18: SPEC2000 results for PowerPC G5 GCC with `-O3` without and with postreload global redundancy elimination.

register allocator from Chow's algorithm. In comparison with the Chaitin/Briggs approach, the priority-based colouring register allocator has an advantage, which is easier implementation of good live range splitting based on register allocation information. It will probably require closer integration of the reload pass and the global register allocator.

## 6 Acknowledgments

I would like to thank Richard Stallman and Earl Killian for answering my questions about GCC's history and the Pastel compiler.

I am grateful to my company, RedHat, for the attention to improving GCC and for permitting me to work on this project. I would like to thank my colleague Andrew MacLeod for providing interesting ideas and his reach experience in register allocation.

Last but not least, I would like to thank my son, Serguei, for the help in proofreading the article.

## References

[Appel96] L. George and A. Appel, *Iterated Register Coalescing*, ACM TOPLAS, Vol. 18, No. 3, pages 300-324, May, 1996.

[Balakrishnan01] Saisanthosh Balakrishnan and Vinod Ganapathy, *Splitting and Shrinking Live Ranges*, CS 701, Project 4, Fall 2001. The University of Wisconsin. (`http://www.cs.wisc.edu/~saisanth/papers/liverange.pdf`).

[Berson98] D. Berson, R. Gupta, and M. Soffa, *Integrated Instruction Scheduling and Register Allocation Techniques*, Languages and Compilers for Parallel Computing, pages 247-262, 1998.

[Briggs89] Articles of Preston Briggs in compiler newsgroup, Nov. 1989.

[Briggs94] P. Briggs, K. D. Cooper, and L. Torczon. *Improvements to graph coloring register allocation*, ACM TOPLAS, Vol. 16, No. 3, pages 428-455, May 1994.

[Chaitin81] G. J. Chaitin, et. al., *Register allocation via coloring*, Computer Languages, 6:47-57, Jan. 1981.

[Chow84] F. Chow and J. Henessy, *Register allocation by priority-based coloring*, In Proceedings of the ACM SIGPLAN

84 Symposium on Compiler Construction (Montreal, June 1984), ACM, New York, 1984, pages 222-232.

[Chow90] F. Chow and J. Hennessy. *The Priority-based Coloring Approach to Register Allocation*, TOPLAS, Vol. 12, No. 4, 1990, pages 501-536.

[GooHsu88] J. R. Goodman and W. C. Hsu, *Code Scheduling and Register Allocation in Large Basic Blocks*, In Proc. of the 2nd International Conference on Supercomputing, pages 442-452, 1988.

[Huang91] Xiaoqiu Huang and Webb Miller, *A Time-Efficient, Linear-Space Local Similarity Algorithm*, Adv. Appl. Math. 12 (1991), 337–357.

[Killian04] Private communications with Earl Killian, March 2004.

[Ladd03] S. R. Ladd, ACOVEA. `http://www.coyotegulch.com`

[Matz03] M. Matz, *Design and Implementation of a Graph Coloring Register Allocator for GCC*, GCC Summit, 2003.

[Morgan98] Robert Morgan, *Building an Optimizing Compiler*, Digital Press, ISBN 1-55558-179-X.

[Muchnick97] Steven S. Muchnick, *Advanced compiler design implementation*, Academic Press (1995), ISBN 1-55860-320-4.

[Norris93] C. Norris and L. Pollock, *A Scheduler-Sensitive Global Register Allocator*, Proceedings of Supercomputing, Portland, Oregon, November 1993.

[Open64] `http://open64.sourceforge.net`.

[ORC] `http://ipf-orc.sourceforge.net`.

[Pro64] `http://oss.sgi.com/projects/Pro64`.

[Simpson96] L. T. Simpson, *Value-driven redundancy elimination*, Ph.D. thesis, Computer Science Department, Rice University.

[Sorkin96] A. Sorkin, *Some Comments on 'The Priority-Based Coloring Approach to Register Allocation'*, ACM SIGPLAN Notices, Vol. 31, No. 7, July 1996.

[Stallman04] Private email from Richard Stallman, March 2004.

# Autovectorization in GCC

*Dorit Naishlos*
IBM Research Lab in Haifa
dorit@il.ibm.com

## Abstract

Vectorization is an optimization technique that has traditionally targeted vector processors. The importance of this optimization has increased in recent years with the introduction of SIMD (single instruction multiple data) extensions to general purpose processors, and with the growing significance of applications that can benefit from this functionality. With the adoption of the new Tree SSA optimization framework, GCC is ready to take on the challenge of automatic vectorization. In this paper we describe the design and implementation of a loop-based vectorizer in GCC. We discuss the new issues that arise when vectorizing for SIMD extensions as opposed to traditional vectorization. We also present preliminary results and future work.

## 1 Introduction

Vector machines were introduced in the 1970's, to increase processor utilization by accelerating the initiation of operations, and keeping the instruction pipeline full. To take advantage of vector hardware, programs are rewritten using explicit vector operations on whole arrays (as in Figure 1b) instead of operations on individual array elements one after the other (as in Figure 1a). This rewrite of loops into vector form is referred to as *vectorization* [3].

The vector notation in Figure 1b implies that all the loads (from arrays a and b) take place before all the stores (into array a); This means that loops like the one in Figure 1d, where each iteration uses a result from a previous iteration, cannot be rewritten in vector form. This situation is an example of a data-dependence that is carried across the iterations of the loop. When no such dependences between loop iterations exist, operations from different iterations can be initiated in parallel, and vectorization may be applied. Data dependence analysis is therefore a fundamental step in the process of vectorization.

Vectorization, when applied automatically by a compiler, is referred to as *autovectorization*. In this paper, we use the two terms interchangeably to refer to compiler vectorization.

In recent years, a different architectural approach to exploit a similar kind of data parallelism has become increasingly common. It follows the Single Instruction Multiple Data (SIMD) model, in which the same instruction simultaneously executes on multiple data elements that are packed in advance in vector registers. The length of these vectors (*vector length*) is relatively small. The number of data elements that they can accommodate determines the degree of parallelism (*vectorization factor*, VF) that can be exploited. This value varies depending on the data-type of the elements.

Vectorizing the loop in Figure 1a for SIMD therefore implies transforming it to operate on VF elements at a time, as illustrated in Fig-

(a) original serial loop:

```
for(i=0; i<N; i++){
  a[i] = a[i] + b[i];
}
```

(b) loop in vector notation:

```
a[0:N] = a[0:N] + b[0:N];
```

(c) vectorized loop:

```
for (i=0; i<(N-N%VF); i+=VF){
  a[i:i+VF] = a[i:i+VF] + b[i:i+VF];
}
for ( ; i < N; i++) {
  a[i] = a[i] + b[i];
}
```

(d) unvectorizable loop (dependence cycle):

```
for (i=1; i<N; i++){
  a[i] = a[i-1] + b[i];
}
```

Figure 1: The vectorization transformation

ure 1c. This is generally equivalent to strip-mining the loops by a factor `VF`, while replacing scalar operations with equivalent vector operations. A serial loop that computes the remaining `N%VF` iterations is also added for the case that `N` does not evenly divide by `VF`.

Applications in many domains have an abundant amount of natural parallelism present in the computations they perform. If this parallelism can be leveraged to exploit the SIMD/vector capabilities of architectures, the performance of these applications can be considerably increased.

The GCC vectorizer implements a loop-based vectorization approach, which means that it focuses on exploiting the data parallelism present across loop iterations. Data parallelism present in straight-line code is not leveraged by the loop-based vectorizer. Vectorization techniques that exploit this type of parallelism, such as [12], could be used as a complementary approach to loop-based vectorization. We briefly discuss this in Section 8.

As we show in the following sections, practical vectorization for vector processors, and in particular the more recent SIMD processors, involves much more than loop dependence analysis and introduces some nontrivial issues and choices especially for a multi-platform compiler like GCC.

## 2 Classic Vectorization vs. SIMD Vectorization

Autovectorization is a mature research area; automatic detection of vector loops in serial code has been discussed in literature for more than a decade [1, 20]. The main focus of classic vectorization is the theory of data dependences. It deals with loop analyses to (1) detect statements that could be executed in parallel without violating the semantics of the program, and (2) increase such occurrences by means of loop transformations.

The classic (data-dependence based) vectorization approach has traditionally targeted the vector machines of the 1970's. Two main developments in recent years have shifted the focus to vectorizing for the modern SIMD architectures. One is the proliferation of SIMD capabilities in modern computing platforms, including gaming machines [18], Digital Signal Processors (DSPs) [7, 10], and even in general purpose processors [15, 8]. The second factor is the growing significance of applications that can benefit from SIMD functionality, particularly those in the multimedia domain.

The classic vectorization theory does not apply very successfully to SIMD machines [16], for several reasons. Traditional vectorization has focused on array-based Fortran programs from the scientific computing domain. Many of the important modern workloads, such as multimedia applications, are written in C and make extensive use of pointers. The presence

of pointers, and other programming language differences [2] give rise to a new domain of problems critical for the success of vectorization.

The primary difficulties in applying classic vectorization to SIMD lie in the architectural differences between SIMD extensions and traditional vector architectures. First, SIMD memory architectures are typically weaker than those of traditional vector machines. The former generally only support accesses to contiguous memory items, and only on vector-length aligned boundaries. Computations, however, may access data elements in an order which is neither contiguous nor adequately aligned. SIMD architectures usually provide mechanisms to reorganize data elements in vector registers in order to deal with such situations. These mechanisms (packing and unpacking data elements in and out of vector registers and special permute instructions) are not easy to use, and incur considerable penalties. For a vectorizer, this implies that generating vector memory accesses becomes much more involved.

In addition, the instruction sets of SIMD architectures tend to be much less general purpose and less uniform. Many specialized domain-specific operations are included, many operations are available only for some data types but not for others, and often a high-level understanding of the computation is required in order to take advantage of some of the functionality. Furthermore, these particular characteristics differ from one architecture to another.

These attributes demand that low-level architecture-specific factors will be considered throughout the process of vectorization. Classic dependence analysis is therefore only a partial solution to vectorizing for SIMD extensions. Code transformation issues require much more attention, as discussed later in the paper.

## 3 Data-Dependence Analysis

The classic approach for vectorization is based on the theory of data-dependence analysis. Some parts of the classic data-dependence-based analyses and transformations are already present in GCC (currently in the loop-nest-optimizations (lno) branch of GCC).

The first step in dependence analysis is the construction of a data dependence graph (ddg). The nodes of the graph are the loop statements, and edges between statements represent a data dependence between them. There are two types of such edges. Edges between scalar variables represent a def-use link between statements. These links can be trivially computed from a SSA representation, such as the one used in the tree-ssa representation level of GCC.

The second kind of edges are those between memory references. The classic data-dependence theory focused on array-based Fortran programs, and therefore only array references have traditionally been considered. In other (e.g., C) programs, memory references can take other forms (e.g., indirect references through pointers), and these are considered by the GCC vectorizer. Memory dependences are determined by applying a set of dependence tests [9, 3] that compare array subscripts. Simpler and faster tests (GCD, Banerjee) are applied to simple subscript forms. More complex and accurate tests (Gamma, Delta, Omega) are applied to more complicated subscripts.

If a dependence is carried by the relevant nesting level then an edge is added to the ddg. For example, in Figure 1, loops (a) and (d) both have a dependence between the two references to array `a`, but only the dependence in loop (d) is carried across the loop itera-

tions and prevents vectorization. In GCC, tests that compute dependences between array references are implemented in the module `tree-data-refs.c`. They are used by the vectorizer to detect dependences between data references in inner-most loops.

The classic data dependence analysis proceeds to detect Strongly Connected Components (SCCs) in the ddg. SCCs that consist of a single node represent a statement that can be executed in parallel at the loop level that is being considered. SCCs that consist of multiple nodes represent statements that are involved in a dependence cycle, and prevent the vectorization of the loop unless the cycle can be "broken."

In order to increase the potential for vectorization, the vectorizable parts can be separated from the groups of statements that are involved in dependence cycles (loop distribution). This is done by creating a separate loop for each SCC, after having topologically sorted the reduced graph in which each SCC is represented by a single node. There is a preliminary implementation of ddg construction in GCC (as part of the scalar-evolutions module) but it is not yet used. Loop distribution to increase vectorization opportunities is not yet supported, however other loop transformations that increase parallelism (loop interchange, scaling, skewing, and reversal) are supported in GCC as part of the tree-loop-linear module.

Lastly, special kinds of dependence cycles can be dealt with if recognized as certain idioms, such as reduction. The GCC vectorizer will be enhanced to recognize and handle such situations in the near future.

# 4 Vectorizer Overview

The vectorization optimization pass is developed in the loop-nest-optimizations (lno) branch (`http://gcc.gnu.org/projects/tree-ssa/lno.html`), at the IR level of SSA-ed GIMPLE trees. The current development status can be found on `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`.
The vectorizer is enabled by the `-ftree-vectorize` flag which also invokes the scalar-evolutions analyzer, upon which the vectorizer relies for induction variable recognition and loop bound calculation. The bulk of the vectorizer functionality can be found in two files (`tree-vectorizer.c` and `tree-vectorizer.h`). The vectorizer also uses loop-related utilities that reside elsewhere, many of which are new contributions developed in the lno branch.

The vectorization pass is in early stages of development; the basic infrastructure is in place, supporting initial vectorization capabilities. These capabilities are demonstrated by the vectorization test-cases, which are updated to reflect new capabilities as they are added. Work is underway to extend these capabilities and to introduce more advanced vectorization features.

### 4.1 vectorizer layout

An outline of the vectorization pass is given in Figure 2. The main entry to the vectorizer is `vectorize_loops(loops)`. The vectorizer applies a set of analyses on each loop, followed by the actual vector transformation for the loops that had successfully passed the analysis phase.

### 4.2 vectorizer analysis

The first analysis phase, `analyze_loop_form()`, examines the loop exit condition and number of iterations, as well as some control-flow attributes such as number of basic blocks and nesting level. One major restriction im-

```
vect_analyze_loop (struct loop *loop) {
  loop_vec_info loopinfo;
  loop_vinfo = vect_analyze_loop_form (loop);
  if (!loop_vinfo)
    FAIL;
  if (!analyze_data_refs (loopinfo))
    FAIL;
  if (!analyze_scalar_cycles (loopinfo))
    FAIL;
  if (!analyze_data_ref_dependences (loopinfo))
    FAIL;
  if (!analyze_data_ref_accesses (loopinfo))
    FAIL;
  if (!analyze_data_refs_alignment (loopinfo))
    FAIL;
  if (!analyze_operations (loopinfo))
    FAIL;
  LOOP_VINFO_VECTORIZABLE_P (loopinfo) = 1;
  return loopinfo;
}

vect_transform_loop (struct loop *loop) {
  FOR_ALL_STMTS_IN_LOOP(loop, stmt)
    vect_transform_stmt (stmt);
  vect_transform_loop_bound (loop);
}
```

Figure 2: Vectorizer outline

(a) uncountable loop:

```
while (*p != NULL){
  *p++ = X;
}
```

(b) reduction - summation:

```
for (i=0; i<n; i++){
  sum += a[i];
}
```

(c) induction:

```
for (i=0,j=0; i<n; j++,i++){
  a[i] = j;
}
```

(d) non consecutive access pattern:

```
for (i=0; i<n; i++){
  a[2*i] = X;
}
```

Figure 3: Loop examples

posed on a loop for vectorization to be applicable, is that the loop is countable—i.e, an expression that calculates the loop bound can be constructed and evaluated either at compile time or at run-time. For example, the loop in Figure 3a is not a countable loop. The loop bound analysis is carried out by the scalar evolution analyzer. To simplify the initial implementation, the vectorizer also verifies that the loop is an inner-most loop, and consists of a single basic block. Multi-basic-block constructs such as if-then-else are collapsed into conditional operations if possible, by an if-conversion pass prior to vectorization.

Next, `analyze_data_refs()` finds all the memory references in the loop, and checks if they are "analyzable"—i.e., an access function that describes their modification in the loop (evolution) can be constructed. This is required for the memory-dependence, access-pattern and alignment analyses (described in Section 5). Other restrictions enforced at this point are there for simplicity of implementation, and will be relaxed in the near future.

Dependences which do not involve memory operations are analyzed directly from the SSA representation. The function `analyze_scalar_cycles()` examines such "scalar-cycles" (dependence cycles which involve only scalar variables), and verifies that any scalar cycle, if exists, can be handled in a way that breaks the cross-iteration dependence.

One kind of such "breakable" scalar cycles are those that represent reductions. A reduction operation computes a scalar result from a set of data-elements. The loop in Figure 3b for example, computes the sum of a set of array elements into a scalar result `sum`. Some reduction operations can be vectorized, generally by computing several partial results in parallel, and combining them at the end (reducing them) to single result. Scalar cycles can also be created by induction variables (IVs). Certain IVs that are used for loop control and for address computation, are handled as an inherent part of vectorization. An example of an IV of this type

is `i` from Figure 3c. Other IVs such as `j` from the same example require special vectorization support. Support for vectorization of reduction and induction will be introduced to GCC in the near future.

The final analysis phase `analyze_operations()` scans all the operations in the loop and determines a vectorization factor. The vectorization factor (VF) represents the number of data elements that will be packed together in a vector, and is also the strip-mining factor of the loop. It is determined according to the data-types operated on in the loop, and the length of the vectors supported by the target platform. Currently we use a simple approach that allows a single vector length per platform and a single data-type per loop, but these restrictions will be relaxed in the near future. `analyze_operations()` also checks that all the operations can be supported in vector form. The cost of expanding them to scalar code in case they are not supported, is expected to offset the benefits of vectorizing the loop. In the future, a cost model should be devised to support the vectorizers decisions on which loops to vectorize.

### 4.3 vectorizer transformation scheme

During the analysis phase the vectorizer records information at three levels of granularity—at the loop level (`loop_vect_info`), at the statement level (`stmt_vec_info`), and per memory reference (`data_reference`). These data-structures are later used during the loop transformation phase.

The vectorization transformation can be generally described as "strip-mine by VF and substitute one-to-one," which implies that each scalar operation in the loop is replaced by its vector counterpart. The loop transformation phase scans all the statements of the loop top-down (defs are vectorized before their uses), inserting a vector statement VS in the loop for each scalar statement S that needs to be vectorized, and recording in the `stmt_vec_info` attached to S a pointer to VS; This pointer is used to locate the vectorized version of statement S during the vectorization of subsequent statements that depend on S.

(a) before vectorization:

```
S1:  x = b[i];
S2:  z = x + y;
S3:  a[i] = z;
```

(b) after vectorization of S1:

```
VS1: vx = vpb[indx];
S1:  x = b[i];   ---> VS1
S2:  z = x + y;
S3:  a[i] = z;
```

(c) after vectorization of S2:

```
VS1: vx = vpb[indx];
S1:  x = b[i];   ---> VS1
VS2: vz = vx + vy;
S2:  z = x + y;  ---> VS2
S3:  a[i] = z;
```

(d) after vectorization of S3:

```
VS1: vx = vpb[indx];
S1:  x = b[i];   ---> VS1
VS2: vz = vx + vy;
S2:  z = x + y;  ---> VS2
VS3: vpa[indx] = vz;
```

Figure 4: The transformation process

After all statements have been vectorized, the original scalar statements may be removed. Stores to memory are explicitly removed by the vectorizer; the remaining scalar statements are expected to be removed by dead code elimination pass after vectorization.

Figure 4 illustrates the transformation process; First, stmt S1 is vectorized into stmt VS1; In order to vectorize stmt S2, the vectorizer needs to find the relevant vector def-stmt for each operand of S2. The figure only shows how

this is done for the operand x: first, the scalar def-stmt of x, S1, is found (using SSA), and the relevant vector def vx is retrieved from the vectorized statement of S1, VS1. Similarly for S3, except that S3 is also removed.

In practice, many cases require additional handling beyond the "one-to-one substitution." Constants and loop invariants require that vectors be initialized at the loop pre-header. Other computations require special epilog code after the loop (e.g., reductions, and inductions that are used after the loop). Some access-patterns require special data manipulations between vectors within the loop (e.g., data interleaving and permutations). Some scalar operations cannot be replaced by a single vector operation (e.g., when mixed data-types are present). Sometimes a sequence of scalar operations can be replaced by a single vector operation (e.g., saturation, and other special idioms). It might be possible to hide some of these complications from the vectorizer, and handle them at lower levels of code generation. We discuss these issues in Section 6.

Finally, the loop bound is transformed to reflect the new number of iterations, and if necessary, an epilog scalar loop is created to handle cases of loop bounds which do not divide by the vectorization factor. This epilog also must be generated in cases where the loop bound is not known at compile time.

# 5 Handling Memory References

Memory references require special attention when vectorizing. This is true in the classic vectorization framework, and even more so when vectorizing for SIMD. The vectorizer currently considers two forms of data-refs—one-dimensional arrays (represented as ARRAY_REFs that are VAR_DECLs), and pointer accesses (INDIRECT_REFs). Once an access function has been computed (for the array index or the pointer) the vectorizer proceeds to apply a set of data-ref analyses, which we describe here.

## 5.1 Dependences and aliasing

As mentioned above, one of the basic restrictions that has to be enforced in order to safely apply vectorization is that no dependence cycles exist. A simplified form of the standard memory dependence analysis, which we briefly described in Section 3, is applied, using simple dependence tests from the tree-data-ref module of the lno-branch.

This analysis can be enhanced in several directions, including: (1) using more complex dependence tests, (2) pruning dependences with distance greater than the vectorization factor, and (3) not failing when a dependence is found, but instead attempting to resolve the dependence by reordering nodes in the dependence graph (consequently distributing the loop).

Pointer accesses require in addition alias analysis to conclude whether any two pointer-accesses in the loop may alias. If we cannot rule out the possibility that two pointers may alias, loop versioning can be used, with a run-time-overlap test to guard the vectorized version of the loop.

## 5.2 Access pattern

When the data is laid out in memory exactly in the order in which it is needed for the computation, it can be vectorized using the simple one-to-one vectorization scheme. However, computations may access data elements in an order different from the way they are organized in memory. For example, the computation in Figure 3d uses a strided access pattern (with stride 2). Non-consecutive access patterns usually require special data manipulations to re-

order the data elements and pack them into vectors. This is because the memory architecture restricts vector data accesses to consecutive vector-size elements.

Some architectures provide relatively flexible mechanisms to perform such data manipulations (gather/scatter operations in traditional vector machines, indirect access through vector pointers [14]). SIMD extensions usually provide mechanisms to pack data from two vectors into one (and vice versa), while possibly applying a permutation on the data elements. Some SIMD extensions provide specialized support for certain access-patterns (most commonly for accessing odd/even elements for computations on complex numbers), but these are usually limited only to a few operations and a few data types.

The underlying data reorganization support determines whether vectorization can be applied, and at what cost. These data manipulations need to be applied in each iteration of the loop and therefore incur considerable overhead. In fact, some access patterns, such as indirect access, cannot be vectorized efficiently on most SIMD/vector architectures. The function `analyze_access_pattern()` verifies that the access pattern of all the data references in the loop is supported by the vectorizer, which is currently limited to consecutive accesses only.

### 5.3 alignment

Accessing a block of memory from a location which is not aligned on a natural vector-size boundary is often prohibited or bears a heavy performance penalty. These memory alignment constraints raise problems that can be handled using data reordering mechanisms. Such mechanisms are costly, and usually involve generating extra memory accesses and special code for combining data elements from different vectors in each iteration of the loop.

In order to avoid these penalties, techniques like loop peeling and static and dynamic alignment detection [11, 13, 4] can be used. Alignment handling therefore consists of three levels: (1) static alignment analysis, (2) transformations to force alignment, including runtime checks, and (3) efficient vectorization of the remaining misaligned accesses.

The functions `compute_data_refs_alignment()` and `enhance_data_refs_alignment()` (called from `analyze_data_refs_alignment()`) are responsible for items (1) and (2) above. `compute_data_refs_alignment()` computes misalignment information for all data-references; currently only a trivial conservative implementation is provided.

Following the alignment computation, the function `enhance_data_refs_alignment()` uses loop versioning and loop peeling in order to force the alignment of data references in the loop. Loop peeling can only force the alignment of a single data reference, so the vectorizer needs to choose which data reference DR to peel for. In the peeled loop, only the access DR is guaranteed to be aligned. Loop versioning could be applied on top of peeling, to create one loop in which all accesses are aligned, and another loop in which only the access DR is guaranteed to be aligned. A cost model should be devised to guide the vectorizer as to which access to peel for, and whether to apply peeling or versioning or a combination of the two, considering the code size and runtime penalties. Figure 5 illustrates these different alternatives.

If data-references which are not known to be aligned still remain after `enhance_data_refs_alignment()`, the vectorizer will proceed to vectorize the loop only if the target platform provides mechanisms to support

(a) original loop, before alignment analysis:

```
for (i = 0; i < N; i++) {
 x = q[i]; //misalign(q) = unknown
 p[i] = y; //misalign(p) = unknown
}
```

(b) after compute_data_refs_alignment():

```
for (i = 0; i < N; i++) {
 x = q[i]; //misalign(q) = 3
 p[i] = y; //misalign(p) = unknown
}
```

(c) option 1—loop versioning:

```
if (p is aligned) {
 for (i = 0; i < N; i++) {
   x = q[i]; //misalign(q) = 3
   p[i] = y; //misalign(p) = 0
 }
} else {
 for (i = 0; i < N; i++) {
   x = q[i]; //misalign(q) = 3
   p[i] = y; //misalign(p) = unknown
 }
}
```

(d) option 2—peeling for access q[i]:

```
for (i = 0; i < 3; i++) {
 x = q[i];
 p[i] = y;
}
for (i = 3; i < N; i++) {
 x = q[i]; //misalign(q) = 0
 p[i] = y; //misalign(p) = unknown
}
```

(e) option 3—peeling and versioning:

```
for (i = 0; i < 3; i++) {
 x = q[i];
 p[i] = y;
}
if (p is aligned) {
 for (i = 3; i < N; i++) {
   x = q[i]; //misalign(q) = 0
   p[i] = y; //misalign(p) = 0
 }
} else {
 for (i = 3; i < N; i++) {
   x = q[i]; //misalign(q) = 0
   p[i] = y; //misalign(p) = unknown
 }
}
```

Figure 5: Alternatives for forcing alignment

misaligned accesses. Figure 6c presents a possible scheme for handling misalignment [6]. It relies on a pair of target hooks: one that calculates the misalignment amount, and represents it in a form that the second hook can use (a shift amount or a permutation mask). The second hook combines data from two vectors, permuted according to the misalignment shift amount. In some cases the code could be further optimized by exploiting the data reuse across loop iterations, as shown in Figure 6d.

Targets that support misaligned accesses directly, do not need to implement these hooks; in this case, misaligned vector accesses will look just like regular aligned vector accesses, as in Figure 6b. Section 6 discusses the tradeoffs involved in this implementation scheme.

## 6 Vectorization issues

An issue that repeatedly comes up during the development of the GCC vectorization is the tension between two conflicting needs. One is the requirement to maintain a high-level, platform-independent program representation. The other is the need to consider platform-specific issues and express low-level constructs during the process of vectorization.

In many ways, the tree-level is the suitable place for the implementation of a loop-based vectorizer in GCC. Arrays and other language constructs are represented in a relatively high-level form, a fact that simplifies analyses such as alignment, aliasing and loop-level data-dependences. Analyses are further simplified due to the SSA representation. Implementing the vectorizer at the tree-ssa level allows it to benefit from the vast suite of SSA optimizations, and in particular, the loop related utilities developed in the lno-branch.

On the other hand, at this IR level it is not so trivial to handle situations in which target-

(a) scalar data-ref:

```
    i = init;
LOOP:
    x = a[i];
    i++
```

(b) vectorized data-ref:

```
    vector *vpx = &a[init];
    indx = 0;
LOOP:
    vector vx = (*vpx)[indx];
    indx++
```

(c) vectorized data-ref with misalignment support:

```
    vector *vpx1 = &a[init];
    vector *vpx2 = &a[init+VF-1];
    shft = target_hook_get_shft
                          (&a[init])
    indx = 0;
LOOP:
    vx1 = (*vpx1)[indx];
    vx2 = (*vpx2)[indx];
    vx = target_hook_combine_by_shft
                   (vx1, vx2, shft)
    indx++;
```

(d) optimized misalignment support:

```
    vector *vpx1 = &a[init];
    vector *vpx2 = &a[init+VF-1];
    shft = target_hook_get_shft
                          (&a[init]);
    indx = 0;
    vx1 = (*vpx1)[indx];
LOOP:
    vx2 = (*vpx2)[indx];
    vx = target_hook_combine
                   (vx1, vx2, shft);
    indx++;
    vx1 = vx2;
```

Figure 6: Handling data-refs (load example)

specific information needs to be consulted, and even less trivial to handle situations in which target-specific constructs need to be expressed.

Misalignment is an excellent example of such a situation. The low-level functionality that supports misaligned accesses must somehow be expressed in the tree IR. The implementation should maintain the following properties: (1) It should hide low-level details as much as possible. (2) It should be general enough to be applicable to any platform. SIMD extensions vary greatly from platform to platform. (3) Despite these restrictions, it should be as efficient as possible on each platform.

In terms of the above criteria, the misalignment scheme that was presented in the previous section: (1) exposes the vectorizer to low-level details of misalignment support, (2) might not be general enough (it assumes that low-order address bits are ignored by load operations), and (3) is potentially inefficient for targets that would be better supported by alternative methods.

To tackle these problems, two alternatives can be considered. Alternative 1: Annotate misaligned accesses and let the subsequent RTL expansion pass handle the details. This is the most natural way to address architecture specific details. However, this solution can potentially be very inefficient, because it neglects to take advantage of data reuse between iterations. To do that, the lower-level RTL passes would have to rediscover the kind of loop-level information the vectorizer already had. Alternative 2: Hide all these implementation details in a "black box" target hook, that would generate the most efficient code for its platform. A disadvantage would be that functionality that is common to many targets would have to be duplicated. Also, the vectorizer would be unaware of what's going on, and would have difficulty estimating the overall cost of applying

vectorization, for example.

Low-level architectural details are not only problematic with respect to representing them at a high-level platform-independent abstraction. Specific architectural vector support can directly affect the vectorization transformation, and even determine whether it should be applied at all. These details must be considered during vectorization because the choices made at the vectorization stage are not easily altered at later low-level stages of compilation. This is especially true in cases of architectural features that require recognition of an entire computational idiom, a task best supported by high-level analysis (reductions for example may be difficult to identify without the entire context of the loop).

These are some of tradeoffs and decisions involved in the implementation of the GCC vectorizer. These kinds of problems often come up in optimizing compilers, but are especially evident in the context of SIMD vectorization, and even more so when implemented in a multi-platform compiler like GCC.

## 7 Status

The first implementation of a basic vectorizer in GCC was contributed to the lno-branch on January 1st, 2004. It has since been enhanced with additional capabilities, including support for vectorization of constants, loop invariants, and unary and bitwise operations. The vector test-suite (`gcc/gcc/testsuite/gdd.dg/tree-ssa-vect/`) reflects the current vectorization functionality. The domain of vectorizable loops can be summarized in terms of the supportable (1) loop forms, (2) data references, and (3) operations. Currently support includes (1) inner-most, single-basic-block loop forms, with a known loop bound divisible by the vectorization factor; (2) consecutive array data references for which alignment can be forced, and (3) operations that do not create a scalar-cycle (no reduction or induction), that all operate on the same data-type, and that have a vector form that can be expressed using existing tree-codes.

Recent development has focused on broadening the range of loop-forms and data references that the vectorizer can support. This includes the vectorization of loops with unknown loop bounds, an if-conversion pass that allows the vectorizer to handle some forms of multi-basic-block loops, vectorization of unaligned loads, and vectorization of pointer accesses. These features are likely to be added by the time this paper is presented, and will soon be followed by support for peeling and versioning for alignment. Other future directions include support for multiple data-types, and for reduction and induction operations. In the next section we discuss additional directions for further development of the vectorizer.

## 8 Future Work

Following is a list of potential enhancements to the vectorizer, organized into four categories:

**Support additional loop forms**. Support for unknown loop bounds and if-then-else constructs is nearly complete. The major remaining restriction on loop form is the nesting level. Vectorization of nested loops will be considered in the future.

**Support additional forms of data references**. Potential extensions in this category include enhancements to the dependence tests (as discussed in Section 5) and support for additional access patterns (reverse access, and accesses that require data manipulations like strided or permuted accesses). Exploiting data reuse as in [17] is an optimization related to data references that we plan to consider in the future.

**Support additional operations**. Vectorization of loops with multiple data-types and type casting is the first extension expected in this category. This capability requires support for data packing and unpacking, which breaks out of the one-to-one substitution scheme, and cannot be directly expressed using existing tree-codes. The next capabilities to be introduced will be support for vectorization of induction, reduction, and special idioms (such as saturation, min/max, dot product, etc.), using target hooks or adding new tree-code as necessary.

**Other enhancements and optimizations**. Two general capabilities that we are planning to introduce are support for multiple vector lengths for a single target, and the ability to evaluate the cost of applying vectorization. This will require some form of cost modelling for the vector operations. Interaction with other optimization passes should also be examined, and in particular, potential interaction with other (new) passes that might also exploit data parallelism. One example could be loop parallelization (using threads). Another example could be straight-line code vectorization (as opposed to loop based), such as SLP [12].

SLP is in many ways suitable for lower representation levels, as it analyzes addresses, and operates like a peep-hole optimization on a single basic block at a time. This is what gives SLP it's main strength—scalar operations are grouped together into a vector operation without a need to prove general attributes about an enclosing loop as a whole. (In fact, it is not even aware of any enclosing loops). This property allows it to vectorize code sequences that the loop based vectorizer does not target. However, this is also its main limiting factor, and it can benefit from loop-level information which is already available to the tree-level loop-based vectorizer. We are therefore considering implementing SLP at the tree-level, as a complementary solution to the loop-based vectorizer.

With the introduction of support for unknown loop bounds, pointers, misalignment, and conditional code, the GCC vectorizer will be in relatively good shape compared to other vectorizing compilers. The major remaining restrictions (inner-most loops, consecutive accesses and a single data-type per loop) tend to be common to vectorizing compilers in general [5, 19]. However, as the (long) list above implies, most of the exciting features still lie ahead.

## 9 Acknowledgments

The vectorizer directly uses, or otherwise benefits from, utilities developed in the lno-branch contributed by Sebastian Pop, Zdenek Dvorak, Devang Patel, and Daniel Berlin. Many thanks to Sebastian for continuious support for smooth interaction of his analyzer with the vectorizer, and to Zdenek for ongoing resposive maintenance of the lno-branch. I would like to thank Olga Golovanevsky for her contributions to vectorizer functionality, to Ayal Zaks and the IBM Haifa team for helpful discussions, and to GCC contributors who offered help, comments and patches.

## References

[1] John Randal Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, October 1987.

[2] Randy Allen and Steve Johnson. Compiling c for vectorization, parallelization, an inline expansion. In *SIGPLAN Conference on Programming Languages Design and Implementation*, June 1988.

[3] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern*

*Architectures—A dependence-based approach*. Morgan Kaufmann, 2001.

[4] A. J. C. Bik, M. Girkar, P. M. Grey, and X. Tian. Efficient exploitation of parallelism on Pentium III and Pentium 4 processor-based systems. *Intel Technology J.*, February 2001.

[5] Aart J.C. Bik, Milind Girkar, Paul M. Grey, and Ximmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):65–98, April 2002.

[6] Apple Computer. `http://developer.apple.com/hardware/ve/`.

[7] Paul D'Arcy and Scott Beach. StarCore SC140: A new DSP architecture for portable devices. In *Wireless Symposium*. Motorola, September 1999.

[8] K. Diefendorff and P. K. Dubey et al. Altivec extension to PowerPC accelerates media processing. *IEEE Micro*, March-April 2000.

[9] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. In *SIGPLAN Conference on Programming Languages Design and Implementation*, June 1991.

[10] Texas Instruments. www.ti.com/sc/c6x, 2000.

[11] Andreas Krall and Sylvain Lelait. Compilation techniques for multimedia processors. *Intl. J. of Parallel Programming*, 28(4):347–361, 2000.

[12] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *PLDI*, 35(5):145–156, 2000.

[13] Samuel Larsen, Emmett Witchel, and Saman Amarasinghe. Techniques for increasing and detecting memory alignment. Technical Memo 621, MIT LCS, November 2001.

[14] Jaime H. Moreno, V. Zyuban, U. Shvadron, F. Neeser, J. Derby, M. Ware, K. Kailas, A. Zaks, A. Geva, S. Ben-David, S. Asaad, T. Fox, M. Biberstein, D. Naishlos, and H. Hunter. An innovative low-power high-performance programmable signal processor for digital communications. *IBM Journal of Research and Development*, March 2003.

[15] A. Peleg and U. Weiser. MMX technology extension to the Intel architecture. *IEEE Micro*, pages 43–45, August 1996.

[16] Gang Ren, Peng Wu, and David Padua. A preliminary study on the vectorization of multimedia applications for multimedia extensions. *16th International Workshop of Languages and Compilers for Parallel Computing*, October 2003.

[17] Jaewook Shin, Jacqueline Chame, and Mary W. Hall. Compiler-controlled caching in superword register files for multimedia extension architectures. In *PACT*, 2002.

[18] Sony. `http://www.us.playstation.com/`.

[19] N. Sreraman and R. Govindarajan. A vectorizing compiler for multimedia extension architectures. *International Symposium on Microarchitecture*, pages 25–36, 1998.

[20] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison Wesley, 1996.

# Design and Implementation of Tree SSA

*Diego Novillo*
Red Hat Canada
dnovillo@redhat.com

## Abstract

Tree SSA is a new optimization framework for GCC that allows the implementation of machine and language independent transformations. This paper describes the major components of Tree SSA, how they interact with the rest of the compiler and, more importantly, how to use the framework to implement new optimization passes.

## 1 Introduction

The main goal of the Tree SSA project is to evolve GCC's optimization infrastructure to allow more powerful analyses and transformations that had traditionally proven difficult or impossible to implement in RTL. Though originally started as a one person hobby project, other developers in the community expressed interest in it and a development branch off the main GCC repository was started. Soon thereafter, Red Hat began sponsoring the project and, over time, other organizations and developers in the community also started contributing. Presently, about 30 developers are actively involved in it[1], and work is underway to implement vectorization and loop optimizations based on the Tree SSA framework. We expect Tree SSA to be included in the next major release of GCC.

Although Tree SSA represents a significant change in the internal structure of GCC, its main design principle has been one of evolution, not revolution. As much as possible, we tried to make Tree SSA a "drop-in" module. In particular, we decided to keep the `tree` and `rtl` data structures so that neither front ends nor back ends needed to be re-implemented from scratch. This was an important engineering decision that (a) allowed us to reach to a working system in a relatively short period of time, but (b) it exposed a few weak spots in the existing data structures that we will need to address in the future (Section 8).

This paper describes Tree SSA from a programmer's point of view. Emphasis is placed on how the different modules work together and what is necessary to implement a Tree SSA pass in GCC. Section 2 provides an overview of the new files and compiler switches added to GCC. Section 3 describes the GENERIC and GIMPLE intermediate representations. Section 4 describes the control flow graph (CFG), block and statement manipulation functions. Section 5 describes how optimization passes are scheduled and declared to the pass manager. Section 6 describes the basic data flow infrastructure: statement operands and the SSA form as implemented on GIMPLE. Section 7 describes alias analysis. Conclusions and future work are in Section 8.

[1] This is a rough estimate based only on ChangeLog entries.

# 2 Overview

## 2.1 Command line switches

Most of the new command line options added by Tree SSA are only useful to GCC developers. They fall into two broad categories: debugging dumps and individual pass manipulation.

All the debugging dumps are requested with `-fdump-tree-`*pass*`-`*modifier*. By default, the tree representation is emitted in a syntax resembling C. Passes can be individually selected, but the most common usage is to enable all of them using `-fdump-tree-all`. When enabled, each pass dumps all the functions in the input program to a separate file. Dump files are numbered in the same order in which passes are applied. Therefore, to see the effects of a single pass, one can just run *diff* between the $N$ and $N + 1$ dumps.

Modifiers affect the format of the dump files and/or the information included in them[2]. Currently, the following modifiers can be used:

- `raw`: Do not pretty-print expressions. Use the traditional tree dumps instead.
- `details`: Request detailed debugging output from each pass.
- `stats`: Request statistics from each pass.
- `blocks`: Show basic block boundaries.
- `vops`: Show virtual operands (see Section 6 for details).
- `lineno`: Show line numbers from the input program.
- `uid`: Show the unique ID (i.e., `DECL_UID`) for every variable.

We currently support enabling and disabling most individual SSA passes. Although, it is not clear whether that will be always supported, it is sometimes useful to disable passes when debugging GCC. Note, however, that even if the bug goes away when disabling an individual pass, it does not mean that the pass itself is faulty. The bug may exist somewhere else and is exposed at this point.

All the new command line switches are described in detail in the GCC documentation.

## 2.2 New files

All the necessary API and data structure definitions are in *tree-flow.h* and *tree-passes.h*. The remaining files can be loosely categorized as basic infrastructure, transformation passes, analysis passes and various utilities.

### 2.2.1 Basic infrastructure

*tree-optimize.c* is the main driver for the tree optimization passes. In particular, it contains `init_tree_optimization_passes`, which controls the scheduling of all the tree passes, and `tree_rest_of_compilation` which performs all the gimplification, optimization and expansion into RTL of a single function.

*tree-ssa.c*, *tree-into-ssa.c* and *tree-outof-ssa.c* implement SSA renaming, verification and various functions needed to interact with the SSA form.

*tree-ssanames.c* and *tree-phinodes.c* implement memory management mechanisms for re-using `SSA_NAME` and `PHI_NODE` tree nodes after they are removed.

[2] Note that not all passes are affected by these modifiers. A pass that does not support a specific modifier will silently ignore it.

*tree-cfg.c* contains routines to build and manipulate the CFG.

*tree-dfa.c* implements general purpose routines for dealing with program variables and data flow queries like immediate use information.

*tree-ssa-operands.c* contains routines for scanning statement operands (`get_stmt_operands`).

*tree-iterator.c* contains routines for inserting, removing and iterating over GIMPLE statements. Two types of iterators are provided, those that do not stop at basic block boundaries (known as *tree statement iterators*) and those that do (known as *block statement iterators*). Most optimization passes use the latter.

*c-gimplify.c*, *gimplify.c* and *tree-gimple.c* contain the routines used to rewrite the code into GIMPLE form. They also provide functions to verify GIMPLE expressions.

### 2.2.2 Transformation passes

*gimple-low.c* removes binding scopes and converts the clauses of conditional expressions into explicit gotos. This is done early before any other optimization pass as it greatly simplifies the job of the optimizers.

*tree-ssa-pre.c*, *tree-ssa-dse.c*, *tree-ssa-forwprop.c*, *tree-ssa-dce.c*, *tree-ssa-ccp.c*, *tree-sra.c* and *tree-ssa-dom.c* implement some commonly known scalar transformations: partial redundancy elimination, dead store elimination, forward propagation, dead code elimination, conditional constant propagation, scalar replacement of aggregates and dominator-based optimizations.

*tree-ssa-loop.c* is currently a place holder for all the optimizations being implemented in the LNO (Loop Nest Optimizer) branch [1]. Presently, it only implements loop header copying, which moves the conditional at the bottom of a loop to its header (benefiting code motion optimizations).

*tree-tailcall.c* marks tail calls. The RTL optimizers will make the final decision of whether to expand calls as tail calls based on ABI and other conditions.

*tree-ssa-phiopt.c* tries to replace PHI nodes with an assignment when the PHI node is at the end of a conditional expression.

*tree-nrv.c* implements the named return value optimization. For functions that return aggregates, this optimization may save a structure copy by building the return value directly where the target ABI needs it.

*tree-ssa-copyrename.c* tries to reduce the number of distinct SSA variables when they are related by copy operations. This increases the chances of user variables surviving the out of SSA transformation.

*tree-mudflap.c* implements pointer and array bound checking. This pass re-writes array and pointer dereferences with bound checks and calls to its runtime library. Mudflap is enabled with `-fmudflap`.

*tree-complex.c*, *tree-eh.c* and *tree-nested.c* rewrite a function in GIMPLE form to expand operations with complex numbers, exception handling and nested functions.

### 2.2.3 Analysis passes

*tree-ssa-alias.c* implements type-based and flow-sensitive points-to alias analysis.

*tree-alias-type.c*, *tree-alias-ander.c* and *tree-alias-common.c* implement flow-insensitive points-to alias analysis (Andersen analysis).

### 2.2.4 Various utilities

*tree-ssa-copy.c* contains support routines for performing copy and constant propagation.

*domwalk.c* implements a generic dominator tree walker.

*tree-ssa-live.c* contains support routines for computing live ranges of SSA names.

*tree-pretty-print.c* implements `print_generic_stmt` and `print_generic_expr` for printing GENERIC and GIMPLE tree nodes.

*tree-browser.c* implements an interactive tree browsing utility, useful when debugging GCC. It must be explicitly enabled with `--enable-tree-browser` when configuring the compiler.

## 3 Intermediate Representation

Although Tree SSA uses the `tree` data structure, the parse trees coming out of the various front ends cannot be used for optimization because they contain language dependencies, side effects and can be nested in arbitrary ways. To address these problems, we have implemented two intermediate representations: GENERIC and GIMPLE [4].

GENERIC provides a way for a language front end to represent entire functions in a language-independent way. All the language semantics must be explicitly represented, but there are no restrictions in how expressions are combined and/or nested. If necessary, a front end can use language-dependent trees in its GENERIC representation, so long as it provides a hook for converting them to GIMPLE. In particular, a front end need not emit GENERIC at all. For instance, in the current implementation, the C and C++ parsers do not actually emit GENERIC during parsing.

GIMPLE is a subset of GENERIC used for optimization. Both its name and the basic grammar are based on the SIMPLE IR used by the McCAT compiler at McGill University [3]. Essentially, GIMPLE is a 3 address language with no high-level control flow structures.

1. Each GIMPLE statement contains no more than 3 operands (except function calls) and has no implicit side effects. Temporaries are used to hold intermediate values as necessary.

2. Lexical scopes are represented as containers.

3. Control structures are lowered to conditional gotos.

4. Variables that need to live in memory are never used in expressions. They are first loaded into a temporary and the temporary is used in the expression.

The process of lowering GENERIC into GIMPLE, known as *gimplification*, works recursively, replacing complex statements with sequences of statements in GIMPLE form. A front end which wants to use the tree optimizers needs to

1. have a whole-function tree representation,

2. provide a definition of `LANG_HOOKS_GIMPLIFY_EXPR`,

3. call `gimplify_function_tree` to lower to GIMPLE, and,
4. hand off to `tree_rest_of_compilation` to compile and output the function.

The GCC internal documentation includes a detailed description of GENERIC and GIMPLE that an implementor of new language front ends will find useful.

## 4 Control Flow Graph and IR manipulation

Data structures for representing basic blocks and edges are shared between GIMPLE and RTL. This allows the GIMPLE CFG to use all the functions that operate on the flow graph independently of the underlying IR (e.g., dominance information, edge placement, reachability analysis). For the cases where IR information is necessary, we either replicate functionality or have introduced hooks.

The flow graph is built once the function is put into GIMPLE form and is only removed once the tree optimizers are done[3].

Traversing the flow graph can be done using `FOR_EACH_BB`, which will traverse all the basic blocks sequentially in program order. This is the quickest way of going through all basic blocks. It is also possible to traverse the flow graph in dominator order using `walk_dominator_tree`.

Each basic block has a list of all the statements that it contains. To traverse this list, one should use a special iterator called *block statement iterator* (BSI). For instance, the code fragment in Figure 1 will display all the statements in the function being compiled (`current_function_decl`).

It is also possible to do a variety of common operations on the flow graph and statements: edge insertion, removal of statements and insertion of statements inside a block. Detailed information about the flow graph can be found in GCC's internal documentation.

## 5 Pass manager

Every SSA pass must be registered with the pass manager and scheduled in `init_tree_optimization_passes`. Passes are declared as instances of `struct tree_opt_pass`, which declares everything needed to run the pass, including its name, function to execute, properties required and modified and what to do after the pass is done.

In this context, properties refer to things like dominance information, the flow graph, SSA form and which subset of GIMPLE is required. In theory, the pass manager would arrange for these properties to be computed if they are not present, but not all properties are presently handled. Each pass will also declare which properties it destroys so that it is recomputed after the pass is done.

To add a new Tree SSA pass, one should

1. create a global variable of type `struct tree_opt_pass`,
2. create an `extern` declaration for the new pass in `tree-pass.h`, and,
3. sequence the new pass in `tree-optimize.c:init_tree_optimization_passes` by calling `NEXT_PASS`. If the pass needs to be applied more than once, use `DUP_PASS` to duplicate it first.

[3] It may be advantageous to keep the CFG all the way to RTL, so this may change in the future.

```
{
  basic_block bb;
  block_stmt_iterator si;

  FOR_EACH_BB (bb)
    for (si = bsi_start (bb); !bsi_end_p (si); bsi_next (&si))
      {
        tree stmt = bsi_stmt (si);
        print_generic_stmt (stderr, stmt, 0);
      }
}
```

Figure 1: Traversing all the statements in the current function.

# 6 SSA form

Most of the tree optimizers rely on the data flow information provided by the Static Single Assignment (SSA) form [2]. The SSA form is based on the premise that program variables are assigned in exactly one location in the program. Multiple assignments to the same variable create new versions of that variable.

Naturally, actual programs are seldom in SSA form initially because variables tend to be assigned multiple times. The compiler modifies the program representation so that every time a variable is assigned in the code, a new version of the variable is created. Different versions of the same variable are distinguished by subscripting the variable name with its version number. Variables used in the right-hand side of expressions are renamed so that their version number matches that of the most recent assignment.

This section describes how the compiler recognizes and classifies statement operands recognized, the process of renaming the program into SSA form and how is aliasing information incorporated into the SSA web.

## 6.1 Statement operands

Tree SSA implements two types of operands: *real* and *virtual*. Real operands are those that represent a single, non-aliased, memory location which is atomically read or modified by the statement (i.e., variables of non-aggregate types whose address is not taken). Virtual operands represent either partial or aliased references (i.e., structures, unions, pointer dereferences and aliased variables).

Since the SSA form uses a versioning scheme on variable names, in principle it would not be possible to assign version numbers to virtual operands. So, the compiler associates a symbol name to the operand and provides SSA versioning for that symbol. Symbols for virtual operands are either created or derived from the original operand:

- For pointer dereferences, a new symbol called a *memory tag* (MT) is created. Memory tags represent the memory location pointed-to by the pointer. For instance, given a pointer `int *p`, the statement `*p = 3` will contain a virtual definition to `p`'s memory tag (more details in Section 7).
- For references to variables of non-aggregate types, the base symbol of the reference is used. For instance, the statement `a.b.c = 3`, is considered a virtual definition for `a`. Other terms to refer to virtual definitions include "*may-defs*," when they refer to aliased stores, and

"*non-killing defs*" when they refer to partial stores to an object of a non-aggregate type. Similarly, virtual uses are known as "*may-uses*."

Using this scheme, the compiler can now rename both real and virtual operands into SSA form. Every symbol that complies with SSA_VAR_P will be renamed. This includes VAR_DECL, PARM_DECL and RESULT_DECL nodes. To determine whether an SSA_VAR_P will be renamed as a real or virtual operand, the predicate is_gimple_reg is used. If it returns true the variable is added as a real operand, otherwise it is considered virtual.

Every statement has 4 associated arrays representing its operands: DEF_OPS and USE_OPS hold definitions and uses for real operands, while VDEF_OPS and VUSE_OPS hold potential or partial definitions and uses for virtual operands. These arrays are filled in by get_stmt_operands. The code fragment in Figure 2 shows how to print all the operands of a given statement. Operands are stored inside an auxiliary data structure known as *statement annotation* (stmt_ann_t). That's a generic annotation mechanism used throughout Tree SSA to store optimization-related information for statements, variables and SSA names.

### 6.2 SSA Renaming Process

We represent variable versions using SSA_NAME nodes. The renaming process in *tree-into-ssa.c* wraps every real and virtual operand with an SSA_NAME node which contains the version number and the statement that created the SSA_NAME. Only definitions and virtual definitions may create new SSA_NAME nodes.

Sometimes, flow of control makes it impossible to determine what is the most recent version of a variable. In these cases, the compiler

```
void
print_ops (tree stmt)
{
  vuse_optype vuses;
  vdef_optype vdefs;
  def_optype defs;
  use_optype uses;
  stmt_ann_t ann;
  size_t i;

  get_stmt_operands (stmt);
  ann = stmt_ann (stmt);

  defs = DEF_OPS (ann);
  for (i = 0; i < NUM_DEFS (defs); i++)
    print_generic_expr (stderr,
                        DEF_OP (defs, i), 0);

  uses = USE_OPS (ann);
  for (i = 0; i < NUM_USES (uses); i++)
    print_generic_expr (stderr,
                        USE_OP (uses, i), 0);

  vdefs = VDEF_OPS (ann);
  for (i = 0; i < NUM_VDEFS (vdefs); i++)
    print_generic_expr (stderr,
                        VDEF_OP (vdefs, i), 0);

  vuses = VUSE_OPS (ann);
  for (i = 0; i < NUM_VUSES (vuses); i++)
    print_generic_expr (stderr,
                        VUSE_OP (vuses, i), 0);
}
```

Figure 2: Accessing the operands of a statement.

inserts an artificial definition for that variable called *PHI function* or *PHI node*. This new definition merges all the incoming versions of the variable to create a new name for it. For instance,

```
if (...)
  a_1 = 5;
else if (...)
  a_2 = 2;
else
  a_3 = 13;

# a_4 = PHI <a_1, a_2, a_3>
return a_4;
```

Since it is not possible to statically determine which of the three branches will be taken at runtime, we don't know which of $a_1$, $a_2$ or $a_3$ to use at the return statement. So, the SSA renamer creates a new version, $a_4$, which is assigned the result of "merging" all three other versions. Hence, PHI nodes mean "one of these operands. I don't know which."

Previously we had described virtual definitions as non-killing definitions, this means that given a sequence of virtual definitions for the same variable, they should all be related somehow. To this end, virtual definitions are considered read-write operations. So, the following code fragment

```
...
# a = VDEF <a>
a[i] = f ();
...
# a = VDEF <a>
a[j] = g ();
...
# a = VDEF <a>
a[k] = h ();
...
```

is transformed into SSA form as

```
...
# a_2 = VDEF <a_1>
a[i] = f ();
...
# a_3 = VDEF <a_2>
a[j] = g ();
...
# a_4 = VDEF <a_3>
a[k] = h ();
...
```

Notice how every VDEF has a data dependency on the previous one. This is used mostly to prevent errors in scalar optimizations like code motion and dead code elimination. Passes that want to manipulate statements with virtual operands should obtain additional information (e.g., by building an array SSA form, or value numbering as is currently done in the dominator optimizers). The SSA form for virtual operands is actually a factored use-def (FUD) representation [5]. When taking the program out of SSA form, the compiler will not insert the copies needed to resolve the overlap. Virtual operands are simply removed from the code.

Such considerations are not necessary when dealing with real operands. SSA_NAMEs for real operands are considered distinct variables and can be moved around at will. When the program is taken out of SSA form (*tree-outofssa.c*), overlapping live ranges are handled by creating new variables and inserting the necessary copies between different versions of the same variable. For instance, given the GIMPLE program in SSA form in Figure 3a, optimizations will create overlapping live ranges for two different versions of variable $b$, namely $b_3$ and $b_7$ (Figure 3b). When the program is taken out of SSA form, prior to RTL expansion, the two different versions of $b$ will be assigned different variables (Figure 3c).

```
foo (a, b, c)
{
  a_4 = b_3;
  if (c_5 < a_4)
    goto <L0>;
  else
    goto <L1>;

<L0>:
  b_7 = b_3 + a_4;
  c_8 = c_5 + a_4;

  # c_2 = PHI <c_5, c_8>;
  # b_1 = PHI <b_3, b_7>;
<L1>:
  return b_1 + c_2;
}
```

(a) Original SSA form.

```
foo (a, b, c)
{
  if (c_5 < b_3)
    goto <L0>;
  else
    goto <L1>;

<L0>:
  b_7 = b_3 + b_3;
  c_8 = c_5 + b_3;

  # c_2 = PHI <c_5, c_8>;
  # b_1 = PHI <b_3, b_7>;
<L1>:
  return b_1 + c_2;
}
```

(b) SSA form after optimization.

```
foo (a, b, c)
{
  if (c < b)
    goto <L0>;
  else
    goto <L1>;

<L0>:
  b.0 = b + b;
  c = c + b;
  b = b.0;

<L1>:
  return b + c;
}
```

(c) Resulting normal form.

Figure 3: Overlapping live ranges for different versions of the same variable.

```
foo (int *p)
{
  #   TMT.1_5 = VDEF <TMT.1_4>;
  *p_1 = 5;

  #   VUSE <TMT.1_5>;
  T.0_2 = *p_1;

  return T.0_2 + 1;
}
```

Figure 4: Representing pointer dereferences with memory tags.

## 7 Alias analysis

Aliasing information is incorporated into the SSA web using artificial symbols called *memory tags*. A memory tag represents a pointer dereference. Since there are no multi-level pointers in GIMPLE, it is not necessary for the compiler to handle more than one level of indirection. So, given a pointer $p$, every time the compiler finds a dereference of $p$ (`*p`), it is considered a virtual reference to $p$'s memory tag (Figure 4).

Given this mechanism, whenever the compiler determines that a pointer $p$ may point to variables $a$ and $b$ (and $p$ is dereferenced), a memory tag for $p$ is created and variables $a$ and $b$ are added to $p$'s memory tag.

The code fragment in Figure 5 illustrates this scenario. The compiler determines that $p_2$ may point to $a$ or $b$, and so whenever $p_2$ is dereferenced, it adds virtual references to $a$ and $b$. Also notice that since both $a$ and $b$ have their addresses taken, they are always considered virtual operands.

The compiler computes three kinds of aliasing information: type-based, flow-sensitive points-to and flow-insensitive points-to[4].

Going back to the code fragment in Figure 5, notice how the two different versions of pointer $p$ have different alias sets. This is because $p_1$ is found to point to either $c$ or $d$, while $p_2$ points to either $a$ or $b$. In this case, the compiler is using flow-sensitive aliasing information and will create two memory tags, one for $p_1$ and another

[4]This one is currently not computed by default. It is enabled with `-ftree-points-to=andersen`

```
foo (int i)
{
  ...
<L0>:
  # p_1 = PHI <&d, &c>;

  #   c_5 = VDEF <c_7>;
  #   d_18 = VDEF <d_17>;
  *p_1 = 5;

  ...
<L3>:
  # p_2 = PHI <&b, &a>;

  #   a_19 = VDEF <a_4>;
  a = 3;

  #   b_20 = VDEF <b_6>;
  b = 2;

  #   VUSE <a_19>;
  #   VUSE <b_20>;
  T.0_8 = *p_2;
  ...

  #   a_21 = VDEF <a_19>;
  #   b_22 = VDEF <b_20>;
  *p_2 = T.1_9;
  ...
}
```

Figure 5: Using flow-sensitive alias information.

for $p_2$. Since these memory tags are associated with SSA_NAME objects, they are known as *name memory tags* (NMT).

In contrast, when the compiler cannot compute flow-sensitive information for each SSA_NAME, it falls back to flow-insensitive information which is computed using type-based or points-to analysis. In these cases, the compiler creates a single memory tag that is associated to **all** the different versions of a pointer (i.e., it is associated with the actual VAR_DECL or PARM_DECL node). Such memory tag is called *type memory tag* (TMT).

Figure 6 is similar to the previous example, but in this case the addresses of $a$, $b$, $c$ and $d$ escape the current function, something which the current implementation does not handle. This forces the compiler to assume that all versions of $p$ may point to either of the four variables $a$, $b$, $c$ and $d$. And so it creates a type memory tag for $p$ and puts all four variables in its alias set.

The concept of 'escaping' is the same one used in the Java world. When a pointer or an ADDR_EXPR escapes, it means that it has been exposed outside of the current function. So, assignment to global variables, function arguments and returning a pointer are all escape sites.

We also use escape analysis to determine whether a variable is call-clobbered. If an ADDR_EXPR escapes, then the associated variable is call-clobbered. If a pointer $P_i$ escapes, then all the variables pointed-to by $P_i$ (and its memory tag) also escape.

In certain cases, the list of may aliases for a pointer may grow too large. This may cause an explosion in the number of virtual operands inserted in the code. Resulting in increased memory consumption and compilation time.

When the number of virtual operands needed

```
foo (int i)
{
  #   a_6 = VDEF <a_3>;
  #   b_4 = VDEF <b_5>;
  #   c_13 = VDEF <c_12>;
  #   d_15 = VDEF <d_14>;
  p_2 = baz (&a, &b, &c, &d);

  #   a_16 = VDEF <a_6>;
  #   b_17 = VDEF <b_4>;
  #   c_18 = VDEF <c_13>;
  #   d_19 = VDEF <d_15>;
  *p_2 = 5;

  #   a_20 = VDEF <a_16>;
  a = 3;

  #   b_21 = VDEF <b_17>;
  b = 2;

  #   VUSE <a_20>;
  #   VUSE <b_21>;
  #   VUSE <c_18>;
  #   VUSE <d_19>;
  T.1_7 = *p_2;
  ...
}
```

Figure 6: Using flow-insensitive alias information.

```
foo ()
{
  #   TMT.5_13 = VDEF <TMT.5_12>;
  p_2 = baz (&a, &b, &c, &d);

  #   TMT.5_14 = VDEF <TMT.5_13>;
  *p_2 = 5;

  #   TMT.5_15 = VDEF <TMT.5_14>;
  a = 3;

  #   TMT.5_16 = VDEF <TMT.5_15>;
  b = 2;

  #   VUSE <TMT.5_16>;
  T.1_7 = *p_2;
  ...
}
```

Figure 7: Effects of alias grouping.

to represent aliased loads and stores grows too large (configurable with `-param max-aliased-vops`), alias sets are grouped to avoid severe compile-time slow downs and memory consumption. The alias grouping heuristic essentially reduces the sizes of selected alias sets so that they are represented by a single symbol. This way, aliased references to any of those variables will be represented by a single virtual reference. Resulting in an improvement of compilation time at the expense of precision in the alias information.

Figure 7 shows the same example from Figure 6 compiled with `--param max-aliased-vops=1`. Notice how all four variables are represented by $p$'s type memory tag, namely *TMT.5*. Even references to individual variables, like the assignment $a = 3$ are considered references to *TMT.5*.

## 8 Conclusions and future work

Tree SSA represents a useful foundation to incorporate more powerful optimizations and

analyses to GCC. Although the basic infrastructure is already functional and produces encouraging results there is still much work ahead of us. Over the next few months we will concentrate on the following areas:

*Compile time performance*. Currently, Tree SSA is in some cases slower than other versions of GCC. In particular, C++ programs seem to be the most affected. Preliminary findings point to memory management inside GCC and various data structures that are being stressed.

Another source of compile time slowness are the presence of RTL optimizations that have been superseded by Tree SSA. While some passes have already been disabled or simplified, there still remain some RTL passes which could be removed.

*Run time performance*. In general, Tree SSA produces similar or better code than other versions of GCC. However, there are still some missing optimizations. Most notably, loop optimizations, which we expect will be incorporated soon.

In the immediate future, most efforts will be focused on stabilizing the infrastructure in preparation for the next major release of GCC. Although the framework is tested daily on several different architectures, there are still some known bugs open against it and there are some architectures that have not been tested or have had limited exposure.

We expect the infrastructure to keep evolving, particularly as new optimizations are added, we will probably find design and/or implementation limitations that will need to be addressed. We have tried to make the basic design sufficiently flexible to permit such changes without overhauling the whole middle-end.

## References

[1] D. Berlin, D. Edelsohn, and S. Pop. High-Level Loop Optimizations for GCC. In *Proceedings of the 2004 GCC Summit*, Ottawa, Canada, June 2004.

[2] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[3] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 406–420. Lecture Notes in Computer Science, no. 457, Springer-Verlag, August 1992.

[4] J. Merrill. GENERIC and GIMPLE: A New Tree Representation for Entire Functions. In *Proceedings of the 2003 GCC Summit*, Ottawa, Canada, May 2003.

[5] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Reading, Mass.: Addison-Wesley, Redwood City, CA, 1996.

# Register Rematerialization In GCC

*Mukta Punjani*
Cadence Design Systems India Pvt Limited
muktap@cadence.com

## Abstract

In most modern processor architectures, difference in data access time for values kept in registers as compared to those in memory is quite high. Thus, compilers should implement efficient register allocation strategies to improve the runtime performance of the code. Register rematerialization is a technique to improve register allocation effectively by improving spill code generation. It is often desirable to compute expressions at a "use" rather than use an earlier spilled value. Normally, "rematerializable" values are derived from registers that are live throughout the function. On register-starved architectures with with addressing modes supporting limited displacement, spilling values, which can be rematerialized, incurs an additional loss in performance due to instructions generated to fetch data from the frame. Hence, rematerialization aids in good usage of registers to give a good gain in execution performance of the code generated. Experimental results indicate a gain of 1-6% in code size and 1-4% improvement in execution performance.

## 1 Introduction

Let us see the register rematerialization (remat) concept in GCC in more detail. We discuss the proposed improved remat implementation in GCC as it occurs as a part of graph coloring register allocator (in the new regalloc branch). This optimization is supported by target architecture hooks and is currently implemented and tested for SH4 architecture.

### 1.1 What is remat?

Certain values in a function can be recomputed at any point, as the required source operands will always be available for the computation. Such values are called never killed values. During global register allocation pass, if such never killed values cannot be kept in registers and need to be spilled, the register allocator should recognize when it is cheaper to recompute the value i.e. to rematerialize it [REMAT], rather than to store and reload it from stack. This often happens with frame pointer (FP) relative address computations as well as address computations of large struct or arrays in local scope, where unnecessary spills are seen. A prime example of this can be cited:

1. GCC calculates FP + offset and stores into r3 (say).
2. GCC spills and restores r3 to (from) stack even though it would be cheaper to clobber the register and recompute the value of FP + offset.

Register remat occurs as part of a larger problem of improved spill code generation during global register allocation. The description below co-relates the two facets—Register Allocation Problem and remat as a method of improved spill code generation.

### 1.2 Register Allocation as Graph coloring Problem

The new register allocator in GCC models register allocation as a graph-coloring problem. It first constructs an interference graph G where nodes in G represent live ranges and edges represent *interferences*. So there is an edge from node $i$ to node $j$ if and only if live range $l_i$ interferes with live range $l_j$ i.e. they are simultaneously live at some point and hence cannot occupy the same register. Live ranges that interfere with $l_i$ are its neighbors in the graph, the degree of $l_i$ is the number of neighbors it has in the graph.

To find an allocation from G, the compiler looks for a k-coloring of G, i.e. an assignment such that neighboring nodes always have distinct colors. If we choose k to match the number of machine registers, then we can map a k-coloring for G into a feasible register assignment for the underlying code. Because finding a k-coloring of an arbitrary graph is NP-complete, the compiler uses a heuristic method to search for a coloring, it is not guaranteed to find a k-coloring for all k-colorable graphs. If a k-coloring is not discovered, some live ranges are spilled, i.e. the values are kept in memory rather than in registers [GCRA].

Spilling one or more live ranges changes both the intermediate code and the interference graphs; hence register allocation. The compiler proceeds by iteratively spilling live ranges and attempting to color the resulting new graph. This process is guaranteed to terminate.

The new register allocator framework takes two approaches while spilling a live range

- A simple *Spill Everywhere* approach involves spilling the entire live range in case it needs. This would involve spilling all the defs of value live range is representing on stack and inserting corresponding reloads before all uses these defs flow into. This spilling technique is fast but not optimal as it would generate lot of spill code.

- An improved but slower *Interference Region Spilling* approach, which involves spilling a live range partly. An interference region for two live ranges can be defined as the portion of the data flow where they are live simultaneously. By spilling interference region for one of the live ranges, they will no longer be live simultaneously, thus will no longer interfere. This effectively removes an edge between the two nodes in the interference graph, making the graph more easily colored. Any spill code addition due to interference region spilling would insert spill code say after a particular use point (only those uses which lie in the interference region). This use point may or may not be the first one in the live range.

### 1.3 Improving the quality of spill code—Remat Opportunities

In the existing implementation of new register allocator, remat is performed only for those values whose definition consists of moving a immediate value to a pseudo. The proposed approach can enhance the scope of remat by taking into account more potential remat candidates and hence improvement in spill code generation.

The opportunities identified for remat can be

- Immediate loads of integer/float constants
- Loads from literal pools
- Computing a constant offset from the stack pointer

- Computing a constant offset from the static data area pointer
- Any Branch Related target Labels
- Address computations for access to non-local names in case of nested functions
- Address computations from pointer to global offset table in case of position independent code
- Address computations resulting from parameter registers (e.g. r4...r7 in case of SH4) or parameter register copied to callee save registers which dont have defs elsewhere in the function
- Address computations resulting from return register (e.g. r0 in case of SH4) or return register copied to other callee save registers which don't have defs elsewhere in the function

# 2 Remat Strategy

The new register allocator is based on the analysis of definitions and uses of all the pseudos in the instructions stream to form webs (nodes of the interference graph), which are the basic entity for allocation to a register. Hence, each web corresponds to the live range of a variable, which can be allocated a distinct symbolic register number. The interference graph for webs consists of a number of such intersecting webs, the intersection between any two webs occurs when they have a use in common. If a web can't be assigned a register then a decision is made to spill it.

The proposed improved remat optimization consists of identifying remat defs during dataflow analysis and propagating this information to the web spilling phase. The spilling phase can choose between spilling or rematerialization of a web based on relative cost analysis. Hence the remat strategy can be segregated into the following sub-problems

- Gather and propagate remat/live-range information
- Criterion for spilling/remat decision
- Performing remat

### 2.1 Remat Information

To identify rematerializable candidates, remat information needs to be built during the dataflow analysis. All the defs can be analyzed to see whether they come from any of the remat sources. Such defs can be tagged with their corresponding remat efinitions. Any remat definition resulting from an operation on two or more rematerializable definitions (say a def consisting of adding a constant value to the label) can be tagged likewise.

### 2.2 Remat Criterion

The obvious criterion for remat/spilling decision would be to compare the relative cost of both the decisions in terms of aggregate cost of instructions each would generate, and choose the one with lower cost. This criterion is described in Section 2.3. But due to some issues mentioned further, it might not be possible to calculate the exact spill costs. Another method to choose remat in the absence of spill cost criterion has been described in Section 2.4.

### 2.3 Spill v/s Remat Cost

Whenever a decision is taken to mark a web for spilling, check if the definition in the web is rematerializable. Calculation of the remat and spill cost will be implemented in a target dependent hook. Remat cost will be calculated in terms of the aggregate of all the insns' costs,

which would be required to recompute a remat definition before every use of such definition. Similarly spill cost can be computed as instruction cost aggregate of all the insn's cost generated to spill (and restore) the value. Choose the one with lower cost. However in the existing new register allocator, it may be difficult to calculate the exact accurate remat (spill) costs of any value because

- During web spilling, actual stack offsets of spill locations are not defined, instead only stack pseudos are assigned
- The `reload_cse` pass may use address inheritance information and may merge some instructions for loading the offset, by reusing any value close to or equal to the offset loaded in some other pseudo. Hence cost decisions may get invalidated later

e.g. In case of SH4, the number of instructions to spill (restore) can be a minimum of 1, if spilling takes place at an offset less than 64 bytes relative to a base register. In case the spill offset is more than 64 bytes, spilling (restoring) may take two or more instructions, one instruction for loading the stack offset to spill (restore) and another consisting of storing (restoring) the value. In order to calculate spill (restore) offset (and the number of instructions required for spill ), stack slot information to which the pseudo is likely to be spilled to, needs to be built and tracked for all such pseudos, depending on which the number of instructions needed to spill (restore) can be calculated.

The `reload_cse` pass may further merge instructions for loading the offset, by reusing any value close to or equal to the offset loaded in some other pseudo. In this case, even if the spill location is at an offset greater than 64 bytes, it may require 1 instruction.

The first problem related to spill cost computation can be partially resolved by tracking the size of frame data and number of pseudos being spilled so as to have an estimate of spill offsets. This can predict the number of instructions that would actually be required to spill (restore). However in the existing framework of new register allocator, spill cost cannot be computed correctly in some cases due to `reload_cse` issue mentioned.

### 2.4 Remat Without Spill Cost Calculation

In the absence of a definite cost available for spilling, spill cases can be segregated according to the reason for their occurrence and also cases, which will definitely be cheaper to rematerialize than to spill

**Definitive Remat** Some defs can be identified as remat cases based on the fact that they require 1/2 insns to compute and re-computing them will always be less than or equal to minimum cost of spilling a def for the given target. Examples for such definitive remat would include

- Constant loads
- Label Loads

Defs having up to 1 insn in their remat insn chain (spill (restore) together would require a minimum of two insns).

For defs having two or more insns in their recomputation sequence, insn merging can be attempted on that sequence based on a target dependent hook. If such sequence merging generates a valid single insn for the target, then it fits as a candidate for definitive remat.

For remat in such cases, all the insns leading to definition of rematerializable value being spilled can be moved immediately before its use.

**Moving Defs Near Uses** This approach may be used for remat cases not covered in previous section. Spilling generally happens as a result of spacing between the actual def and use of values in case of high register pressure. This can happen in case of complex computations involved in some program statements, where LHS computations and some other intermediate computations have to be spilled. Such cases of spilling require generation of 2–4 spill and restore instructions (e.g. 64-byte stack offset criterion in SH4). In case the computation being spilled is rematerializable, spilling becomes unnecessary. The remat cost criterion need not be taken into account here because spilling is definitely not required.

In both the approaches for spilling described above, unnecessary spill cases can be identified as those having their spill point just after def (before 1st use).

The following assembly illustrates this case:

```
chanserv.i/load_cs_dbase (in
stress 1.17) compiled with -O2
-ml -m4 -fnew -ra
-fno-argument-alias
-fno-schedule -insns
-fno-schedule-insns2 -g -S
-fpic generates

.L1404:
  .loc 1 2844 0
  mov.l   .L1036,r0   <-- 1
  mov.l   @(r0,r12),r0<-- 2
  mov.w   @r0,r1      <-- 3 (remat
                    insn chain 1,2)
  mov.w   r1,@r14     <-- Spill
                 before 1st use
.L612:
  .loc 1 2845 0
  mov.w   .L1037,r0
  mov.w   @r14,r1     <-- Reload
  mov.w   r1,@(r0,r11)
  .loc 1 2848 0
  mov     r14,r4
  mov.l   .L1038,r1
  bsrf    r1
```

For remat in such cases, the all the insns leading to definition of rematerializable value being spilled can be moved immediately before its use.

There are certain issues regarding the movement of insns in this case. The placement of a def before use requires addition of insns before the use point which leads to increased register usage there. Hence, a good heuristic needs to be devised to make sure that such insn insertion keeps the register pressure in check and does not actually end up increasing it.

**Interference Region Spills** As discussed earlier, in case of interference region spilling, spill point may or may not be before the first use point. Hence in such a case for spilling, we have to choose between spill/rematerialize (in the absence of cost of spilling). This case of spilling generally occurs in spilling calculated offsets for arrays/structs, and may not require a lesser remat cost in most of the cases. Here rematerialization decision is not taken, as spilling might actually be cheaper. Nevertheless, the decision may not be correct for all the cases.

### 2.5 Performing Remat

Remat of a value involves inserting recomputation sequence for a definition before use points by moving the insns forming the definition before use points.

## 3 Implementation in GCC

The patch at the link

`http://gcc.gnu.org/ml/gcc-patches/2003-12/msg01985.html`

is the implementation of the ideas presented in the paper in GCC. This implementation scope and its constraints are discussed as follows.

### 3.1 Current Implementation Approach

**Data Flow Phase** The improved remat handler detemines the defs, which have been generated from never-killed sources and creates a remat pattern sequence to recompute those defs. This code is implemented along with the data flow routines in df.c

```
Eg. r159 <- const1      (1)
r160 <- r14 +  r159     (2)
r161 <- mem (r160)      (3)
```

Here defs for r159, r160, r161 are all remat, as all have been derived from never-killed sources. The data flow routines determine all such defs and store the respective patterns that would be required for recomputing each potential remat definition.

So in the above example,
def(r159)->remat_sequence = (1)
def(r160)->remat_sequence = (1), (2)
and def(r161)->remat_sequence = (1), (2), (3)

**Web Construction Phase** The cost of webs in `ra_build.c` is modified to accommodate the cost of those webs which have rematerializable defs. The cost for all the defs is added up whether for remat defs or non-remat ones.

**Web Colorize Phase** The webs, which have potentially, remat defs and the cost consequently is lower than other defs are promoted for spilling. This advocates their spilling, in turn relieving the conflict edges. Remat handler later picks up such webs and appropriate processing is done there. This happens in `ra_colorize.c`

**Web Spilling Phase** In this file i.e. `ra_rewrite.c`, the allocator is let to spill as it was doing earlier. After the first level spilling is done, the spilled webs whose defs can be remat are picked up and the remat patterns for those defs can replace the restore insns for the defs spilled. At the point of inserting computations, it needs to be ensured that the regs that form the remat sequence

- Do not increase the register pressure at that point.
- Live range of those regs is not exceeded.

The remat handler only concentrates on the first pass of the allocator and spills generated for the first time are handled only. Later rounds do not call the remat sequence building routines.

**Problems Encountered With Full Scope**

- In the absence of a good register pressure estimation heuristic, insertion of defs with multiple insns in the remat sequence poses problems. Also, the register allocator has strong asssumpions about the web structure. Hence after inserting recomputation patterns of length $> 1$ in place of the restore insn, the allocator got stuck up in a lot of in tight consistency checks of `ra_build.c` especially in `parts_to_webs_1`.
- Due to the same problem, function pointer and return register are not being considered for remat.

**Current Implementation Scope** Due to the problems cited above, the current implementation implements the following remat handling:

- The data flow phase constructs remat sequences in full and then tries to collapse

them (if allowed by the target) in `df_remat_validate`. Rest of the chains are discarded at this point.

- The build phase involving modifying costs for remat webs functions as before.
- The colorize phase functions as before.
- The rewrite pass replaces the source of restore insns before the spilled uses of the webs and replaces it with the remat source (a single pattern).

The restricted implementation is scalable enough to support the original concept of remat envisaged and should stand any changes in the register allocator passes.

## 4 Performance Data

The performance improvement due to register rematerialization depends on the following factors.

1. Register Pressure and consequently number of values spilled.
2. The values spilled from rematerializable sources and found to be obeying the constraints for performing remat

During performance analyis, select benchmarks were compiled using GCC-2.3 20021119 (new-regalloc branch) for SH4 target using options `-O2 -ml -m4 -static`. A new option namely `-fimproved-remat` is introduced to enable improved remat. The benchmarks were executed on SH4 evaluation boards with QNX 6.1. It is observed that best performance improvement for execution performance is 4% and that for code size is 6.46%. Table 1 gives code size comparisons of stress1.17 files with and without improved remat. The execution results for some benchmarks are shown in Table 2.

| File Name | size with new-ra | size with improved-remat | decrease (%) |
|---|---|---|---|
| L3bitstream.o | 7424 | 6944 | 6.46 |
| aiunit.o | 18880 | 17760 | 5.93 |
| scanline.o | 2400 | 2272 | 5.30 |
| advdomestic.o | 8280 | 7960 | 3.86 |
| tif_fax3.o | 10208 | 10176 | 3.13 |
| tif_packbits.o | 1316 | 1284 | 2.43 |
| layer3.o | 20752 | 20272 | 2.31 |
| melee2.o | 27516 | 26940 | 2.09 |
| navion_aero.o | 1600 | 1568 | 2.00 |
| chanserv.o | 63296 | 62112 | 1.87 |
| s_serv.o | 29740 | 29196 | 1.82 |
| im_decode.o | 290000 | 285936 | 1.40 |
| quantize.o | 10012 | 9948 | 0.64 |
| wizard1.o | 24064 | 23964 | 0.41 |
| blowfish.o | 8808 | 8776 | 0.36 |
| map_fog.o | 26272 | 28880 | -9.6 |

Table 1: Code Size Comparisons

| Benchmark | Input Data Size | Gain (%) |
|---|---|---|
| GZIP Compression | 80.5 MB | 4 |
| Mpg 123 | - | 4 |
| GZIP Decompression | 16.2 MB | 2.6 |
| GSM Compression | 1.71 MB | 0.05 |
| GSM Decompression | 361 KB | 0 |

Table 2: Execution Timings

## 5 Further Improvements In New RA

### 5.1 Loop Variable Spilling

Ideally, register allocation should take into account, the variables present inside the loops and in case of high register pressure, try to assign registers to frequently accessed variables in a loop (for example loop indexes) on a priority basis. But such an allocation scheme is not being observed in some cases. The following example illustrates this fact

`chanserv.i/check_modes compiled`

```
with -O2 -ml -m4 -fnew-ra -fno-
argument-alias -fno-schedule-insns
-fno-schedule-insns2 -g -S:

  .loc 1 4743 0
  mov.l   .L2970,r7
  mov.b   @r7,r1
  cmp/pl  r1
  bf/s    .L2942
  and     r0,r3
  mov     #0,r1       <-- (1)
  mov     #64,r0      <-- (2)
  mov.l   r1,@(r0,r14)<-- (3)
  mov     #0,r2
  .loc 1 4745 0
  mov     #64,r0
.L3043:
  mov.l   @(r0,r14),r6   <-- (4)
  add     r7,r6          <-- (5)
  mov.l   r6,@(r0,r14)   <-- (6)
  mov     r6,r0
  mov.l   @(4,r6),r6
  tst     r6,r3
  bt      .L2922
  mov.b   @r0,r1
  .loc 1 4747 0
  mov.b   r1,@r12
  add     #1,r12
  .loc 1 4748 0
  not     r6,r6
  mov.l   @(56,r10),r1
  and     r6,r1
  mov.l   r1,@(56,r10)
  .loc 1 4743 0
.L2922:
  add     #8,r2         <-- (7)
  mov     #64,r0        <-- (8)
  mov.l   r2,@(r0,r14)  <-- (9)
  mov     r2,r0
  mov.b   @(r0,r7),r1
  cmp/pl  r1
  bt/s    .L3043
  mov     #64,r0
```

In the above example, the calculation corresponding to register r1 in (1) is being spilled within a loop instructions (4) through (9). Such an example clearly illustrates inefficient register allocation.

### 5.2 Loop Invariant Code Spilling

In case of any loop, invariant part of the code is moved outside the loop. In some cases such address computations might be spilled onto a stack locations outside the loop. Inside the loop these values are reloaded from stack. Such cases are NOT direct candidates of remat (as generally remat cost will be higher than the spill cost). However in case the computation requires single instruction within the loop (e.g. loads within 64 byte window to a base register i.e. r14, r12, r11 etc.) then it should well be computed inside the loop instead of being moved out as invariant code. This would result in saving one instruction per loop iteration.

However, changes required in this case would involve GCC passes, which move loop invariant code.

## 6 Acknowledgments

I would like to thank the GCC community, especially Michael Matz, Denis Chertykov, and Vladimir Makarov, for their ideas. I would also like to thank Sandeep Khurana and Naveen Sharma from HCL Technologies and Toshiyasu Morita from Renesas for their valuable ideas and suggestions.

## References

[SRC] Mainline Sources of GNU GCC 3.3

[GCC] GCC Internals Manual. `http://gcc.gnu.org`

[GCRA] Briggs, P., Cooper, K. D., and Torczon, L. 1994. *Improvements to Graph Coloring Register Allocation*

[REMAT] Preston Briggs, Keith D. Cooper, Linda Torczon *Rematerialization*

# Addressing Mode Selection in GCC

*Naveen Sharma*
HCL Technologies Ltd.
naveens@noida.hcltech.com

*Rakesh Kumar*
HCL Technologies Ltd.
rakeshku@noida.hcltech.com

## Abstract

GCC has no formal addressing mode selection mechanism. It uses target hooks to generate valid addressing modes for a target. However, a significant amount of high level information is destroyed while doing this, especially for targets lacking a rich set of addressing modes. This leads to poor aliasing, and subsequently poorer CSE, GCSE, and scheduling. Hence, an unoptimal object code. This paper proposes an abstraction over RTL to generate machine independent addressing modes to achieve better aliasing. The actual addressing modes of the target are exposed after the first scheduling pass, where they are selected based on current execution scenario. Inter block address inheritance is also done at this point. The idea can be extended to specify a general "mid-level" RTL for GCC.

## 1 Introduction

### 1.1 Addressing Modes

In simple terms, addressing modes specify the way instruction operands are chosen at run time. In most general purpose machines, an addressing mode can specify a location in memory, a register or a constant/literal. This paper talks about addressing modes mainly in context of memory loads and stores i.e. operations which move data between registers and memory. In both operations, the destination gets affected; the source is not changed. When discussing about memory, the effect of Memory Management Unit can be ignored, since we are concerned with the addresses generated by the compiler.

Architectures have wide variance of features while considering addressing modes of a machine. Every processor, based on its application domain, has its unique set of addressing mode features. This choice is usually a function of various parameters like register set, instruction size and alignment restrictions. The most common addressing modes for loads/stores on a typical RISC architecture are:

- Displacement addressing mode: It is used when data is at known offset from some base address in a register.

```
mov.l @(4, r1),r2
```
[1]

- Register Indirect addressing mode: It is used when memory address of the required data is taken from register.

```
mov.l @r1,r2
```

- Register Index addressing mode is used when the exact offset from a base address is not known.

```
mov.l @(r0,r1),r2
```

[1]The assembly snippets correspond to SH4 processor.

- Auto Increment/Decrement modes: They combine memory access and address arithmetic.

```
mov.l @r1+,r2 !post-inc
mov.l r1,@-r2 !pre-dec
```

Some processors also allow pc-relative loads, where effective address is relative to the current point of execution. The above mentioned modes can also occur with some restrictions e.g. the SH4 processor allows only 4-bit displacement in displacement addressing mode.

### 1.2 Addressing Mode Selection

The compiler owns the responsibility of producing optimized code that exploits the features of target processor. The optimal choice of addressing modes aims at reduced code size and increased performance. GCC traditionally uses `RTL` as its intermediate language. Although `RTL` representation is machine independent, the `RTL` actually generated for a target is machine dependent. This is because `RTL` is generated directly from information in the machine description file. The machine description contains the description of exact instruction set of the target. The `RTL` can therefore be described as low-level intermediate form(or target `RTL`). This form is not very suitable for several high level/mid-level optimizations. The `tree-ssa` work overcomes the difficulty to a large extent by defining a new high level intermediate form. But some sort of mid-level `RTL` would is desirable for effective optimizations by GCC's `RTL` optimizer. In one sense, the notion of infinite pseudo registers can be considered a mid-level `RTL` abstraction.

We propose that addressing modes can also be abstracted as part of mid-level `RTL`. The advantages would be:

- Several initial `RTL` optimization passes would be able to perform better.

- Addressing Mode Selection based on execution scenario is likely to be better as address arithmetic is reduced.

### 1.3 Address Inheritance Problem

Every addressing mode has its associated cost. This cost could be evaluated in terms of pipeline characteristics of the processor, the instructions involved in address arithmetic, or the cost imposed by the target design. The concept of address inheritance encourages the reuse of address calculations. It states that wherever possible, the side effects of address arithmetic instructions should be carried forward, so that there are no recalculations at the point of next load/store operation. It looks similar to `CSE/GCSE` but there is a subtle difference. `CSE/GCSE` look at exact expressions.They do not know the relationship between two expressions of the form `reg+k1` and `reg+k2`, where `k1, k2` are constants. The fact that these can possibly be derived from each other in target specific way is out of scope for their functionality.

Address inheritance can be viewed as function with two parameters—time and space. Spatially related addresses are those which do not alias and which can be accessed from the same base without address arithmetic. The temporal local addresses are those which do not alias and which are separated by minimum number of instructions. Both can assume two attributes—`near` and `far`. Spatially related addresses represents a range of addresses allowed in reg+displacement addressing mode. Temporal relation is determined by number of available registers and control flow graph.

| Time | Space | Inherit(Yes/No) |
|---|---|---|
| near | near | Yes |
| near | far | Sometimes Possible |
| far | near | Register Pressure Issues |
| far | far | No |

Architectures with relatively more number of registers are potential candidates for "far & near" combination, whereas architectures having more number of bits reserved for offset in displacement addressing mode are potential candidates for "near & far" combination.

## 2 Problem Description

The machine description files are used to generate target-IL at the time of RTL generation. As already emphasized, though RTL representation is machine independent, but its generation is machine dependent. GCC imposes the restriction that every pass should generate valid target-RTL. This strategy hampers the addressing mode optimization, since subsequent passes are more restricted.

### 2.1 The Current Scheme

In the current situation, GCC relies on several target macros. It uses GO_IF_LEGITIMATE_ADDRESS to verify all memory address related changes. During RTL generation, the macro LEGITIMIZE_ADDRESS, is used to break large offsets to valid target-RTL form. When using one addressing mode, GCC queries whether the chosen mode is too expensive for the target. It uses the target hook TARGET_ADDRESS_COST to compute the cost of an addressing mode. Targets define TARGET_ADDRESS_COST as simple heuristic values. The hook exhibits a limited form of cost model for addressing mode choice, but it is not a complete framework and certainly misses optimal choice in most cases.

```
{
  i = 234;
  a[i] = 12123;
  ...
  i = 290;
  a[i] = 12123;
  ...
  i = 236;
  a[i] = 12123;
  ...
  i = 228;
  a[i] = 12123;
  ...
}
```

Figure 1: Random Accesses in an Array

```
      mov.w .L3,r1
      mov.w .L4,r0
      mov.l r1,@(r0,r14)
      mov.w .L5,r0
      mov.l r1,@(r0,r14)
      mov.w .L6,r0
      mov.l r1,@(r0,r14)
      add #-32,r0
      mov.l r1,@(r0,r14)
.L3:
      .short 12123
.L4:
      .short 936
.L5:
      .short 1160
.L6:
      .short 944
```

Figure 2: Output without AMS for Figure 1

Figure 1 illustrates some aspects of addressing mode selection problem.

Figure 2 shows the code generated by current implementation of GCC. There are few points noteworthy here:

- The value of *i* is known at compile time,

```
1       mov.w   .L3,r1
2       mov.w   .L4,r2
3       add     r14,r2
4       mov.l   r1,@(24,r2)
5       mov.w   .L5,r0
6       mov.l   r1,@(r0,r14)
7       mov.w   .L6,r0
8       mov.l   r1,@(32,r2)
9       mov.l   r1,@r2
10  .L3:
11     .short 12123
12  .L4:
13     .short 912
14  .L5:
15     .short 1160
```

Figure 3: Output with AMS for Figure 1

but copy propagation could not take advantage of it due to target restrictions.

- Since GCC assumes *i* is not known at compile time, it has chosen register index addressing mode. It could have done better as Figure 3 shows.

- There are three related addresses[2] in the snippet, viz. `a[228], a[234], a[238]`; but GCC is unable to recognize the fact.

- Extra pc-relative loads are generated.

The optimal assembly for the above snippet is shown in Figure 3. The line numbers are not part of assembly; these are kept for further reference.

With this improvement, even in this trivial example, we get 4 bytes of code size reduction, lesser stress on `r0`, the only index register available on SH4, and one fewer PC-rel load.

[2]The notion of related addresses is explained in the next subsection.

The selection of the optimal addressing modes with minimal code size and minimal execution time depends on many parameters and is NP complete in general[Eckstein]. One important criteria for choosing appropriate mode is the execution scenario. The choice which seems to be best in one scenario may prove to be unoptimal in another execution sequence. For example in Figure 3, dual register indirect addressing mode is used in line 6. Note that `r0` suffices many needs on SH4, and it is generally advisable to avoid the use of `r0` wherever possible. Still, this mode is the best choice in this execution sequence. The other choice left is register-indirect which would generate the sequence

```
mov.w   .L5,r3
add     r14,r3
mov.l   r1,@r3
```

The former choice is better since it is saving one address arithmetic instruction. The above example shows the choice of addressing mode should be determined by the execution scenario. Hence, it should be decided flexibly, and not rigidly as done in GCC currently.

### 2.2 Address Inheritance in GCC

GCC implements address inheritance in limited form through two passes—`regmove` and `reload_cse`. `regmove` intents register to register copy elimination. As a side effect, it does the following transformation:

```
pX<-pA+N |   pX <- pA + N
...      |->   ...
pX<-pA+M |   pX <- pX + (M - N)
```

This transformation is an address inheritance transformation as the the address computed in `pX` is reused subsequently. `regmove` is ineffective in several cases because:

- CSE and GCSE both run before regmove, and they attempt to optimize address arithmetic prior to regmove. They pull address calculations near basic block boundaries where regmove cannot optimize them.
- regmove pass cannot see beyond basic blocks and is unable to propagate information across basic blocks.
- regmove is able to do the required transformation only for SImode accesses for SH4.

reload_cse is simple CSE pass over hard registers after reload. The functions of reload_cse include:

1. It eliminates no-op moves where two different registers are assigned to the same hard register, and then copied one to the other.
2. It detects cases where we load a value from memory into two registers, and changes it to simply copy the first register into the second register if memory is more expensive than registers.
3. It scans the operands of each instruction to see whether the value is already available in a hard register. If possible, it replaces the operand with the hard register.

### 2.3 Alias Analysis

Several passes need alias information for doing effective optimizations. Alias information is most important for passes like CSE, loop invariant code motion, instruction scheduling, and register allocator. GCC can successfully determine aliasing between two memory references if they

```
void foo (float *a, float *b)
  {
    a[17] = a[0] + a[18];
    b[17] = b[1] + a[18];
  }
```

```
mov    r4,r2
add    #72,r2
fmov.s @r2,fr2 !Load a[18].
mov    r4,r3
fmov.s @r4,fr1 !Load a[0].
add    #68,r3
fadd   fr2,fr1 !Add.
fmov.s fr1,@r3 !Store a[17].
fmov.s @r5,fr1 !Load b[0].
fmov.s @r2,fr2 !Load a[18] again.
fadd   fr2,fr1
fmov.s fr1,@r1 !Store b[17].
```

Figure 4: The Alias problem

- use distinct constant offsets from the same register
- one of them points to stack

For machines that do not have "reg + displacement" addressing mode, pointer arithmetic is necessary to compute a pointer to the desired address. GCC lacks the mechanism to determine aliasing between such computed pointers[Sanjiv]. Consider the code in Figure 4. The Figure 4 also shows the corresponding SH4 assembly with -O2 option.

Since SH4 doesn't support "reg + displacement" addressing mode for floats, GCC alias analysis mechanism fails. Hence CSE is unable to determine if a value can be retained in a register across a write and a[18] is loaded twice.

# 3 Solution Strategy

### 3.1 Designing an Abstraction over RTL

It is desirable to have some sort of abstraction to hide target addressing modes to eliminate problems highlighted in the previous sections. We initially pretend some standard high level addressing modes. The change to target's addressing mode is done in a separate pass after first scheduling pass. The scheduler can do better load/store scheduling with abstract modes. There is a new macro called `TARGET_USE_ABSTRACT_MODES`. If this is nonzero, this will force the front end to generate memory references with following abstractions.

- Infinite displacement (natural register size) for register+offset addressing mode.
- Dual register indexed mode with two general purpose pseudos—i.e., @(rm, rn)—is supported.
- Auto-ionic modes are disabled as they effect the scheduling adversely.

### 3.2 Addressing mode selection pass

The addressing mode selection pass (AMS) lowers mid-level `RTL` to a low-level `RTL` by imposing target's constraint on addressing modes. At the same time, it would generate the required arithmetic. Address inheritance is part of the functionality of the AMS pass.

**Virtual Displacement Handling:** The transformation of infinite virtual displacements to target specific displacements is done as follows. The pointer pseudo is given the following attributes:

1. The bias of a pointer is the value currently added to the base pointer.
2. The mode of a pointer is the mode in which the register is accessed or used.
3. The slack of a pointer is the maximum negative value, which can currently be added to the pointer and still properly address the memory references which have already been assigned to this pointer.

The algorithm also defines a `locked_pool` of pointer pseudos which contains bias values at a specific execution point. A locked register is a register which is usable for an address within the currently visible lookahead window without any bias changes. The look ahead window is normally a basic block. We also define `unlocked_pool` registers with each register's bias. An unlocked register is a register which is currently not usable for an address within the currently visible lookahead window without any bias changes. E.g., consider the reference sequence with addresses:

```
(plus:Pmode (fp,124))
(plus:Pmode (fp,120))
(plus:Pmode (fp,128))
```

where `fp` is the frame pointer. It can be any base register in general. Initially, a new pseudo (say `rn`) is created with a `<bias, slack>` value pair as `<124, 60>` for SH4. We can then access memory at (fp, 124) simply as `(rn, 0)` with displacement addressing mode. At second access, (fp, 120), we note that we can reuse `rn` with a bias change of 4. So we change the `<bias, slack>` value to `<120, 56>`. When an offset is not reachable with any pseudo in the locked pool, then a new pseudo (say `rn+1`) is created.

By applying the above reasoning the following output is generated for SH4(which has 60 byte valid displacement):

```
mov     #120,r2
```

```
add     r14,r2     !r14 is fp
mov.l   @(4,r2),r1 !fp+124
mov.l   @r2,r3     !fp+120
..
mov.l   r1,@(8,r2) !fp+128
```

With current framework GCC ends up generating code like this for SH4.

```
mov     r14,r2
add     #64,r2
mov.l   @(60,r2),r1
mov.l   @(56,r2),r2
...
mov     r14,r2
add     #124,r2
mov.l   r1,@(4,r2)
```

The address arithmetic is reduced in former case. To avoid creating too many pseudos during the process, some heuristics have been tried. Limiting pseudos to approximately half of the register set usually turns out be good. With a proper register rematerialization framework `new-ra`, limiting pseudos might become un-necessary.

**General index register mode:** We can tackle this mode in two ways depending on architecture features and register pressure. There can be weird limitations on use of index registers. While in common cases, the index registers form a `REGISTER_CLASS`, there may be cases like SH4 where `r0` is the sole legal index register. Excessive pressure build up on `r0` as ABI specifies it as a return value register too. So in many cases, it is simply desirable to convert from index register mode to register index mode. The register pressure estimation is still in experimental phases, and forms part of several other problems in compiler technology. We use very simple register pressure heuristics based on machine modes of register. The results would be updated once a general infrastructure for register estimation can be implemented.

**Inter-Block Address Inheritance:** The technique described needs to be extended to retain `<bias, slack>` values across basic blocks. Taking control flow graph into account is a difficult problem. For simplicity, we propagate the `locked_pool` information only to `fallthru` basic blocks. So some address calculations are saved across basic blocks. The overall strategy is still in investigative phase.

## 4 Conclusion

Implementation of the ideas presented here have confirmed the expected aliasing gains. The implementation has been tested for SH4 and IA-64. Preliminary benchmarking indicate that execution gains can be as high as 5-7%. However, some more work is required for the idea to work on CISC machines.

## 5 Acknowledgements

We owe thanks and credit to Toshiyasu Morita (Renesas Technologies). Much of this work is based on his ideas and insights. As always, GCC developers are so helpful. We specially thanks all global maintainers, whose insights often save us weeks of work.

Our special thanks to Mr. Sandeep Khurana at HCL Technologies for his invaluable inputs.

## References

[GCCINT] GCC Internals Manual,
`http://gcc.gnu.org`

[SH4]
SH4 Programming Manual, Version 4.0
Renesas Technologies,
`http://www.renesas.com`.

[Eckstein] Erik Eckstein, Bernhard Scholz:
Addressing Mode Selection.

[Sanjiv] Sanjiv K. Gupta, Naveen Sharma:
Alias Analysis for Intermediate Code.
GCC Summit 2003

# Statically Typed Trees in GCC

*Nathan Sidwell*
CodeSourcery, LLC
nathan@codesourcery.com

*Zachary Weinberg*
CodeSourcery, LLC
zack@codesourcery.com

## Abstract

The current abstract syntax tree of GCC uses a dynamically typed über-union for nearly all nodes. The desire for a statically typed tree design has been raised several times over recent years, but there has been no concerted effort to implement such a design. We describe the impacts of the current design, both in implementation and performance degradation. We present a design for statically typed trees, along with case studies of part of the conversion. We outline a plan for full conversion and discuss further improvements that this would enable.

## 1 Current architecture

GCC uses a data structure called a **tree** for its high-level intermediate representation. The parser and semantic analyzer for a given programming language construct an initial tree representation of the program to be compiled. The high-level optimizers work directly on this tree. After they are done, the "expander" converts the optimized tree to a lower-level representation called **RTL** for further optimization and assembly output. We will not be discussing RTL in this paper, but it is worth mentioning that many of the same issues also apply.

A tree structure is a directed graph of **nodes**. Each node is a block of memory (a C `struct`) on the heap; the graph edges are pointers between these blocks. Tree nodes are dynamically typed. All variables and structure fields pointing to tree nodes have the type `tree`, which can address any node no matter what its internal structure is. To access the data carried in a node, one must use the macros defined in `tree.h`. These hide the exact representation and can be configured to carry out consistency checks at runtime (of GCC). We discuss the in-memory representation and the accessor macros in more detail below.

The **code** of a tree node determines its dynamic type. The generic (language independent) portion of the compiler defines approximately 150 codes. Front ends can define additional codes if necessary. There are ten **classes** (conceptual categories) of tree codes; each has a tag character to identify it. Front ends cannot define new classes. Presently, the classes are

- `'c'`, constants
- `'1'`, unary arithmetic operators
- `'2'`, binary arithmetic operators
- `'<'`, comparison operators
- `'r'`, references (e.g. array indexing)
- `'e'`, other expressions (e.g. `?:`)
- `'s'`, statements
- `'d'`, declarations
- `'t'`, types
- `'x'`, miscellaneous

Here are some example tree nodes, with the information they carry:

**`STRING_CST`** (class "constant")
A string constant. The node holds a

pointer to a separately-allocated byte array, and the length of this array.

**`PLUS_EXPR`** ("binary expression")
An addition operation. The node holds pointers to tree nodes representing the two addends.

**`IF_STMT`** ("statement")
An `if` statement. The node holds pointers to tree nodes representing the controlling expression, the "then" clause, and the "else" clause.

**`VAR_DECL`** ("declaration")
A declaration of a variable. The node is the root of a directed graph of nodes which collectively describe the properties of the variable.

**`INTEGER_TYPE`** ("type")
A description of an integer data type, either intrinsic to the programming language or defined on the fly by the program being compiled. Again, the node is the root of a directed graph describing the properties of the type.

**`TREE_LIST`** ("miscellaneous")
A linked list of other trees. Each node of the list can point to up to three different trees (known as the *type*, *purpose*, and *value*); however, usually only one of these slots is used.

**`ERROR_MARK`** ("miscellaneous")
A placeholder used when an error is encountered during compilation. This node carries no information. The compiler allocates only one `ERROR_MARK` node per invocation.

Trees exhibit three levels of polymorphism, which we will refer to as **substructure**, **multipurposing**, and **overloading**.

### 1.1 Substructure

The `tree` type is a pointer to a `union` of `struct`s. We will call these structs "substructures."

```
union tree_node
{
  struct tree_common common;
  struct tree_type type;
  struct tree_decl decl;
  struct tree_list list;
  ...
};
typedef union tree_node *tree;
```

All tree nodes include the fields of `struct tree_common`.[1] Most nodes also carry additional information stored in one of the other substructures. The tree code, which is a field of the common substructure, determines which substructure is active.

We can therefore categorize tree structures according to which substructure is valid. This categorization is similar, but not identical, to the categorization into classes. Front ends can also define new substructures, if necessary. Unfortunately the mechanism for this is somewhat awkward, since there is no way in C to augment the contents of a union.

Naturally, accessing the wrong substructure of a node can have grave consequences. To prevent this, GCC can be configured so that the accessor macros inspect the tree code and verify that they have been applied to the proper kind of tree. These checks are partially ad-hoc and partially machine-generated. The code is only known when the compiler is running, so the checks perforce must occur then. If one fails, GCC halts translation with the infamous "internal compiler error" (ICE) message.[2]

[1] because all the other substructures include `struct tree_common` as their first member.

[2] Jeff Law added the checking mechanism in 1998.

| Accessor | Used with | Content |
|---|---|---|
| TYPE_VALUES | ENUMERAL_TYPE | A list of CONST_DECLs, one for each enumeration constant. |
| TYPE_DOMAIN | SET_TYPE, ARRAY_TYPE | An integer type whose range determines the set of all valid indexes of this type. |
| TYPE_FIELDS | RECORD_TYPE, UNION_TYPE | A list of FIELD_DECLs, one for each data member of the type. |
| TYPE_ARG_TYPES | FUNCTION_TYPE, METHOD_TYPE | A list giving the type of each parameter, in order, to the function or method. |
| TYPE_DEBUG_ REPRESENTATION_ TYPE | VECTOR_TYPE | The type to use when describing this type to the debugger. (Most debuggers do not understand vectors.) |

Table 1: Multipurposing of the values field of tree_type

### 1.2 Multipurposing

Some fields of a substructure have different meanings for different tree codes. When there is more than one possible meaning, we say that that field is multipurposed. For instance, a tree_type structure represents a data type in the program being compiled. There are twenty tree codes that use this substructure. Eight of them assign one of five possible meanings to the values field. Table 1 enumerates the possibilities. The field goes unused in type nodes with one of the other twelve codes.

A relatively common special case of multipurposing is when a field has only one possible meaning, but only a subset of the tree codes for that substructure need to use that field. The others leave it as NULL.

### 1.3 Overloading

Many of the fields of a tree node are pointers to other nodes. These, like all pointers to tree nodes, have the type tree; as far as the C type system is concerned, they can point to any tree node. The operands of an PLUS_EXPR need not be expressions; they can be declarations, constants, types, or anything else.

Of course, not all possibilities can occur within a valid tree structure. The accessor macros partially validate the targets of pointer fields, and hand-coded assertions finish the job. When a field can legitimately point to more than one kind of node, we say that the field is overloaded.

The distinction between overloading and multipurposing is whether the code of the node containing the field determines what the field points to. The values field discussed above is multipurposed. An PLUS_EXPR's operand fields are overloaded—we do not know, upon encountering an PLUS_EXPR, whether its operands are expressions, declarations, or constants. (We *do* know that they are in one of those three categories.)

## 2 Issues of the status quo

The present architecture has a number of design issues, which manifest either as runtime overhead (both space and time) or as increased burden on the maintainers of the program. For an obvious example of both, the runtime checking done by the accessor macros slows the compiler down 5–15% (depending on in-

put). This is substantial enough that checking is disabled in release builds, which can mean that bugs go undetected. It is on by default in development builds, which means GCC developers all put up with a slower compiler for the sake of dynamic type safety. A slow compiler, hence a slow edit-compile-link-debug cycle, is a maintenance burden in itself; also, the checking mechanism is complicated and easy to misprogram (see section 2.2 for an example).

Each of the above varieties of polymorphism has its own set of issues, which we will discuss in turn. We will also discuss a number of related issues that we intend to address at the same time.

### 2.1 Substructure overhead

The dynamic type system has a certain level of intrinsic overhead. In many cases, GCC's own source code, not the content of the program being compiled, completely determines the code of a tree node. However, we must still maintain the node header, which is a full word (the code plus 24 flags). For smaller nodes, this can be a considerable amount of memory overhead.

In the larger substructures, many of the fields are only applicable to a few of the tree codes that use those substructures. This obviously wastes memory. It is a particularly severe problem for type and declaration nodes; the content of a `CONST_DECL` could fit into 16 bytes or so on a 32-bit host, but it occupies 116 bytes anyway. The other side of this problem is that adding a new field to a substructure consumes memory proportional to the total number of nodes using that substructure, not just the number of nodes it's relevant to. People therefore avoid adding fields to substructures. Instead they add new purposes to existing fields, which adds to maintenance burden instead. We could solve this within the existing framework by defining new substructures, at the cost of additional complexity in the accessor macros.

While many nodes have fields that are never used, some nodes do not have enough, which leads to ancillary data being maintained outside the tree structure. This may consume more memory than would have been required otherwise, and it also makes the program harder to maintain, since all the necessary information is not in one place. Ironically, the declaration structure is also an example of this, with substantial ancillary data being carried in the `cgraph_node` structures.

### 2.2 Multipurposing and generic accessors

In the past, the accessor macros and the debugging pretty-printer (`debug_tree`) did not know anything about multipurposing. One would use the same accessor macro (`TYPE_VALUES`) for all five purposes listed in Table 1. This led to confusion about which tree codes might use a given field. While considerable work has gone into introducing more specific accessors, some generic accessors still exist. Furthermore, the set of valid codes for each accessor may be incorrect. As we were writing this paper, we discovered that two of the accessor macros for the `values` field allowed a `VECTOR_TYPE`. Obviously the same field cannot serve two purposes simultaneously. Tightening up the checks exposed a harmless bug in `expr.c` and a more serious bug in `cp/decl.c`.

Accessors for fields with only one use are still likely to check only that the substructure is correct, not that the field is relevant to the specific code. They thus fail to document or enforce which codes the fields *are* meaningful for. Generic routines that inspect trees (such as the debug-info generators) won't bother to check for an appropriate code; they'll rely on the fields being `NULL` when they are irrelevant. This situation can persist unnoticed until some-

one decides to introduce a second purpose for one of these fields. In the process that person will tighten the checking macros, which will probably cause the generic routines to fail.

### 2.3 Abusive overloading

Tree overloading sometimes happens naturally. For instance, the tree the parser builds for a complex arithmetic expression will consist of `EXPR` nodes which may point to other `EXPR`s, to `DECL`s, or to constants. This is a straightforward way to represent an abstract syntax tree, and it rarely causes trouble.

However, since all pointers to trees have the generic type `tree`, overloading can potentially happen anywhere. Since this flexibility is available, it has been used whenever it was locally convenient, without thought for global consequences. Indeed, usually there are none—at the time. Once overloading has been added to a tree, every routine that examines it must be prepared for whatever it might find in the overloaded field. The only way to prove that a given tree field is not overloaded is to do a global data flow analysis, which can be very difficult. Thus, global consequences creep into the compiler over time, as new routines are added that inspect trees that might be overloaded.

An example of these creeping consequences is the `name` field of `struct tree_type`. This usually points to a `TYPE_DECL` node, but sometimes it points to an `IDENTIFIER_NODE` instead. When you get which, and what that means, is not documented anywhere. Routines that just want to know the printable name of a type have to use locutions like the following:

```
name = TYPE_NAME (t);

if (TREE_CODE (name)
    == TYPE_DECL)
  name = DECL_NAME (name);

if (TREE_CODE (name)
    != IDENTIFIER_NODE)
  abort ();
```

A less troublesome, but still unwise, case of overloading is the C and C++ parsers' reuse of expression nodes while parsing declarations. Normally a `CALL_EXPR` represents a call to a function; its operands are the function to call, and a list of actual arguments. But the C and C++ front ends also use this expression to represent a function declaration; then its operands are the function's name, and a list of formal parameter declarations. This is convenient for the parser, but necessitates a complicated conversion routine (`grokdeclarator`) to generate the type and declaration structures expected by the rest of the compiler. These peculiar expressions are intended never to escape the C front end, so they have not had creeping global consequences. However, from time to time one does escape and cause an ICE elsewhere in the compiler.

We can generate a crude estimate of the number of places that have to take care when inspecting overloaded trees by counting uses of the `TYPE_P` and `DECL_P` macros. As of March 15, there were 41 and 80 uses, respectively, of these macros in the main `gcc` directory, or about one use every 4000 lines. The C++ front end had more, 143 and 67 uses respectively, or about one use every 500 lines. This is due to heavy overloading in the trees used to represent templates; see section 5.3 for further discussion.

### 2.4 Lists of trees

Linked lists are very common within trees. This data structure is convenient when the size of the list is not known in advance. However,

```
struct tree_list {
  struct tree_common {
    tree chain;
    tree type;
    enum tree_code code :8;
    /* 24 flag bits */
  };
  tree purpose;
  tree value;
};
```

Substructure of `TREE_LIST`

linked lists have notably more overhead than vectors on several different grounds.

Singly linked lists can be constructed using reserved fields in the nodes carrying the data, or using separate "cons cells." Ignoring malloc overhead, a linked list using reserved fields in the data nodes consumes exactly the same amount of memory as a vector of pointers to those nodes. Either way, there is one extra pointer for each node. Linked lists built out of separate cons cells, on the other hand, use twice as much memory as a vector; two extra pointers per node. In exchange, a data node can be on more than one list if separate cons cells are used. Either way, traversing a linked list is more likely to cause memory-cache thrashing than traversing the vector.

All tree nodes have a `chain` field, reserved for chaining the node into a linked list. However, this field goes unused in approximately two-thirds of all nodes (not counting `TREE_LIST`; see more detailed analysis below, in Section 3.1). Instead, separate lists are built out of `TREE_LIST` nodes. This is the "cons cell" technique, but with far more overhead, because each node in the list has the ability to point to three data nodes instead of just one.

In practice, slightly more than half of all lists use only one data pointer per node, and almost all the rest use only two. Also, the node header (as always) consumes a full word; it is fair to consider that entirely wasted, since lists are always known from context and the flag bits go unused. (See section 3.2 for details.) For a list with only one data pointer per node, this structure is 60% wasted space; compared to a vector or an internally chained list, 80%.

Because all the pointers are generic, a `TREE_LIST` does not reveal any information about its contents. Code that processes lists must know from context what the list contains, or else be prepared to encounter anything. Context determines the content in most cases; again, this will be discussed in detail in section 3.2.

### 2.5 Language-specific trees

As we mentioned above, language front ends have the ability to define new tree codes. Often these codes do not need their own substructures. For instance, all of the language-specific codes defined by the C front end are for C-specific operators, which use the generic "expression" substructure. However, some languages need their own substructures. The C++ front end defines five such. Since the definition of the basic `tree` type is in a language-independent header file, there is no way to include these substructures in the tree union. Thus, the accessor macros for those substructures must include casts to the appropriate type, which is a minor hassle. Also, the garbage collector must assume that language-specific substructures can be encountered anywhere, which adds both runtime overhead (determining which substructure is active costs two function calls per node visited) and source complexity (special annotations to indicate that the tree union is not exhaustive).

The `type` and `decl` substructures include an opaque pointer field that front ends can use to attach their own special data to type and declaration nodes. This mechanism provides a

clear separation between generic and language-specific data. It requires no casting, since the opaque pointer refers to a forward-declared `struct` type. Front ends simply provide a complete declaration. However, it does require a second memory allocation, which adds overhead.

Also, the front end might need to multipurpose this field—storing different information depending on what sort of type or declaration it is—but this is inconvenient, since these structures are *not* trees and cannot use the machinery that exists for tree polymorphism. The C++ and Java front ends solve this problem by duplicating much of that machinery. The Ada front end, instead, pretends that the field points to a tree, which can then be multipurposed in the normal fashion. Neither is an ideal solution.

The substructure for a bare identifier (code `IDENTIFIER_NODE`) also provides for front ends to attach their own data. Because identifiers are so frequent, this data is appended to the generic substructure instead of being separately allocated. This is efficient, but requires front ends to define complex macros to access their own data, just as they would for entirely language-specific substructures. Also, `IDENTIFIER_NODE`s are used in contexts where the language-specific data will never be used (notably `DECL_ASSEMBLER_NAME`) but space is allocated for it anyway.

The `tree_common` structure carries seven flag bits specifically for use by front ends, and several more that have generic names but are only relevant to front ends. The `type` substructure carries another seven, the `decl` substructure eight. These are not overhead as they occupy space that would otherwise be padding. However, they are a maintenance burden, because they are heavily multipurposed. It is often unclear which front ends use which bits for what, and `debug_tree` prints them with generic names.

Languages sometimes invent their own multipurposings for fields that would otherwise go unused. The C front end has recycled the `TYPE_VFIELD` field of incomplete `RECORD_TYPE` nodes to carry a list of `VAR_DECL`s with the incomplete type, so that it can adjust them later if the type is completed. This is much more efficient than the previous approach of carrying around a list of all variables with incomplete types in the translation unit. However, it directly violates the language-independent compiler's assumptions about what can appear in `TYPE_VFIELD`. Several bugs have been traced to this list escaping the C front end.

`TYPE_VFIELD` is available for use in the C front end because `RECORD_TYPE`s in C never have vtables. The `RECORD_TYPE` code is used for object classes as well as "plain old data" structs, so it has all the fields necessary to handle both, even though classes never occur in C. More generally, language-independent trees carry fields needed to represent the constructs of *all* the languages that GCC supports, even if they are being used to represent a language that doesn't have those constructs. This is memory overhead, no more... unless, as with `TYPE_VFIELD`, someone gets clever.

### 2.6 Memory allocation, precompiled headers

GCC uses a garbage-collecting allocator for all trees. This is convenient, because no one ever has to worry about the lifetime of these data structures.[3] It also facilitates precompiled headers (PCH). The current implementation, to first order, simply serializes to disk all live data in garbage-collected memory.

[3] Before the garbage collector was introduced, in 1999, use-after-free bugs appeared about once every two weeks; now they are unheard of.

When the garbage collector was first introduced, the marking routine for each data structure had to be written by hand. Now instead we use special "`GTY`" annotations in the source code, and a program called `gengtype` which understands a subset of C's type grammar. It scans the source code and generates marking routines, directed by the annotations. It also generates slightly different walking routines which are used for PCH save and restore.

Both these things are great achievements from a software maintenance standpoint. In the normal course of affairs, programmers need never worry about memory lifetime. PCH requires slightly more attention as one must ensure that everything that needs serialization is properly annotated. The `gengtype` program is a powerful tool for doing introspection on GCC's data structures. We used it for this paper, to gather statistics on how fields of tree nodes are used. We discuss below some other ways it could be helpful.

On the other hand, the garbage collector is not at all efficient. It allocates memory out of fixed-size buckets, with pages reserved for allocations of a given size, which causes considerable memory fragmentation. The collector uses a naïve mark-and-sweep algorithm, which has to scan the entire active memory set on each collection. This is so slow that GCC contains throttling heuristics that effectively disable all memory reuse for average-size translation units. The auto-generated marking routines require that type tags be in the same block of memory as the unions they disambiguate; in some places (notably the C++ front end's `struct lang_decl`) this forces the creation of a redundant tag.

This paper does not directly address any of the problems with the garbage collector. However, we expect our changes will cause trees to use substantially less memory and have somewhat more predictable lifetimes. In conjunction with the "zone collector" project, which is working towards a generational collection algorithm, this should offer substantial performance improvements.

## 3 Measurements

In order to make sensible plans to solve the problems we have discussed, we need hard data on how severe they are. Code inspection can reveal potential problems, but does not tell us what the actual allocation patterns are, and there is no way to get a sense of the "big picture." Overloading in particular is very hard to discover by code inspection.

We therefore modified the `gengtype` program to generate instrumentation which would measure how much overloading appeared in the trees produced by compilation of a test program. We classified each node twice, once by its tree code and once by its substructure.

For each field that pointed to another tree node, we recorded what kinds of tree it could point to, including nothing. When substructures contained arrays, such as `struct tree_exp`, we considered each element a separate field. This reveals for instance that the first operand of a `CALL_EXPR` is usually an `ADDR_EXPR`, and the second is always a `TREE_LIST`. We instrumented lists specially, recording their average length and the value distribution of the entire list, instead of treating each node as a separate entity.

Using CVS HEAD as of 15 March 2004, we measured allocations for the compilation of GCC's own C and C++ front ends (this exercises only the C compiler) and for a small STL-based C++ program. Each of these was compiled in a single pass, using GCC's intermodule mode. All inlining was disabled, and all function bodies retained, so that each function body

would be counted exactly once. Measurements were taken once at the end of compilation, so transitory tree nodes were not inspected. Unfortunately this means we missed some of the more bizarre things done with trees, such as the declaration expressions discussed in Section 2.3.

The C compiler generated a similar distribution of tree nodes during compilation of both front ends, so we present here only data for the C++ front end. Compiling this C program generated about 1 million instrumented nodes, occupying 75MB of storage. The C++ program was smaller. Compiling it generated about 150,000 nodes, occupying 9MB.[4]

### 3.1 Fields of `tree_common`

The `tree_common` substructure contains two tree-pointer fields, `chain` and `type`, which are present in every node whether it needs them or not. The utilization of these fields is laid out in Tables 2 and 3. (The "proportion" column is proportion of total GC memory allocation; not all of this is trees.) It is immediately clear that memory could be saved just by excluding these fields from substructures that never use them.

For our C++ test case, removing the `chain` pointer from nodes where it isn't used saves 134KB, or 1.5% of the total memory allocation. Removing the `type` pointer saves 58KB, or 0.6% of total memory. The numbers are more impressive for C: removing `chain` saves 2.3MB, or 3.1% of memory; removing `type` saves 780KB, or 1.0% of memory. If internal memory fragmentation is reduced by this change, which is likely as many of the affected nodes are one word bigger than a power of two, memory savings could be even bigger.

With more code changes, all of the uses of the `chain` field could be eliminated, saving even more memory. `DECL`s and `BLOCK`s are chained together to indicate the lexical scope of declarations and these lists could easily be replaced with vectors. Furthermore, in the GIMPLE representation (which had not yet been merged when these measurements were taken) statements are held in sequence with an external doubly-linked list, so they do not need internal chaining either.

### 3.2 List distribution

`TREE_LIST` nodes are used for all external singly-linked lists. If we looked at these nodes in isolation, all their fields would appear to be heavily overloaded. However, our instrumentation captured the context of each list, revealing that most lists have predictable dynamic types.

The C front end allocated roughly 300,000 list nodes while compiling the C++ front end. There were seven major contexts, which are enumerated in Table 4. Of these, only two have nontrivial amounts of overloading, and one of those is because `CONSTRUCTOR` nodes are used to initialize both arrays and structures. It is also apparent that the `type` field of these lists is completely unused, and the `purpose` field is unused in half of the cases. We could save roughly 5MB (7% of the total allocation) by converting them all to specialized vectors.

The C++ front end uses a wider variety of lists. Our C++ test case produced 70,000 tree nodes in about 30 different uses, which are enumerated in Table 5. Like the C front end, the `type` field is unused in nearly all contexts, and the `purpose` field is unused in about half of the cases. There is quite a bit of overloading, but in most cases there is one primary usage and a few outliers. The structures used to represent templates, however, will require special attention and is discussed in Section 5.3. If all of these uses were converted to specialized vec-

[4] All statistics are for a host architecture with 32-bit pointers.

tors, we might be able to save about $2/3$MB of memory (8% of the total).

We did not instrument `TREE_VEC` as carefully as `TREE_LIST`, but it shows similar properties. It does not carry three data pointers per entry, but it does have the full overhead of a `tree_common` header, whose `chain` and `type` fields go unused. The entries are, as usual, declared as `tree`s rather than anything more specific, but in most cases the entries are homogeneous within a given class.

### 3.3 Overloaded fields

Tables 6 and 7 show the distribution of overloaded and/or multipurposed fields for the C and C++ test programs respectively. Multipurposed fields are in *italics*. We only show cross-class overloading, as we are not proposing to get rid of within-class overloading. Most overloading occurs among one primary class and a few outliers. Where there are "secondary" uses, appearing in more than 5% of measured nodes, that is usually a case of multipurposing.

The primary class is not always what one expects—in C, both BLOCK.supercontext and EXPR.operands are 99% `DECL`s, where one might expect to find more `BLOCK`s and `EXPR`s respectively. This reflects the form of the typical C program. Inner scopes tend not to have variable declarations, and therefore not to need `BLOCK` nodes. Expressions tend to be simple, hence most `EXPR` nodes point directly to variable `DECL`s rather than to subexpressions. The C++ front end does more overloading than C, but we still observe the same pattern of primary uses and outliers, except where there is multipurposing. Expressions appear to be more complicated in C++ than in C, but still 94% of `EXPR`s point directly to `DECL`s.

TYPE.context and DECL.context are anomalous in having substantial secondary targets without multipurposing being involved. These fields point "upward" in the abstract syntax tree, toward larger lexical structures. Since `TYPE`s and `DECL`s can nest inside each other (especially in C++), the context fields need to be able to point to both `TYPE`s and `DECL`s.

## 4 Redesign

Our primary goal in redesigning trees is to reduce runtime overhead and maintenance burdens. As we have discussed, overhead comes first from wasted memory. The primary causes of wasted memory are unused fields in various tree substructures, and overuse of linked lists.

We could address unused fields without introducing any new static types. We could simply promote all instances of multipurposed fields to substructures. Constants are already like this. Each code in the "constant" class (integer, real, complex, string, vector) has its own substructure. Structure initializers are exceptional in that they are not treated as constants, but as expressions—this should probably be changed. It would not be hard to extend this to other structures. We would also want to break up `tree_common`, moving its pointers into the substructures where they are actually used.

Furthermore, we already have a `TREE_VEC` node that could replace `TREE_LIST` whenever the list length is known in advance and only one pointer per element is needed. For instance, it would be feasible to do this for `BLOCK_VARS`. Where this will not work, we could invent new lists with only one or two data pointers per node.

These changes would reduce maintenance burdens only because accessor macros would have more specific names, and the documentation would be improved. They would do nothing at all for the overhead entailed by runtime type checking. In fact, they might make it worse,

since many checking macros would become more specific. For instance, TREE_CHAIN and TREE_TYPE currently do no checking at all; in the above regime they would be replaced by several new macros, which would check for specific substructures.

In order to go any further, we need to make the static types of trees more specific. That is, we need to stop using tree as the type declaration of every pointer to a tree. If we are to do this, we must decide how specific to be in our static declarations. Where possible we will use pointers to specific structures. However, some degree of overloading is necessary. We propose to introduce four new types, each of which covers a subset of the present tree classes. A pointer with one of these types can be overloaded freely within that subset, but not outside. We discuss techniques for removing cross-class overloading in section 4.3. The replacement types are:

**TYPE** Type nodes: the present 't' class. For instance, INTEGER_TYPE, POINTER_TYPE, and RECORD_TYPE.

**DECL** Declaration nodes: the present 'd' class. For instance, FUNCTION_DECL, VAR_DECL, and TYPE_DECL.

**EXPR** Expression nodes: the present '1', '2', 'r', '<', and 'e' classes. For instance, PLUS_EXPR, LE_EXPR, and ADDR_REF.

**CONST** Constant nodes: the present 'c' class. For instance, INTEGER_CST and STRING_CST.

The 's' class is not included in this mapping because, with the introduction of GENERIC and GIMPLE, the language-independent compiler no longer makes a strong distinction between statements and expressions. For instance, COND_EXPR can be either a ?: operator or an if statement. This does not preclude a front end from making a strong distinction in its own data structures, if that is appropriate to the language it recognizes.

Each of the miscellaneous trees (class 'x') requires individual attention. Some of them can be replaced with plain C structs that never participate in overloading. The BLOCK node for instance will get this treatment. Other nodes will be be recategorized into one or more of the above classes. For instance, we need equivalents of ERROR_MARK for each of the above categories; these should *not* be unique, so that they can carry information (such as the location of the error).

Obviously it will not be possible to continue using one structure, carrying no static type information, for all linked lists. However, as we detail in Section 3.2, most lists point to data items whose dynamic types are both predictable and homogeneous. Therefore, with a moderate amount of effort we can replace TREE_LIST with specialized list nodes for each of the classes.

### 4.1 Type safety

Under the old design, all pointers had the same static type, so there was never any need to convert them. Under the new design, we would like to make the static types of pointers as specific as possible. The four classes above are base types in the C++ sense, and each substructure is a derived type. We will need a type-safe and terse way to convert between base and derived type pointers. Unfortunately the C language does not provide convenient facilities for this sort of operation. Pointers to different structs are not assignment-compatible. There is only one cast operator, (*type*), which does not validate the incoming type at all.

We can simulate the C++ derived-type compatibility rule and `dynamic_cast<>` operator in C, with a small amount of extra verbosity and some GNU extensions. In Figure 1 we illustrate one way to implement the conversion operations, and the associated structure layout. Code written to this convention should look almost the same as code written to the old convention, but with specific variable types and occasional explicit conversions. It might be possible to use `gengtype` to generate all of the accessor macros and checking logic from the substructure definitions, thus eliminating that source of bugs and tedium.

There would be a `_common` structure for each of the four major static types. Any fields that truly are common to all substructures of that type can be placed there. In the example, we included two boolean fields which are documented as relevant to all constants. We have not yet decided what naming convention use for the new types; the mixture of struct tags and all-caps typedefs in figure 1 is only one possibility.

The GNU extensions are only necessary for type checking. When GCC is built with a compiler that does not support them, the macros can expand to unchecked casts; the compiler will still work. The compile-time error message produced by these macros is suboptimal; it could be improved with a `__builtin_error` primitive. Also, in real life the runtime checks would call a more specific ICE-reporting routine than `abort`. These details were omitted from the example for brevity.

Some checking does still occur at runtime. We expect that the overhead will be substantially lower in this scheme, but we can still disable runtime checking in release builds for efficiency.

### 4.2 Language augmentations

The coding convention shown in Figure 1 deliberately does not use unions, unlike the current convention. This is because the union cannot include any language-specific substructures, and we want to put them on an equal footing with language-independent substructures. The checked-cast approach is similar to what is done now for language-specific substructures, but safer. If the macros are automatically generated, it will also be much less tedious. Front ends are also free to declare new polymorphic classes; for instance, a language that wants a strong distinction between statements and expressions can invent a **STMT** class.

We also want to make it easier to add language-specific data to generic substructures. It is straightforward for a language to declare an augmented substructure and accessors, as they do now for `IDENTIFIER_NODE`. However, the garbage collector must be advised to allocate more memory for the augmented structure, and to walk the complete structure for pointers when marking live data. This is done for `IDENTIFIER_NODE` with special `GTY` markers and language hooks, which do not scale. We have not yet decided on a tactic for this problem.

Finally, we intend to make tree codes more specific so that languages do not have to incur overhead for functionality they do not use. For instance, the `RECORD_TYPE` code will apply only to "plain old data;" we will introduce a new `CLASS_TYPE` node for object classes.

### 4.3 Adaptor nodes

Section 3.3 outlined instances of cross-class overloading, that is, cases where `tree` pointers can refer to more than one of the four static classes discussed in Section 4. We can elimi-

nate many of these, but some are legitimate.

We do not want to combine the `DECL`, `EXPR`, and `CONST` classes, but we could introduce **adaptor** nodes, which fit into one class and carry a pointer to another class. They might or might not carry other information. We already have the notion of a `TYPE_DECL`; we could reuse it as an adaptor for context fields pointing to a `TYPE`. Context fields can also point to `BLOCK`s; for that, we would need a new `BLOCK_DECL` adaptor.

The statistics in tables 6 and 7 show that 94–99% of expression operands are `DECL`s, so it would be most efficient to make that the unmarked case. We would add an `EXPR_DECL` adaptor for subexpressions, and use the existing `CONST_DECL` as an adaptor for literal constants. This could facilitate conversion to GIMPLE form, where all subexpressions are separated from their contexts.

## 5 Conversion plan

Converting to statically typed trees is a considerable amount of work. It will have to be done either piecemeal on the mainline, or on its own dedicated branch. If the work is done on a branch, it will rapidly become very hard to merge in changes from the mainline. However, if the work is done on the mainline, it is likely to be disruptive to other projects. The conversion may not be monotonic, and there are several issues as yet unresolved, for which experimentation will be necessary. Also, this project is more work than one person can do alone. Collaboration by emailing patches back and forth is tedious, compared to collaboration by working on the same branch.

On balance, we believe that most of the work should be done on a branch. However, in order to avoid severe divergence, the project should be broken into steps which can be merged back to mainline when complete. We will partition these steps into three stages.

The first stage of the process is to promote all multipurposed fields to substructures. It may be feasible to do this stage before branching. It is very simple and low-impact for fields whose accessor macros are already as specific as they can get. Fields that have non-specific accessor macros require more thought, and the change may be quite large, but still mostly mechanical. The `chain` and `type` fields of `tree_common` will migrate into the substructures that actually use them. It would be nice to do the same for the common flag bits, but that may not be feasible without introducing unwanted padding.

The tree-ssa branch has introduced a number of new `'x'` nodes that are used in expressions, such as `SSA_NAME`. These are not in class `'e'` mainly to avoid wasting memory on useless fields attached to all expressions. If the substructure conversion is done properly it will be possible to put them in class `'e'` or possibly a new expression subclass.

The second stage is to eliminate as much overloading as possible, particularly what we might describe as "abusive" overloading. We discuss approaches to some of these in sections 5.1–5.3. The branch will be merged after each abuse has been rectified. This stage will have to occur semi-concurrently with the next one, because we do not know where all of the problems are.

The third stage is to peel off the major tree classes from the über-union, one at a time. The branch will be merged after each step. Except where we encounter unexpected abuses, the substantial changes in this stage affect only the implementation of the accessor macros. However, this is the stage where we change variable declarations, introduce explicit conversions, and rename accessor macros to con-

form to a naming scheme that facilitates automatic generation. This will entail mechanical changes all over the compiler. We propose to do this stage in the following order:

**Identifiers** With the exception of C++ template bodies, there are only a few places where a tree node might or might not be an identifier, and they are all arguably bugs. The new C++ parser should make it feasible to use custom data structures for C++ template bodies, so that `IDENTIFIER_NODE` need not be an overloading candidate at all. In some places, identifiers are used where unboxed strings would suffice; we will remove all such identifiers in this step.

**`ERROR_MARK`** There is one error mark node, which can appear in any context where the tree is incomplete because the input program was incorrect. It carries no information. We mean to replace it with separate `INVALID_TYPE`, `INVALID_DECL`, `INVALID_EXPR`, and possibly `INVALID_CST` codes. These nodes will not be unique, and will carry enough information that later stages of compilation do not need to be aware of them.

**Lists and vectors** `TREE_LIST` must be replaced with specialized list nodes that carry static type information. It is also desirable to use vectors where possible, instead of lists. In this step we will design a macro API for synthesizing vector and list types, and the associated runtime API for building lists, converting lists to vectors, etc. This will allow us to save memory immediately, by removing the unused pointers from most lists. In further steps we will use it to define specialized list and vector types as needed.

**Blocks** The lexical binding node, `BLOCK`, can only appear within certain nodes and contexts, and therefore can be separated out relatively easily. It contains a list of `DECL`s, which will be the first use of specialized vector types.

**Types** Of the remaining tree nodes, types are the most distinct; there is rarely cross-class overloading between types and other things. However, we will need to create specialized lists of types, and we expect to find abuses in their relationship to declarations.

**Constants** In this step we will replace overloading between declarations and strings with anonymous `CONST_DECL` adaptors. Also, trees which are always `INTEGER_` or `STRING_CST` nodes will be replaced with unboxed integers or strings.

**Expressions** Next, we give expressions a distinct type, and make their operands always be `DECL` nodes. Subexpressions will be wrapped in `EXPR_DECL` adaptor nodes. This is one of the most invasive changes to be made; however, a suitably clever definition of `TREE_OPERAND` should make it possible to do it piecemeal.

**Declarations** At this point the only things left in the tree union are declarations. We can replace all remaining `tree` variables with `DECL` variables, and delete the union entirely.

We will now discuss a few conversion steps in more detail.

### 5.1 C declaration parsing

The C and C++ parsers reuse expression nodes for temporary structures while parsing declarations, as described in section 2.3. This is incompatible with static typing. Also, it is inefficient; the temporary structure is far larger

```
struct binfo {
  unsigned int flags;
  tree type;
  struct binfo *next;
  struct binfo *inheritance;
  tree offset;
  tree vtable;
  tree virtuals;
  tree vptr_field;
  unsigned int num_bases;
  struct base {
    tree access;
    struct binfo *base;
  } bases[];
};
```

Custom BINFO structure

than it needs to be (for instance, lists of identifiers are used in places where flag words would suffice) and the entire thing is discarded after processing by grokdeclarator, producing lots of garbage.

We plan to replace these expressions with a custom data structure. It need only contain fields for the information added at its level (cv-qualifiers, attributes, array or function parameters), an enumeration of what is being declared (array, pointer, etc), and a pointer to the structure for the next level. It would use the polymorphism techniques described in Section 4.1, but static type constraints would ensure that it never escaped the front end.

We expect this project to have the pleasant side effect of replacing grokdeclarator with a set of simpler functions, none of which is 1200 lines long.

### 5.2 BINFOs

The RECORD_TYPE for each class declared in a C++ program has a set of BINFO structures to represent its base class organization. There is one BINFO for each base class, arranged in a directed acyclic graph which mirrors the class hierarchy. They carry data such as the location of the base sub-object, the class type of the base, etc.

A BINFO is a TREE_VEC with indexes defined for each piece of information. Information about a BINFO's base BINFOs is held in two additional TREE_VECs, which is unnecessary fragmentation. There is a comment in tree.h suggesting that this be changed:

> ??? This could probably be done by just allocating the base types at the end of this TREE_VEC (instead of using another TREE_VEC). This would simplify the calculation of how many basetypes a given type had.

As with declarator expressions, we mean to replace BINFO with a custom structure. The fields that point to BINFOs are never overloaded, so we do not need to make it a tree substructure. An example structure is shown above, as it would appear before conversion to specific static types. Further memory savings are possible: we can store less information in the BINFO and more in the RECORD_TYPE of the base class, where it is not copied for every derived class. The virtuals field is a long list, with one entry for every virtual function in that class. If it can't be moved to the RECORD_TYPE, we can at least convert it to a specialized vector.

### 5.3 Template arguments and levels

C++ template parameters may be types, expressions, or nested templates. Presently, the C++ front end takes advantage of overloading to put all these things in a single parameter vector. Many of the uses of TYPE_P and DECL_P within the C++ front end are due to this overloading. In this context, types are the most

```
struct inner_vec {
  unsigned int num_args;
  tree args[];
};
struct outer_vec {
  unsigned int num_levels;
  struct inner_vec *levels[];
};
```

Two-dimensional template parameter array

common sort of parameter. We could use C++-specific `EXPR_TYPE` and `DECL_TYPE` adaptor nodes. Another option is to use a tagged array of unions, but then we would have to find somewhere to put the tags.

It is possible for a template to have more than one level of template parameters. Such templates have a vector of parameter vectors, one for each level. To avoid overhead, templates with only one level of parameters omit the outer vector. This is another kind of overloading, and it costs quite a bit of complexity (mostly in `cp-tree.h`'s macros for manipulating template trees). A specialized two-dimensional array would have substantially less overhead. One possible structure layout is shown here.

## 6 Closing remarks

This paper concentrates mostly on the common code, and the C and C++ front ends. The Java, Ada, Fortran and Objective C front ends will no doubt have specific issues during conversion. With the possible exception of Ada, we expect that these will be no more trouble than the C++ front end. We will need support from front end maintainers to complete the conversion for all front ends.

We have glossed over the process of defining specialized list and vector types. By the time that is necessary, we will have already converted some list usages, giving experience in the features that are necessary. We expect that at that time a good approach will be obvious.

## 7 Acknowledgments

We would like to thank Diego Novillo and Christian Lavoie for commenting on drafts of the paper, and Sumana Harihareswara, Michael Ellsworth, and Julia Bernd for copyeditorial assistance above and beyond the call of duty.

## Tables and figures

In Tables 4–7, upper case indicates nodes with a particular tree structure; lower case indicates nodes with a particular tree code. An entry with just a dash (—) indicates a field that was never used.

| | | Utilization | |
|---|---|---|---|
| Class | Proportion | chain | type |
| BLOCK | 1.61% | 47.78% | 0.00% |
| DECL | 26.46% | 89.81% | 99.30% |
| EXPR | 35.72% | 0.00% | 100.00% |
| STMT | 14.85% | 60.21% | 0.00% |
| IDENTIFIER | 1.72% | 0.00% | 0.00% |
| CONSTANT | 14.75% | 0.00% | 100.00% |
| TYPE | 4.89% | 0.00% | 71.42% |

Table 2: `tree_common` utilization by class in C program

| | | Utilization | |
|---|---|---|---|
| Class | Proportion | chain | type |
| BLOCK | 3.85% | 2.35% | 0.00% |
| DECL | 33.60% | 60.80% | 99.68% |
| EXPR | 19.23% | 0.00% | 43.45% |
| STMT | 14.46% | 38.93% | 0.00% |
| IDENTIFIER | 7.26% | 0.00% | 7.40% |
| CONSTANT | 3.18% | 0.00% | 100.00% |
| TYPE | 12.80% | 0.00% | 65.98% |

Table 3: `tree_common` utilization by class in C++ program

| Field | Null | Len | Type | Purpose | Value |
|---|---|---|---|---|---|
| call_expr.op[1] | 2% | 3.5 | — | — | EXPR |
| record_type.minval[a] | 99% | 3.0 | — | — | DECL |
| function_type.values | 0% | 3.7 | — | — | TYPE |
| enumeral_type.values | 0% | 23.1 | — | identifier | integer_cst |
| DECL.attributes | 91% | 1.4 | — | identifier | —[b] |
| TYPE.attributes | 98% | 1.9 | — | identifier | list |
| constructor.op[0] | 0% | 9.6 | — | field_decl 65%<br>integer_cst 35% | EXPR |
| TYPE.attributes.value | 0% | 2.1 | — | — | identifier 26%<br>integer_cst 74% |

[a] `C_TYPE_INCOMPLETE_VARS`; the C front end has invented its own multipurposing for this field (see section 2.5).

[b] This field is non-NULL for some attributes, none of which are used in the program we measured.

Table 4: Lists in C program

| Field | Null | Len | Type | Purpose | Value |
|---|---|---|---|---|---|
| record_t.pure_virtuals | 99% | 8.7 | — | — | method_t |
| record_t.befriending_classes | 96% | 1.3 | — | — | record_t |
| record_t.vfields | 85% | 1.0 | — | — | record_t |
| record_t.friend_classes | 97% | 2.3 | — | — | record_t |
| type_d.initial.value | 0% | 2.0 | — | — | DECL |
| var_d.initial | 17% | 2.3 | — | — | EXPR |
| nw_expr.operands[0] | 77% | 1.0 | — | — | EXPR |
| call_expr.operands[1] | 32% | 2.4 | — | — | EXPR >99% |
| | | | | | identifier <1% |
| TYPE.attributes.value | 0% | 1.6 | — | — | integer 82% |
| | | | | | identifier 18% |
| function_t.binfo | 73% | 1.0 | — | — | null 99% |
| | | | | | record_t 1% |
| method_t.binfo | 82% | 1.0 | — | — | null >99% |
| | | | | | record_t <1% |
| cast_expr.operands[0] | 32% | 1.1 | — | — | DECL 55% |
| | | | | | EXPR 38% |
| | | | | | CONST 7% |
| namespace_d.initial | 57% | 1.0 | — | namespace | — |
| namespace_d.saved_tree | 71% | 1.0 | — | namespace | — |
| DECL.attributes | 96% | 1.4 | — | identifier | — |
| TYPE.attributes | 99% | 1.7 | — | identifier | list |
| type_d.initial | 99% | 1.3 | — | identifier | list |
| enumeral_t.values | 0% | 16.9 | — | identifier | integer |
| record_t.vcall_indices | 85% | 5.6 | — | function_d | integer |
| constructor.operands[0] | 0% | 8.6 | — | integer | EXPR |
| record_t.template_info | 24% | 1.0 | — | DECL | vec |
| record_t.vbases | 98% | 1.0 | — | record_t | vec |
| template_d.arguments | 0% | 1.0 | — | int_cst | vec |
| DECL.template_info | 63% | 1.0 | — | DECL >99% | vec |
| | | | | overload <1% | |
| ctor_initializer.operands[0] | 10% | 2.1 | — | DECL 95% | list |
| | | | | record_t 5% | |
| record_t.decl_list | 50% | 19.4 | — | record_t 99% | DECL |
| | | | | null 1% | |
| function_t.values | <1% | 3.3 | — | null >99% | TYPE |
| | | | | EXPR <1% | |
| method_t.values | 0% | 3.3 | — | null 97% | TYPE |
| | | | | EXPR 3% | |
| TEMPLATE_PARMS | 0% | 1.0 | — | null 74% | DECL |
| | | | | TYPE 25% | |
| | | | | EXPR 1% | |
| template_d.vindex | 96% | 3.4 | — | vec | record_t 97% |
| | | | | | null 3% |
| template_d.size | 56% | 2.0 | null 99% | vec | DECL 99% |
| | | | record_t 1% | | vec 1% |
| namespace_d.vindex | 57% | 1.0 | — | null 67% | null 67% |
| | | | | namespace 33% | namespace 33% |

Note: **_t** is short for **_type**, **_d** for **_decl**.

Table 5: Lists in C++ program

In Tables 6 and 7, *italics* indicate a multipurposed field; roman font indicates an overloaded field.

| Field | Primary | | Secondary | | Outlier | |
|---|---|---|---|---|---|---|
| BLOCK.supercontext | DECL | 99% | | | BLOCK | 1% |
| DECL.context | DECL | 100% | | | TYPE | <1% |
| *DECL.initial* | DECL | 79% | EXPR | 19% | TYPE | 2% |
| | | | | | BLOCK | <1% |
| DECL.result | TYPE | 86% | DECL | 14% | | |
| EXPR.operands | DECL | 99% | | | EXPR | <1% |
| | | | | | IDENTIFIER | <1% |
| | | | | | LIST | <1% |
| | | | | | BLOCK | <1% |
| TYPE.context | DECL | 87% | BLOCK | 13% | | |
| TYPE.name | DECL | 100% | | | IDENTIFIER | <1% |
| *TYPE.values* | LIST | 76% | DECL | 24% | TYPE | <1% |

Table 6: Multipurposing and overloading in C program

| Field | Primary | | Secondary | | Outlier | |
|---|---|---|---|---|---|---|
| BLOCK.supercontext | DECL | 98% | | | BLOCK | 2% |
| *DECL.arguments* | DECL | 79% | LIST | 14% | | |
| | | | INT_CST | 7% | | |
| DECL.context | DECL | 98% | | | TYPE | 2% |
| *DECL.initial* | TYPE | 54% | DECL | 16% | LIST | 1% |
| | | | BLOCK | 12% | STRING | <1% |
| | | | INT_CST | 11% | | |
| | | | EXPR | 5% | | |
| DECL.befriending_classes | LIST | 60% | | | | |
| | DECL | 40% | | | | |
| DECL.result | DECL | 98% | | | TYPE | 2% |
| DECL.saved_tree | EXPR | 100% | | | LIST | <1% |
| DECL.size | INT_CST | 88% | LIST | 12% | | |
| *DECL.vindex* | DECL | 54% | INT_CST | 22% | TYPE | 4% |
| | | | LIST | 19% | | |
| EXPR.operands | DECL | 94% | EXPR | 5% | LIST | <1% |
| | | | | | INT_CST | <1% |
| | | | | | BLOCK | <1% |
| | | | | | STRING | <1% |
| | | | | | TYPE | <1% |
| TYPE.context | DECL | 62% | TYPE | 38% | | |
| *TYPE.values* | LIST | 67% | DECL | 22% | IDENTIFIER | 1% |
| | | | TPI | 9% | EXPR | <1% |
| | | | | | TYPE | <1% |

Table 7: Multipurposing and overloading in C++ program

```
/* If V has type T, return V, else issue an error.  */
#define verify_type(T,V)                            \
  (__builtin_choose_expr                            \
   (__builtin_types_compatible_p (typeof(V), T),    \
    (V), (void) 0))

/* If V has type T or F, return (T)V, else issue an error.  */
#define validated_cast(T,F,V)                       \
 (__builtin_choose_expr                             \
  (__builtin_types_compatible_p (typeof(V), T)      \
   || __builtin_types_compatible_p (typeof(V), F), \
  (T) (V), (void) 0))

/* If V has static type F or T and dynamic type K, return (T)V, else
   issue an error.  F and T are checked at compile time, K at runtime.  */
#define with_dynamic_type(K,T,F,V)                  \
  ({ T _v = validated_cast(T,F,V);                  \
     if (_v->common.kind != K)                      \
       abort ();                                    \
     _v; })

enum cst_kind { INTEGER_CST, ... };

struct cst_common
{
  enum cst_kind kind : 8;
  bool warned_overflow : 1;
  bool overflow : 1;
  /* possibly other flag bits */
};
typedef struct cst_common *CONST;
#define CONST(C) verify_type(CONST, &C->common)

#define CONST_OVERFLOW(C) CONST(C)->overflow
#define CONST_WARNED_OVERFLOW(C) CONST(C)->warned_overflow

struct cst_int
{
  struct cst_common common;
  unsigned HOST_WIDE_INT low;
  HOST_WIDE_INT high;
};
#define CST_INT(C) \
 with_dynamic_type(INTEGER_CST, struct cst_int *, CONST, C)

#define CST_INT_LOW(C) CST_INT(C)->low
#define CST_INT_HIGH(C) CST_INT(C)->high
```

Figure 1: Structure and macro conventions for type safety

# Gcj: the new ABI and its implications

*Tom Tromey*
Red Hat, Inc.
tromey@redhat.com

*Andrew Haley*
Red Hat, Inc.
aph@redhat.com

**What is binary compatibility?**

The Java Language Specification [2] has an entire chapter, Chapter 13, dedicated to binary compatibility. This chapter lays out rules for writing binary compatible programs: programs can be changed in these ways without requiring the recompilation of dependent modules. This covers some simple, obvious things, such as the fact that adding or removing the `synchronized` keyword from a method won't affect binary compatibility. It also covers more complex rules, so for instance it is possible to override an inherited method or rearrange fields in a class without affecting compatibility.

Note that binary compatibility and source compatibility differ. For instance, it is binary compatible to change a field's access from `protected` to `public`. This is not source compatible in some situations.

Binary Compatibility has a great promise: with a few restrictions, you will never have to recompile libraries again.

## 1 Why we want it

Initially, the gcj project paid no attention to Chapter 13. In practice we implemented a more static language than Java, and it looked as if it would be difficult to get good performance from pre-compiled code that adhered to the binary compatibility rules.

This led to one important restriction on gcj-compiled code, namely that two classes with the same name could not both be loaded at once: this is PR 6819 [4]. Over time, this has proved to be more and more difficult to work around. For instance, in 2003 we split out some libraries from libgcj because some programs shipped their own copies; this in turn caused other problems.

Another important problem we tried to solve in 2003 was the proper operation of class loaders. As it turned out, class loading and binary compatibility are related, and we realized we could solve both problems with the same implementation.

In particular, sophisticated applications such as Eclipse rely on Java's lazy loading and linking capabilities to control class loading and visibility. It isn't possible both to satisfy the proper semantics of a class loader and to have ordinary ELF-style linking.

The Java language gives programmers facilities that go far beyond what is possible in more conventional programming languages. For example, you may define a class loader to load your own classes into the virtual machine. Your class loader will have its own name space and it will inherit classes from the base Java class loader but its own loaded classes will not be externally visible. You can define your own scheme for resolving symbols.

It is quite possible for the same Java class to be loaded several times by several different class

loaders, and in each case its references will be resolved differently.

When a class is loaded, references it makes to other classes are not immediately resolved. This allows mutually dependent classes to be loaded, and later fixed up by calling `resolveClass`.

All of this is a very long way from what can be achieved by using conventional ELF linkage.

## 2 Implementation

The implementation of a new binary compatibility ABI for gcj began several years ago with the work of Bryce McKinlay, and the paper Yu [1].

The basic idea behind our implementation approach is to put all references made by a class into two special tables, called the `atable` and the `otable`.

The `otable`, or Offset Table, is a table of offsets from some base pointer. The `atable`, or Address Table, is a table of absolute addresses. Every class has an `atable` and an `otable`. Initially these tables are filled with symbolic references. Later, when the class is linked, these symbolic references are turned into offsets or addresses, as appropriate.

### 2.1 Class references

Class references are handled via the constant pool, a table that already existed in the old ABI. Entries in the constant pool are resolved when a class is prepared; an operation like `new` or `instanceof` refers to an entry in the pool.

### 2.2 Static methods and fields

A static method or field is referred to via the `atable`. Each symbolic entry in the `atable` consists of three parts: a class name, a member name, and a type signature. At class preparation time, the appropriate class and member are found, access checks are done, and then the address of the member is written into the `atable` slot. So, code in Class A that refers to a static member of Class B does so via an index into the `atable` belonging to Class A.

If a static method is not found, we simply write the address of a function which will throw the appropriate exception.

If a static field is not found, we throw an `IncompatibleClassChangeError` at class preparation time. In Yu [1] this is mentioned as a bug in the design; however, we believe that this behavior is specifically allowed by the linking rules in section 12.3 of the Java Language Specification [2].

### 2.3 Instance methods

Instance methods are handled via the `otable`, not the `atable`. Like the `atable`, the `otable` holds class names, member names, and type signatures. However, instead of mapping these to addresses, it instead maps them to offsets.

When computing the value of an `otable` slot for an instance method, we load and lay out the target class and all its superclasses as well. As part of this process, we compute the target class's vtable; from this we find the correct value to put in the `otable` slot.

Old-ABI code calls virtual methods like:

```
(((vtable *) obj)[index]) (obj, ...)
```

With the new ABI, this is transformed to:

```
(((vtable *) obj)[otable[index]])
                 (obj, ...)
```

If an instance method is not found, we put a special value into the `otable` slot which, when the vtable lookup is done, results in a call to a method that throws `IncompatibleClassChangeError`.

### 2.4 Instance fields

Instance fields are handled similarly to instance methods. Where old ABI code compiles a field reference:

```
*((type *) (obj + offsetof (field)))
```

the new ABI produces the equivalent of:

```
*((type *) (obj +
            otable[field_index]))
```

Although this is an extra memory reference, it is less painful than might first appear: the `otable` and `atable` have good locality, typically being referred to many times in a method.

Note that because all class layout is done dynamically, even references to one's own private fields must go through the `otable`, as one's superclass might add or remove fields and this will change the offsets of all subclass fields.

### 2.5 Interfaces

Interface dispatch also requires an extra indirection via the `otable`, and it requires us to compute interface dispatch tables at runtime, much as we compute the vtables and class layout at runtime.

### 2.6 Exception handlers

For `catch` clauses we write a class name (instead of a reference to a Class object) into the DWARF-2 exception table. The class name is suitably mangled so that the type matching function for a catch block can distinguish between old and new ABI code.

When an exception is thrown, these class names are looked up by the appropriate class loader and turned into references to the corresponding classes.

### 2.7 Versioning

gcj still statically generates an instance of Class for each class that is compiled. In the future we plan instead to generate a class descriptor, which will be instantiated as a Class at runtime. This will insulate compiled code from changes to java.lang.Class, and it will also make it slightly easier for us to handle ABI versioning. We intend to add an ABI version number to the class descriptor, and then let the runtime library handle compatibility as desired.

### 2.8 libgcj API

Compiled code must still make references to symbols exported from libgcj. For instance, operations such as `new` or `instanceof` are implemented by means of exported `_Jv_` functions; the compiler generates direct calls to these functions.

We have considered redirecting calls to these functions via the `atable` as well, but as there are only twenty or so it seems simpler to handle these according to the usual versioning rules for shared libraries.

Compiled code continues to know the layout of array types. We don't anticipate arrays changing incompatibly.

We plan to continue to compile parts of the core library—in all likelihood at least `java.lang` and `java.io`—using the old ABI. Ap-

plication code cannot portably replace these classes, so there is no drawback to compiling them old-style.

### 2.9 Bytecode Verification

One related problem is that of bytecode verification with an ahead-of-time compiler.

In Java, the compile-time and runtime environments might be very different. In order to handle this and still ensure runtime type safety, a typical JVM will perform bytecode verification in the runtime environment.

gcj includes a bytecode verifier as part of its compilation, when compiling from bytecode to object code. However, this is insufficient when the bytecode can be loaded into an arbitrary runtime environment. In particular it would be possible to construct an environment where all the requirements of the compiled code (names of types and methods) are met, but where the result allows subversion of the type system.

For example, a class f might be defined:

```
class f implements B
{
  ...
```

and a user could write an initializer

```
B thing = new f();
```

but if an incompatible change were made to f

```
class f
{
  ...
```

the variable `thing` would now refer to an object that did not implement B. This is a violation of the type system.

The solution to this is to perform bytecode verification in two steps. The first step, still in gcj, works much like an ordinary verifier. All the "static" properties of bytecode, such as whether the declared stack depth is sufficient, can be verified once. Now, when the verifier is asked to verify a fact about a type or method, it always yields `true`, and adds a "verification assertion" to the generated code.

At runtime, these assertions are verified when the class is linked. This process is much quicker than ordinary bytecode verification, which requires modeling the control flow of the code. These assertions are of the form 'A implements B' or 'A extends B', which are very easy to check.

### 2.10 Type assertions for source code

A similar problem occurs when compiling from Java source to native code. In this situation, there is no verification step to split. Instead, the assertion table is filled based on any implicit upcasts that appear in the source; each such cast represents a constraint on the type hierarchy that must remain true at runtime.

### 2.11 CNI

CNI, the Compiled Native Interface, is a way to write Java `native` methods in C++ with zero overhead. With CNI, Java classes are used to generate C++ header files, which then enable relatively ordinary C++ code to make calls on Java objects.

CNI is also going to require some changes. In essence this will involve duplicating some of the `atable` and `otable` logic from gcj in g++ and arranging for these references to be resolved at runtime when appropriate. We anticipate accomplishing this by emitting static initializers which will register table contributions from the current compilation unit with the libgcj runtime.

We plan to make several other CNI changes

now, while we're changing the ABI, in order to postpone any other needed ABI changes. In particular we plan to introduce smart pointers to allow seamless `NULL`-pointer checking on all platforms, and we plan to tighten the rules about what parts of memory can be assumed to be scanned by the garbage collector.

# 3 Consequences

This approach to binary compatibility has some very interesting consequences for gcj and gcj-compiled code.

### 3.1 gcj as JIT

Due to the new runtime linkage model and the new approach to bytecode verification, gcj can now compile a single .class file in complete isolation. That is, compiling a class file doesn't require gcj to read any other classes, not even `java.lang.Object`. This works because a class file has complete symbolic information about its dependencies—just what the `atable` and `otable` require—and because verification will answer "yes" to any type-related question without actually examining any other types until runtime.

This property in turn lets us use gcj itself as a caching JIT. Conventionally, `ClassLoader.defineClass()` takes an array of bytes that is the binary code for a class and loads it into memory. Instead, we compute a cryptographic checksum of the bytes and use it as a key into a cache of shared libraries. If the class is found, we simply `dlopen()` it. If not, we invoke gcj (which is possible and relatively efficient because we only need the class file in isolation) to put a new shared library in the cache.

We're also considering the possibility of making it easy to prime the libgcj cache. To make existing Java applications run with decent performance, you would then only need to compile each .jar file and copy the resulting .so into the cache. No application changes would be needed. Another approach we're investigating is to change `URLClassLoader` to transparently find shared libraries corresponding to `.jar` files on its class path.

### 3.2 VM independence

The code generated by gcj is also surprisingly VM-independent. It refers to the various tables (`otable`, `atable`, assertion table), and to the small number of libgcj builtin functions known to gcj. This means that gcj-compiled code could easily be loaded into any VM implementing this interface; the biggest assumption is that the runtime includes a conservative garbage collector. Even that may not necessarily be true in the future: a few garbage collection hooks would remove even that requirement.

The generated code is also quite independent of other aspects of the runtime environment, for instance the kernel or libc. It should be possible to compile Java code once, and then simply never recompile it even as the rest of the system, including libgcj, is upgraded.

We're hoping other free Java implementations will adopt this same approach as the basis of a "pluggable JIT" interface.

### 3.3 Performance and Size

It is too early to know the precise impact of the new ABI. For some cases, we know that the penalty will be small: for instance, the cost of a static method invocation via the `atable` is similar to the cost of indirection via the PLT.

On the other hand, we expect some costs to be larger: for example, instance field references

will be more expensive.

The Yu [1] paper quotes an average performance penalty of less than 2%; however, their implementation did not implement field indirection.

## 4 Problems and gotchas

It is possible that important Java programs may rely on the precise link-time behavior of existing VMs. In case it becomes necessary to change our approach, we believe we can emulate the more lazy behavior of other VMs in one of two ways. On machines with the required support, we can map a special, unwriteable memory segment, and then fill `atable` slots with pointers into this area. This approach will let us differentiate between `NULL` pointer traps and invalid field traps, and then throw the appropriate exception. For other platforms, we can add extra instrumentation to the compiled code, at some performance cost.

## 5 Today and Tomorrow

As of this writing, Andrew is still finishing the implementation of the core parts of the new ABI. His work builds on some earlier patches from Bryce, and is checked in on `gcj-abi-2-dev-branch`. Tom hopes to begin work on the verification problem soon.

Andrew has built a demo version of gcj-as-JIT and posted some results to the gcj list; see his post [3]. The results are surprisingly good—a longer startup delay, as would be expected, but performance falling between that of Sun's and IBM's JITs on Linux. We anticipate some useful performance gains from tree-ssa as well, eventually—in particular smarter array bounds checking.

Ideally we would like to see a supported, but perhaps still preliminary, version of this ABI in GCC 3.5, with real compatibility promised starting with 3.6.

## 6 Acknowledgements

We would like to thank Bryce McKinlay for his initial and ongoing work in this area, Jeff Sturm for his contributions to the new ABI project, and Sarah Woodall for editing.

## References

[1] Zhong Shao Dachuan Yu and Valery Trifonov. Supporting binary compatibility with static compilation. In *2nd USENIX Java™ Virtual Machine Research and Technology Symposium (JVM'02)*. Usenix, August 2002. `http://www.usenix.org/publications/library/proceedings/javavm02/yu/yu_html/`.

[2] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley, Boston, Mass., 2000.

[3] Andrew Haley. gcj-jit, 2003. `http://gcc.gnu.org/ml/java/2003-01/msg00022.html`.

[4] Oskar Liljeblad. Pr 6819, 2002. `http://gcc.gnu.org/PR6819`.