Reprinted from the

Proceedings of the GCC Developers' Summit

June 28th–30th, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon Incorporated*
C. Craig Ross, *Linux Symposium*

Review Committee

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

Proceedings Formatting Team

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

Replacement special loop form by a call of built-in function

Tomáš Bílý
SUSE CR
tbily@suse.cz

Abstract

There are some patterns in computer programs (especially loops) which could be successfully replaced by a call of some built-in function and letting the GCC expanders decide which implementation is the best. As the built-in functions are coded specifically for a target platform such replacement can improve the runtime performance.

In this report, we present an implementation of this problem for some special cases of loops which can be transformed to `memset` or `memcpy` call. We also present performance results and a discussion limits of a current implementation.

1 Introduction

One of frequently used optimization methods is a loop transformation. There are some loop constructions with special statement patterns that could be transformed into a built-in function call and an expander infrastructure in GCC chooses the best possible implementation of this function. This is a way how a compiler could possibly take the best implementation of the program constructions for a specific platform and thus a runtime performance can be improved.

Many of program constructions initialize some arrays or copy the content of one array into another array. We were inspired by this phenomenon and we tried to search loops constructions and patterns for matching that could be used for transformation into `memset` or `memcpy` built-in call.

We can see that it is not necessary to transform the whole loop. This case is a particular instance of loop distribution (see [4]) where one of the loops is transformed to built-in call.

Finding these patterns is very similar to a vectorization approach. Some of the constructions could be optimized by use of current auto-vectorization infrastructure. But this could be done only on architectures that can support vector instructions. It is possible that the current auto-vectorization optimization could not be the best way to use.

Basic approach to handle the searching of these patterns is based on the theory of data-dependence analysis (see [1]). It must be examined that reordering the statements could not produce incorrect results.

Current GCC contains implementation of data dependence analyzer, scalar evolution infrastructure (see [2]) and auto-vectorization infrastructure (see [5]). We have heavily used all those infrastructures in our implementation of

outline problem.

Rest of paper is organized as follows. In Section 2 we describe our implementation. In Section 3 we present current measurements of runtime performance. In Section 4 we present summary and we describe some future plans for enhancement.

2 Implementation

We will call *builtinizer* an algorithm implementation that solves pattern matching and loop transformation into built-in call.

Our implementation of builtinizer is developed at the IR level of GIMPLE trees in SSA form (see [3]). Builtinizer are trying to handle

- array references (see Figure 1a or Figure 3a)
- indirect (pointer) references (see Figure 1b or Figure 3b)
- multidimensional arrays
- patterns on every level of nested loops

Patterns that builtinizer currently recognizes are

- `x[]..[] = 0`
- `*x = 0`
- `x[]..[i] = y[]..[i]` or `r = y[]..[i]; x[]..[i] = r;` (in GIMPLE)
- `*p = *q` or `r = *q; *p = r;` (in GIMPLE)

(a) array ref

```
for (i = 0; i < N; i++)
 {
  ...
  x[i] = 0;
  ...
 }
```

(b) pointer ref

```
for (i = 0; i < N; x++, i++)
 {
  ...
  *x = 0;
  ...
 }
```

Figure 1: `memset` built-in call pattern before transformation

```
memset (x, 0, N * sizeof (*x));
for (i = 0; i < N; i++)
 {
  ...
 }
```

Figure 2: `memset` built-in call pattern after transformation

Type of items must be one of the types `char`, `short`, `int`, `float`, `double`.

Current implementation handles an multidimensional arrays in inner most component only.

2.1 builtinizer structure

Builtinizer applies a set of analysis on each loop, followed by the built-in call transformation for the loops that had successfully passed the analysis phase. Examples such transformations you may see in Figure 1 and Figure 2 or Figure 3 and Figure 4.

(a) array refs

```
for (i = 0; i < N; i++)
 {
  ...
  x[i] = y [i];
  ...
 }
```

(b) pointer refs

```
for (i = 0; i < N; x++, y++, i++)
 {
  ...
  *x = *y;
  ...
 }
```

Figure 3: `memcpy` built-in call pattern before transformation

```
memcpy (x, y, N * sizeof (*x));
for (i = 0; i < N; i++)
 {
  ...
 }
```

Figure 4: `memcpy` built-in call pattern after transformation

```
for (i = 0; x [i] == -1; i++)
 {
  x [i] = 0;
 }
```

Figure 5: uncountable loop

2.2 builtinizer analysis

The first analysis phase probes the loop exit condition and number of iterations. Than examine some control-flow attributes (for example nesting level). One major restriction is required for a loop that can be builtinized. This is that loop is countable (an expression that calculates the loop bound could be constructed and evaluated at compile time or at runtime). For example the loop in Figure 5 is not countable loop. The loop bound analysis is done by scalar evolution analyzer.

Next step finds all memory references in the loop and checks if an access function that describes their modification in the loop can be constructed. This informations are required for the memory dependence tester and the access pattern analysis. Dependences that do not cover up memory operations are analyzed directly from SSA representation.

The final analysis phase scans all the statements in the loop and determines if they match pattern rules for transformation to built-in call.

2.3 builtinizer transformation

The analysis phases gather useful information about the loop, the statements and the data references. These informations are used during transformation phase. Data structures that store this informations are used from the vectorizer and the data dependence analyzer. They are

- `loop_vect_info` – holds information at the loop level
- `stmt_vect_info` – holds information at the statement level
- `data_reference` – holds information at the memory reference level

The loop transformation phase scans all accepted statements from the analysis phase and these statements grouped to patterns. The statement groups remove from the loop and insert relevant built-in call statement before the loop. The memory reference statements are implicitly removed by builtinizer but remaining scalar statements (that have some relevance to the removed statements) are expected to be removed by dead code elimination.

Figure 6 illustrates the transformation process. First, statements are grouped (by a pattern recognition) to $G_1 = \{S1\}$ and $G_2 = \{S3, S4\}$. Then group G_1 is transformed to `memset` built-in call (see Figure 6b) and group G_2 is transformed to `memcpy` built-in call (see Figure 6c).

(a) before builtinization

```
for (i = 0; i < N; i++)
 {
S1:  x[i] = 0;
S2:  y[2*i] = 0;
S3:  a = z[i];
S4:  w[i] = a;
     ...
 }
```

(b) after builtinization of S1

```
memset (x, 0, N * sizeof (*x));
for (i = 0; i < N; i++)
 {
S2:  y[2*i] = 0;
S3:  x = z[i];
S4:  w [i] = x;
     ...
 }
```

(c) after builtinization of S3 and S4

```
memset (x, 0, N * sizeof (*x))
memcpy (w, z, N * sizeof (*z))
for (i = 0; i < N; i++)
 {
S2:  y[2*i] = 0;
     ....
 }
```

Figure 6: The transformation process

3 Experimental results

We have gathered current experimental results. Testing systems were Pentium-M 1.6 GHz and AMD Athlon 64 X2 Dual Core Processor 2200. List of Pentium runtime performance results is in Table 1. List of Athlon runtime performance results is in Table 2. Numbers of `memset` and `memcpy` patterns recognized are in Table 3.

4 Conclusion and future plans

As could be seen in experimental results the current implementation improve runtime performance slightly in some cases. Almost all

loops in test-cases have small number of iterations then they are handled well by current loop optimizations.

In the future we will improve handling of multidimensional arrays, we will expand variety of `memset` filling constant and we will try to extend number of patterns for matching.

5 Acknowledgments

We would like to thank Andrew Pinski because this work is based on his patch (see [6]) and to Honza Hubička for his helpful advice.

References

[1] Randy Allen and Ken Kennedy. *Optimizing Compiler for Modern Architectures: A dependence based approach*. Morgan Kaufmann, 2001.

[2] Sebastian Pop Daniel Berlin, David Edelsohn. High-level loop optimization for gcc. In *Proceedings of the 2004 GCC Developer's Summit*, 2004.

[3] Free Software Foundation. Gcc internals manual. `http://gcc.gnu.org/onlinedocs/gccint/`.

[4] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[5] Dorit Naishlos. Autovectorization in gcc. In *Proceedings of the 2004 GCC Developer's Summit*, 2004.

[6] Andrew Pinski. [patch] [improvements branch?] loops to memset. `http://gcc.gnu.org/ml/gcc-patches/2004-09/msg00873.html`.

GCC options	without builtinizer (average result)	with builtinizer (average result)
tramp3d -n 50		
`-O2 -ffast-math`	3m03.58s	3m03.09s
`-O3 -ffast-math`	2m44.90s	2m44.88s
SPEC2000 164.gzip		
`-O2 -ffast-math`	188 (745)	186 (751)
`-O3 -ffast-math`	186 (753)	184 (762)
SPEC2000 175.vpr		
`-O2 -ffast-math`	146 (958)	146 (960)
`-O3 -ffast-math`	142 (986)	141 (993)
SPEC2000 181.mcf		
`-O2 -ffast-math`	209 (862)	209 (862)
`-O3 -ffast-math`	198 (910)	198 (910)
SPEC2000 253.perlbmk		
`-O2 -ffast-math`	156 (1151)	163 (1104)
`-O3 -ffast-math`	157 (1148)	163 (1105)
SPEC2000 300.twolf		
`-O2 -ffast-math`	226 (1325)	220 (1358)
`-O3 -ffast-math`	213 (1406)	218 (1373)
SPEC2000 177.mesa		
`-O2 -ffast-math`	186 (751)	184 (762)
`-O3 -ffast-math`	175 (802)	173 (814)
SPEC2000 179.art		
`-O2 -ffast-math`	159 (1639)	158 (1645)
`-O3 -ffast-math`	150 (1737)	149 (1740)
SPEC2000 188.ammp		
`-O2 -ffast-math`	323 (681)	322 (684)
`-O3 -ffast-math`	323 (681)	321 (686)

Table 1: Pentium-M experimental results

GCC options	without builtinizer (average result)	with builtinizer (average result)
tramp3d -n 100		
`-O2 -ffast-math`	1m51.04s	1m50.80s
SPEC2000 164.gzip		
`-O2 -ffast-math`	122 (1146)	121 (1150)
`-O3 -ffast-math`	123 (1141)	120 (1166)
SPEC2000 175.vpr		
`-O2 -ffast-math`	137 (1020)	135 (1034)
`-O3 -ffast-math`	134 (1044)	133 (1054)
SPEC2000 181.mcf		
`-O2 -ffast-math`	283 (636)	283 (636)
`-O3 -ffast-math`	281 (640)	281 (640)
SPEC2000 252.eon		
`-O2 -ffast-math`	74.2 (1752)	70.6 (1842)
`-O3 -ffast-math`	58.0 (2240)	58.0 (2241)
SPEC2000 253.perlbmk		
`-O2 -ffast-math`	129 (1390)	133 (1358)
`-O3 -ffast-math`	135 (1114)	136 (1103)
SPEC2000 300.twolf		
`-O2 -ffast-math`	259 (1159)	255 (1175)
`-O3 -ffast-math`	256 (1172)	255 (1177)
SPEC2000 177.mesa		
`-O2 -ffast-math`	106 (1325)	99.9 (1402)
`-O3 -ffast-math`	96.7 (1448)	96.6 (1450)
SPEC2000 179.art		
`-O2 -ffast-math`	191 (1360)	188 (1379)
`-O3 -ffast-math`	191 (1358)	192 (1357)
SPEC2000 188.ammp		
`-O2 -ffast-math`	165 (1333)	164 (1338)
`-O3 -ffast-math`	166 (1329)	164 (1339)

Table 2: Athlon experimental results

program name	# `memset` occurrences	# `memcpy` occurrences
tramp3d	102	125
SPEC2000 164.gzip	10	3
SPEC2000 175.vrp	4	0
SPEC2000 181.mcf	0	0
SPEC2000 186.crafty	47	3
SPEC2000 252.eon	6	23
SPEC2000 256.bzip	8	2
SPEC2000 300.twolf	5	3
SPEC2000 177.mesa	21	84
SPEC2000 179.art	1	3
SPEC2000 183.equake	15	9
SPEC2000 188.ammp	13	5

Table 3: number of `memset` and `memcpy` pattern occurrences