# An Evaluation of the New Framework for Loop Nest Optimization in GCC

Sebastian Pop[1], Albert Cohen[2], and Georges-André Silber[1]

[1] Centre de recherche en informatique, École des mines de Paris, Fontainebleau, France
[2] ALCHEMY group, INRIA Futurs, Orsay, France

**Abstract.** This paper presents an evaluation of the new framework for loop nest optimization of the GNU Compiler Collection (GCC). Multiple structural and experimental aspects of this infrastructure are considered: performance of the generated code, effectiveness of the analyzers and optimizers, compilation overhead, together with possible improvements. The evaluation is based on the SPEC CPU2000 and JavaGrande benchmarks.

## 1 Introduction

The GNU Compiler Collection (GCC) is the leading compiler suite for the free and open source software. The large set of target architectures and the standard compliant front ends makes GCC the de facto compiler for portable developments and system design. However, until recently, GCC was not an option for high performance computing: it had no infrastructure for data access restructuring nor for automatic parallelization. Furthermore, until recently, the compilation research community had no robust, universal, free, and industry supported platform where new ideas in compiler technology can be implemented and tested on large scale programs written in standard programming languages. For more than a decade, the field of advanced compilation has been plagued by redundant development efforts on short-lived projects, with little impact on production environments.

In this paper, we propose an overview and a preliminary evaluation of the new infrastructure of GCC for analysis and loop nest optimizations. Based on this evaluation we aim to show the potential of GCC to address the needs of a free compiler in the compiler research and high performance computing communities.

Modern compilers implement some of the sophisticated optimizations introduced for supercomputing applications [17, 2]. They provide performance models and transformations to improve fine-grain parallelism and to take into account the memory hierarchy. Most of these optimizations are loop-oriented and assume a high-level code representation with rich control and data structures: `do` loops with regular control, constant bounds and strides, typed arrays with linear subscripts. Yet these compilers are architecture-specific and designed by processor vendors, e.g., IBM, SGI, HP and Intel. In addition, the most advanced optimizations are limited to Fortran and C, and performance is dependent on the recognition of specific patterns in the source code. Some source-to-source compilers implement advanced loop transformations driven by architecture models and profiling [8]. However, good optimizations require manual efforts

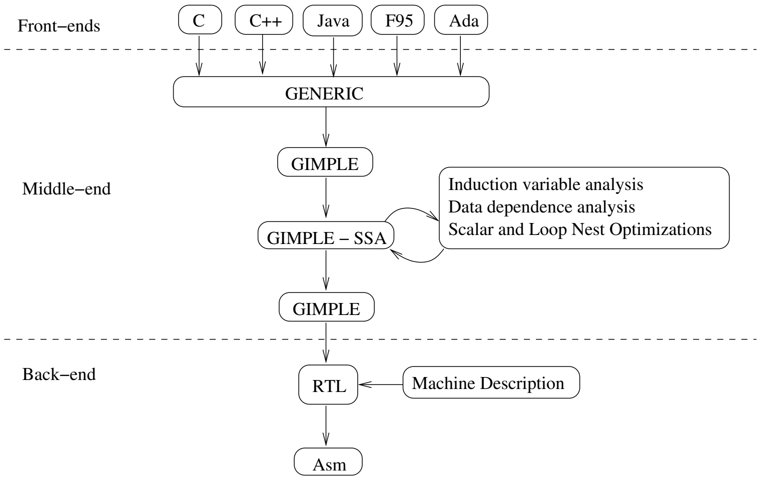


**Fig. 1.** The infrastructure of GCC 4.0.

in the syntactic presentation of loops and array subscripts (avoiding, e.g., `while` loops, imperfect nests, linearized subscripts, pointers). In addition, the source-to-source approach is not suitable for low-level optimizations, including vectorization for SIMD instructions and software pipelining. It is also associated with pattern-based strategies that do not easily adapt to syntactic variations in the programs. Finally, these compilers require a huge implementation effort that cannot be matched by small development teams for general-purpose and/or free software compilers.

Several works demonstrated the interest of type enriched low-level representation [9]. They build on the normalization and simplicity of three address code, adding data types, *Static Single-Assignment* (SSA) form [5, 12] to ease data-flow analysis and scalar optimizations, and control and data annotations (loop nesting, heap structure, etc.). Recently, GCC 4.0 adopted such a representation called GIMPLE [14, 11], a three-address code derived from SIMPLE, the representation of the McCAT compiler [7]. The GIMPLE representation was proposed by Sebastian Pop, and Diego Novillo from RedHat, for minimizing the effort in the development and the maintenance of new analyzes and transformations. GIMPLE is used for building the SSA representation that is then used by the analyzers and the scalar and loop nest optimizers. After these optimizations, the intermediate representation is translated into RTL, where machine specific informations are added to the representation. Figure 1 presents the overall structure of GCC.

The front ends C, C++, Java, Fortran95 and Ada are translating their intermediate representation to a common representation called GENERIC, then to GIMPLE that is eventually translated to RTL. Before presenting the loop optimizations that we will evaluate, we give an overview of the GIMPLE intermediate representation on which the loop optimizers are acting.

A three-address representation like GIMPLE seems a priori not suitable to implement program transformations for high performance: control-flow is represented by only two primitives, a conditional expression `if`, and a goto expression `goto`. Loops have to be discovered from the control-flow graph [1]. Although the number of iterations is not captured by the GIMPLE representation, it is often discovered by analyzing the scalar variables involved in the loop exit conditions; this allows to compute precise information lost in the translation to a low-level representation, but it may also be useful when the source level does not expose enough syntactic information, e.g., in `while` loops, or loops with irregular control constructs such as `break` or exceptions. In GIMPLE, subscript expressions, loop bounds and strides are spread across a number of elementary instructions and basic blocks, possibly far away from the original location in the source code. This is due to the lowering of the intermediate representation and to optimization phases such as dead-code elimination, partial redundancy elimination, optimization of the control flow, invariant code motion, etc. However, the SSA form provides fast practical ways to extract information concerning values of scalar variables.

We recall some SSA terminology [5, 12] to facilitate further discussions: the *SSA graph* is the graph of def-use chains in the SSA form; each assignment targets a unique variable; $\phi$ nodes occur at merge points of the control flow and restore the flow of values from the renamed variables. Figure 2 illustrates the role of a $\phi$ node used for merging values assigned in different branches of a condition expression. Assuming the natural loops [1] information has been recovered, $\phi$ nodes whose arguments are defined at different loop depths are called *loop-*$\phi$ nodes and they have a specific semantics related to a potentially recursive definition. In Figure 3, that presents the syntax and the semantics of a loop-$\phi$ node, variable $b$ is self defined in the argument coming from the loop body.

We developed an analyzer [15] that extracts this semantics from the SSA graph and represents successive values of $b$ in a more compact way. The algorithm reconstructs the loop-$\phi$ node as $b = loop1\phi\ (0, b+7)$, leading to an explicit view of the evolution of $b$ in the loop: an initial value 0 and an increment of 7 for each step of the loop. We survey this analyzer in the next section of this paper. In Section 3, we present several analyzes that use our evolution analysis, and in Section 4, we present preliminary performance results obtained with loop optimizations based on the GIMPLE-SSA representation and using our analysis framework.

```
v = 5
if (E)
  then v = 0
  else v = 1
endif
... = v
```

$\xrightarrow{SSA}$

```
v = 5
if (E)
  then a = 0
  else b = 1
endif
c = ϕ (a, b)
... = c
```

$\xrightarrow{Sem}$

$$v = 5$$
$$a = 0$$
$$b = 1$$
$$c = \begin{cases} \text{a} & \text{if } E \text{ is true,} \\ \text{b} & \text{otherwise.} \end{cases}$$

**Fig. 2.** Syntax and semantics of a $\phi$-node.

```
a = 0
f = 7
begin1:
  b = loop1ϕ (a, c)
  c = b + f
  if (E)
    then goto end1
    else goto begin1
  endif
end1:
```

$\xrightarrow{Sem}$

$$
\begin{aligned}
b(0) &= a \\
b(x) &= c(x-1) \text{ for } x > 0 \\
b(x) &= b(x-1) + f(x-1) \text{ for } x > 0 \\
&\ \ \cdots \\
b(x) &= a + \sum_{j=0}^{x-1} f(j) \text{ for } x \geq 0 \\
b(x) &= a + x \cdot f \text{ for } x \geq 0 \\
b(x) &= 7x \text{ for } x \geq 0
\end{aligned}
$$

**Fig. 3.** Syntax and semantics of a loop-ϕ node.

## 2 Induction variables analysis

An induction variable is a variable that is updated in one or more loops. Sometimes, it is possible to characterize by an affine function the sequence of values that are successively assigned to an induction variable. In this case, loop analyzers and optimizers need a more compact information than the SSA graph, under the form: initial value and step in the loop. We have added in GCC an induction variable framework that abstracts the information available in the SSA graph, and corresponds to the extraction of the semantics of scalar variables updated in loops as illustrated in Figure 3. The induction variable analysis we implemented [4, 15] in GCC has been inspired by the chains of recurrences [3], and allows the same expressiveness and applications as the framework presented by van Engelen [16]. The major interest of the chains of recurrences mathematical framework is to provide a fast computation of any value taken by an induction variable at any step of its enclosing loops [3], allowing powerful analyzers to be developed.

For evaluating the precision of our analyzer, we have performed the following experiment: the analyzer was asked to determine a compact representation of all the loop-ϕ nodes in all the loops for the SPEC CPU2000 and JavaGrande benchmarks. Figure 4 gives a classification of induction variables handled by our analyzer. The first column is the benchmark name. The "Total" column counts the number of loop-ϕ for which we try to build a compact representation of their evolution. The next 5 columns describe the break-down of these scalar evolutions into the following classes: affine univariate (i.e. the variable has an evolution in only one loop), affine multivariate (i.e. the variable has an evolution in two or more nested loops), other kinds of polynomials, other compound expressions containing determined components, but that cannot be reduced to a compact polynomial form, and undetermined evolutions ($\top$). More details about those representations are given in [15].

| **CINT2000** | **Total** | **A.U.** | **A.M.** | **O.P.** | **Other** | ⊤ | **Loops** | **Trip** | **Approx.** | ⊤ |
|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|---:|
| `164.gzip` | 1138 | 285 | 0 | 119 | 144 | 590 | 223 | 90 | 4 | 55 |
| `175.vpr` | 4084 | 1384 | 0 | 321 | 575 | 1804 | 505 | 109 | 0 | 207 |
| `176.gcc` | 18403 | 2835 | 4 | 1646 | 1636 | 12282 | 3506 | 663 | 25 | 982 |
| `181.mcf` | 80 | 21 | 0 | 7 | 38 | 14 | 72 | 2 | 0 | 32 |
| `186.crafty` | 2503 | 513 | 6 | 101 | 386 | 1497 | 464 | 119 | 13 | 192 |
| `197.parser` | 2220 | 564 | 0 | 120 | 450 | 1086 | 786 | 67 | 22 | 308 |
| `252.eon` | 2115 | 601 | 0 | 164 | 146 | 1204 | 541 | 186 | 6 | 49 |
| `253.perlbmk` | 8048 | 919 | 1 | 791 | 380 | 5957 | 993 | 155 | 0 | 297 |
| `254.gap` | 26913 | 3782 | 9 | 3051 | 1115 | 18956 | 2143 | 350 | 10 | 773 |
| `255.vortex` | 3106 | 479 | 0 | 308 | 278 | 2041 | 332 | 19 | 2 | 81 |
| `256.bzip2` | 42 | 26 | 0 | 4 | 2 | 10 | 14 | 2 | 0 | 4 |
| `300.twolf` | 10536 | 1577 | 0 | 895 | 849 | 7215 | 1014 | 47 | 0 | 699 |
| **CFP2000** | **Total** | **A.U.** | **A.M.** | **O.P.** | **Other** | ⊤ | **Loops** | **Trip** | **Approx.** | ⊤ |
| `168.wupwise` | 235 | 94 | 0 | 12 | 123 | 6 | 84 | 70 | 0 | 1 |
| `171.swim` | 191 | 84 | 0 | 26 | 22 | 59 | 32 | 25 | 0 | 0 |
| `172.mgrid` | 962 | 165 | 0 | 420 | 157 | 220 | 71 | 58 | 0 | 0 |
| `173.applu` | 1808 | 645 | 5 | 630 | 199 | 329 | 184 | 150 | 0 | 12 |
| `177.mesa` | 10308 | 4100 | 1 | 1304 | 1123 | 3780 | 1158 | 689 | 34 | 124 |
| `179.art` | 496 | 213 | 2 | 72 | 64 | 145 | 100 | 38 | 0 | 33 |
| `183.equake` | 1212 | 438 | 2 | 104 | 102 | 566 | 105 | 52 | 1 | 24 |
| `188.ammp` | 1632 | 649 | 6 | 102 | 286 | 589 | 546 | 169 | 14 | 34 |
| `189.lucas` | 1901 | 250 | 0 | 425 | 121 | 1105 | 109 | 68 | 1 | 31 |
| `191.fma3d` | 6759 | 3691 | 14 | 593 | 1112 | 1349 | 2835 | 1971 | 17 | 493 |
| `200.sixtrack` | 8549 | 1723 | 17 | 2298 | 781 | 3730 | 1109 | 563 | 0 | 274 |
| `301.apsi` | 3986 | 1087 | 5 | 1190 | 785 | 919 | 387 | 284 | 0 | 15 |
| **JavaGrande v2.0** | **Total** | **A.U.** | **A.M.** | **O.P.** | **Other** | ⊤ | **Loops** | **Trip** | **Approx.** | ⊤ |
| `section1` | 777 | 93 | 0 | 0 | 396 | 288 | 201 | 79 | 0 | 77 |
| `section2` | 311 | 121 | 0 | 20 | 19 | 151 | 107 | 2 | 0 | 3 |
| `section3` | 567 | 120 | 0 | 2 | 18 | 427 | 173 | 3 | 0 | 6 |

**Fig. 4.** Scalar induction variables and loop trip count in SPEC CPU2000 and JavaGrande benchmarks.

## 3 Analyzes based on induction variable analysis

*Unification of wrap around loop-$\phi$.* A wrap around loop-$\phi$ is a loop-$\phi$ that is not self defined. In other words, the value assigned by the loop-$\phi$ node during the first iteration frequently has no relation with the values assigned in the subsequent iterations. As illustrated in Figure 5, the initial value of variable $a$ is the value contained in variable $b$, and it is not involved in the definition of the values of $a$ for iterations $x > 0$. In some cases, the first value can be unified with subsequent values, and the wrap around loop-$\phi$ can be rewritten such that it becomes a simple induction variable (i.e. a self defined loop-$\phi$). We have presented the details of this method in a research report [15]. Figure 6 gives an example of a simple unification.

The importance of this method is illustrated by the number of cases that occur in benchmarks: in the SPEC CPU2000 we have found 29 wrap around loop-ϕ that can be unified, on the GCC code itself we have found 337 unification opportunities, and on the JavaGrande benchmarks we have found 5 occurrences. In all these cases, the subsequent passes that are using the induction variable information are ineffective if the unification is not performed: they conservatively reject the wrap around cases. A possible interpretation of the important number of unifications is that a part of the wrap around loop-ϕ are created by one of the previous optimization passes.

```
a = loop1ϕ (b, c)
c = loop1ϕ (d, c + 1)
```

$\xrightarrow{Sem}$

$$
\begin{aligned}
a(0) &= b \\
a(x) &= c(x-1) \text{ for } x > 0 \\
c(x) &= d + x \text{ for } x \geq 0 \\
a(x) &= d + x - 1 \text{ for } x > 0
\end{aligned}
$$

**Fig. 5.** Syntax and semantics of a wrap around loop-ϕ: *a*.

```
a = loop1ϕ (0, c)
c = loop1ϕ (1, c + 1)
```

$\xrightarrow{Rewrite}$

```
a = loop1ϕ (0, a + 1)
c = loop1ϕ (1, c + 1)
```

**Fig. 6.** Wrap around loop-ϕ *a* can be rewritten as a simple induction variable.

*Computation of the loop-trip count.* Because loops at GIMPLE level are represented by goto constructs, the number of iterations in a loop can only be computed based on induction variable information. It corresponds to the first iteration that satisfies the loop's exit condition. This information can be statically computed in some cases as presented in the last columns of Figure 4. These columns describe respectively the number of *natural* loops in the benchmark, the number of *single-exit* loops whose trip count is successfully analyzed, the number of loops where a precise trip count cannot be determined, but for which it is possible to give an upper bound approximation based on the accessed data sizes in the loop, and the number of *single-exit* loops where we fail to compute the trip count.

*Data dependence analysis.* The design of the data dependence analysis is based on a compromise between the precision and the compilation time effectiveness. The results of the data dependence tests are obtained on demand for a loop nest, such that the analysis can be restricted to the loop to be optimized. Furthermore, the dependence tests are ordered such that the fastest tests are executed first, then more expensive analyzes are triggered if the previous ones fail.

In order to show the effectiveness of the data dependence analyzer, we have measured the compilation time of the vectorization pass that make use of the data dependence information. For the SPEC CPU2000 benchmarks, the vectorization pass does not exceed 1 second per compiled file, nor 5 percent of the compilation time per file, showing that the dependence analyzer is fast in practice. This experiment was performed on a Pentium4 2.40 GHz.

Figure 7 shows an evaluation of the precision of the information provided by the analyzer while computing the data dependence relations for all the array accesses in a function. Trivial dependence tests, such as when independence is deduced from accesses to different arrays, are not included in this table. First column is the name of the benchmark, the second column gives the total number of dependence tests. Following columns give: the number of tests that have been classified as dependent, independent, undetermined (either because the analyzer is too weak, or because the test is not decidable at compile time). Following columns split the dependence tests into zero induction variable (ZIV) tests (i.e. the array is referenced by a constant), single induction variable (SIV) tests and multiple induction variables (MIV) tests [2].

We have to stress that this evaluation is quite artificial because an optimizer would focus the data dependence analysis only on a few loop nests. The number of dependence tests and the MIV column witness the stress on the dependence analyzer: an important number of tests involve arrays accessed in different loops, that could be successive loops separated by an important number of statements. Even with these extreme test conditions, our data dependence analyzer catches an important number of dependence relations, and the worst case of computation time for the SPEC CPU2000 is 15 seconds and 70 percent of the compilation time on a Pentium4 2.40 GHz.

## 4 Evaluation of existing optimizations

Based on our induction variable analysis, several scalar optimizations have been contributed, e.g. by Zdeněk Dvořák from SuSE: strength reduction, induction variable canonicalization and elimination, loop invariant code motion and other classic scalar loop optimizations [1]. These are enabled by default with the option `-O2` in GCC.

In order to use the vector units of the target architecture, such as the Altivec, SSE or MMX, the "simdization" pass [6] recognizes loop patterns that can be rewritten using SIMD instructions. This pass is enabled with the option `-ftree-vectorize` and has been contributed by Dorit Naishlos [13] from IBM Haifa.

A linear loop transformation framework has been recently implemented in GCC, but for the moment only the loop interchange transformation is supported [4]. The loop interchange pass has been contributed by Daniel Berlin and Sebastian Pop. The heuristic that selects the profitability for the loop interchange is very simple, but is sufficient for catching the interchange in the SPEC CPU2000 `swim` benchmark.

Figure 8 presents the results of the previous optimizations on the SPEC CPU2000 benchmarks. We have used *gcc version 4.0.0 20050131 (experimental)* on a Pentium4

| CINT2000 | # tests | d | i | u | ZIV | SIV | MIV |
|---|---|---|---|---|---|---|---|
| 164.gzip | 2130 | 228 | 301 | 1601 | 380 | 53 | 43 |
| 175.vpr | 1465 | 362 | 452 | 651 | 542 | 147 | 69 |
| 176.gcc | 91493 | 29022 | 30040 | 32431 | 52955 | 2041 | 2783 |
| 181.mcf | 147 | 6 | 1 | 140 | 2 | 2 | 1 |
| 186.crafty | 38387 | 5967 | 8544 | 23876 | 12354 | 1210 | 705 |
| 197.parser | 1609 | 296 | 40 | 1273 | 103 | 45 | 44 |
| 252.eon | 97386 | 34474 | 62080 | 832 | 96369 | 1078 | 19 |
| 253.perlbmk | 12345 | 733 | 534 | 11078 | 993 | 32 | 693 |
| 254.gap | 35084 | 1273 | 1798 | 32013 | 3065 | 495 | 725 |
| 255.vortex | 15303 | 129 | 70 | 15104 | 138 | 10 | 7 |
| 256.bzip2 | 13 | 1 | 0 | 12 | 0 | 0 | 0 |
| 300.twolf | 7873 | 689 | 1404 | 5780 | 2041 | 188 | 45 |
| **CFP2000** | **# tests** | **d** | **i** | **u** | **ZIV** | **SIV** | **MIV** |
| 168.wupwise | 818 | 6 | 0 | 812 | 0 | 0 | 6 |
| 171.swim | 677 | 212 | 57 | 408 | 9 | 29 | 197 |
| 172.mgrid | 2014 | 39 | 20 | 1955 | 23 | 17 | 0 |
| 173.applu | 32786 | 3312 | 9109 | 20365 | 450 | 1074 | 13081 |
| 177.mesa | 302103 | 14414 | 19600 | 268089 | 26379 | 5705 | 38411 |
| 179.art | 168 | 51 | 19 | 98 | 49 | 6 | 8 |
| 183.equake | 845 | 417 | 216 | 212 | 308 | 176 | 210 |
| 187.facerec | 9185 | 284 | 102 | 8799 | 206 | 76 | 25 |
| 188.ammp | 19578 | 3443 | 8846 | 7289 | 10899 | 1170 | 692 |
| 189.lucas | 93946 | 221 | 31 | 93694 | 74 | 86 | 26 |
| 191.fma3d | 54013 | 14841 | 18448 | 20724 | 19874 | 6867 | 4029 |
| 200.sixtrack | 130773 | 9657 | 41939 | 79177 | 46935 | 2260 | 3518 |
| 301.apsi | 8149 | 1006 | 295 | 6848 | 223 | 434 | 340 |
| **JavaGrande v2.0** | **# tests** | **d** | **i** | **u** | **ZIV** | **SIV** | **MIV** |
| section1 | 73028 | 9247 | 63767 | 14 | 72999 | 0 | 14 |
| section2 | 1926 | 386 | 430 | 1110 | 212 | 411 | 83 |
| section3 | 12185 | 3724 | 3169 | 5292 | 3043 | 2230 | 819 |

**Fig. 7.** Classification of data dependence tests in SPEC CPU2000 and JavaGrande.

2.40 GHz with 512 Kb of cache, 2 GB of RAM, on a Debian Sarge with a Linux kernel 2.6.8. Second column gives the SPEC CPU2000 scores for the base compiler flags: `-O2`, third column gives the SPEC CPU2000 scores for the peak compiler flags: `-O2 -ftree-vectorize -msse2 -ftree-loop-linear`, and last column gives the percentage of improvement. The results for all the other benchmarks not included in the table are between $-2.13$ and $2.59$ percent of improvement. Overall scores are: CINT2000 base at 654, peak at 663 (1.3%); CFP2000 base at 521, peak at 539 (3.4%).

The improvement in the `swim` benchmark is due to the loop interchange that optimizes the locality of the accessed data in the loop, the other improvements are due to the vectorizer. Our guess is that the slowdown for `wupwise` could be the result of a code size increase: the heuristic used by the vectorizer for loop unswitching could be improved.

| Benchmark | Base | Peak | % |
|---|---|---|---|
| 168.wupwise | 638 | 577 | -9.56 |
| 171.swim | 796 | 1320 | 65.83 |
| 177.mesa | 447 | 756 | 69.13 |
| 253.perlbmk | 717 | 773 | 7.81 |
| 300.twolf | 572 | 605 | 5.77 |

**Fig. 8.** Performance of generated code.

## 5 Conclusion and perspectives

As we have seen through the paper, the infrastructure of GCC for loop nest optimizations is quite new, and more work is needed to enhance the preliminary results. However, it constitutes a starting basis for a complete infrastructure for advanced loop restructuring. The paper has given some experimental results for illustrating the precision of the induction variable and data dependence analyzers, then the results of two loop optimizers: the vectorizer and the linear loop transformation framework. We have shown that the compiler is robust for a large set of benchmarks written in several programming languages (C, C++, Java, Fortran) and has a low overhead compilation time.

We still work on refining the data dependence tests, as optimizations are needing more precision for enabling more transformations. More precisely, we are working on better support for the MIV case, the handling of symbolic dependence tests, and the embedding of polyhedral techniques [10, 18] to refine dependence information.

GCC is much appreciated for its excellent production quality, standards compliance and speed. We believe we have shown that GCC has all the merits for academic research and for target specific research and development in industry. We hope that the high performance and compilation communities are going to use GCC as their main research and implementation platform.

## 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*. Morgan and Kaufman, 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. D. Berlin, D. Edelsohn, and S. Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004. `http://www.gccsummit.org/2004`.
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. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.
6. A. E. Eichenberger, P. Wu, and K. O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82–93. ACM Press, 2004.
7. L. Hendren, C. Donawa, M. Emami, G. R. 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*, number 757 in LNCS, pages 406–420. Springer-Verlag, 1993.
8. KAP C/OpenMP for Tru64 UNIX and KAP DEC Fortran for Digital UNIX. `http://www.hp.com/techsevers/software/kap.html`.
9. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar. 2004.
10. V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), Dec. 1997. `http://icps.u-strasbg.fr/PolyLib`.
11. J. Merill. GENERIC and GIMPLE: a new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, pages 171–180, 2003. `http://www.gccsummit.org/2003`.
12. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufmann, 1997.
13. D. Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. `http://www.gccsummit.org/2004`.
14. D. Novillo. Tree SSA - a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers Summit*, pages 181–193, 2003. `http://www.gccsummit.org/2003`.
15. S. Pop, P. Clauss, A. Cohen, V. Loechner, and G.-A. Silber. Fast recognition of scalar evolutions on three-address ssa code. Technical Report A/354/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2004. `http://www.cri.ensmp.fr/classement/doc/A-354.ps`.
16. R. van Engelen, J. Birch, Y. Shou, B. Walsh, and K. Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pages 106–115, 2004.
17. M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.
18. D. Wonnacott and W. Pugh. Nonlinear array dependence analysis. In *Proc. Third Workshop on Languages, Compilers and Run-Time Systems for Scalable Computers*, 1995. Troy, New York.