

IST ACOTES Project
Deliverable D3.9
Loop Nest Optimization
Prototype

IST-034869

Public

History

Project Number	IST-034869
Project Title	Advanced Compiler Technologies for Embedded Streaming
Deliverable Type	Prototype

Deliverable Number	D3.9
Title of Deliverable	Loop Nest Optimization Prototype
Nature of Deliverable	Public
Internal Document Number/version	acotes-d3.9-final
Contractual Delivery Date	31 May 2008
Actual Delivery Date	25 August 2008
WP(s)	3
Author(s)/Affiliation	Albert Cohen, Louis-Noel Pouchet, Konrad Trifunovic (INRIA)

Abstract

This deliverable consists of a set of libraries for loop nest optimization and automatic parallelization, and a new compilation pass of the GNU Compiler Collection (GCC) called Graphite to integrate those libraries into the ACOTES tool flow. These libraries support locality optimization, vectorization-enhancing transformations, and instruction-level parallelism extraction. Experimental validation and publications are described in the document. The Graphite pass is a pioneering attempt at bringing polyhedral compilation techniques to production compilers.

Keyword list

Loop nest optimization, automatic parallelization, task-level optimization, static control loop nests, polyhedral compilation in a production compiler.

Contents

1	Introduction	3
2	Graphite Compilation Pass	3
3	Example	3
4	Graphite Analyses, Transformations and Heuristics	6
4.1	Features	6
4.2	Transformations	6
4.3	Heuristics	6

1 Introduction

The goal of the ACOTES project is to substantially improve the programmer’s productivity in developing applications that process massive streams of data on programmable, parallel embedded architectures. The target application areas of ACOTES are video processing and advanced radio communications for consumer systems. Work Package (WP) 2 of ACOTES defined a Streaming Programming Model (SPM) and an Abstract Streaming Machine (ASM). The ASM itself consists of two parts: the first part deals with the inter-core/tile/cluster characteristics of the target platform, including coarse-grained Data Level Parallelism (DLP) features, and serves as an input to the high-level analyzes of WP2 and optimizations of WP3; the second part deals with the capabilities of the target platform at the level of each individual core/tile/cluster (including fine-gained DLP), and serves as an input primarily to the low-level optimizations and primarily auto-vectorization (WP4).

This deliverable presents the first prototype of a loop nest optimization framework, aiming at offering performance portability of single-threaded loop nests. This framework will also allow to connect task-level optimizations described in Deliverable D3.4 to the middle-end of GCC. Interactions between these loop nest optimizations and finer grain optimizations have been discussed in Deliverable D4.2. Therefore, the present deliverable focuses on task-level optimizations performed at compilation time.

2 Graphite Compilation Pass

The Graphite project has been initiated in 2006 and presented at the GCC Summit [2]; the motivations, design and development plans have been discussed there as well as in previous experiments in the context of the Open64 compiler [1].

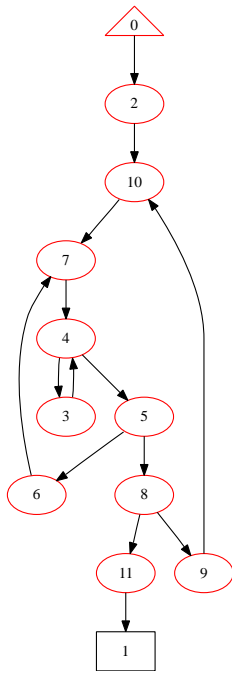
It can be downloaded via SVN:

```
svn co svn://gcc.gnu.org/svn/branches/graphite gcc
```

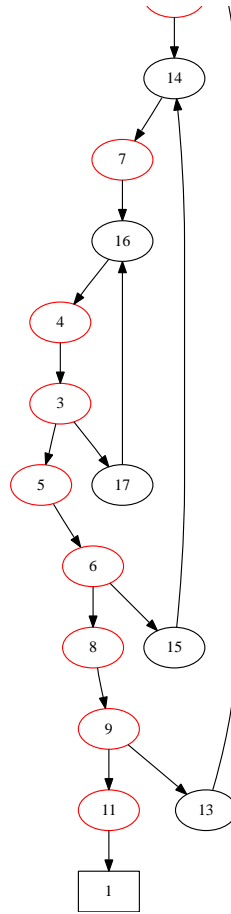
In the current state of the Graphite project, static control loop nests are automatically recognized and mapped to a polyhedral representation. Then, any affine transformation may occur, before calling the code generation algorithm and regenerating GIMPLE trees and control-flow graphs to proceed with the normal compilation flow. All transformations of the iteration domains and schedules are supported, but it is not yet the case for array expansion/contraction and affine data layout optimizations (these are left for future work and may involve a collaboration with IBM Haifa).

3 Example

As an illustration, the figures below show the control-flow before and after the Graphite pass, on a matrix-matrix multiplication code. No actual schedule transformation was performed; the loop nesting is preserved, as this example just illustrates the generic conversion of loop nests in and out of the polyhedral representation.



Control-flow before SCoP formation.



Control-flow after polyhedral code generation.

The corresponding GIMPLE intermediate representation is provided in the following listings.

```

matmult (float[1000] * A)
{
    float pretmp.13__I_lsm.25, prehitmp.18, pretmp.17;
    float[1000] * pretmp.16, * pretmp.13;
    long unsigned int pretmp.15, pretmp.12;
    unsigned int k, j, i;
    float C[1000][1000], B[1000][1000];
    float D.1588, D.1587, D.1586, D.1585, D.1584;

<bb 2>:
    goto <bb 10>;

<bb 3>:

<bb 4>:
    # SMT.24_17 = PHI <SMT.24_17(3), SMT.24_2(7)>
    # prehitmp.18_41 = PHI <D.1587_24(3), pretmp.17_44(7)>
    # k_47 = PHI <k_25(3), 0(7)>
    # VUSE <B_34(D)>
    D.1584_19 = B[1_45][k_47];
    # VUSE <C_35(D)>
    D.1585_22 = C[k_47][j_46];
    D.1586_23 = D.1584_19 * D.1585_22;
    D.1587_24 = D.1586_23 + prehitmp.18_41;
    pretmp.13__I_lsm.25_37 = D.1587_24;
    k_25 = k_47 + 1;
    if (k_25 <= 999)
        goto <bb 3>;
    else
        goto <bb 5>;

<bb 5>:
    # pretmp.13__I_lsm.25_12 = PHI <pretmp.13__I_lsm.25_37(4)>
    # SMT.24_48 = VDEF <SMT.24_17>
    (*pretmp.13_42)[j_46] = pretmp.13__I_lsm.25_12;
    j_26 = j_46 + 1;
    if (j_26 <= 999)
        goto <bb 6>;
    else
        goto <bb 8>;

<bb 6>:

<bb 7>:
    # SMT.24_2 = PHI <SMT.24_1(10), SMT.24_48(6)>
    # j_46 = PHI <0(10), j_26(6)>
    # VUSE <SMT.24_2>
    pretmp.17_44 = (*pretmp.16_32)[j_46];
    # VUSE <SMT.24_2>
    pretmp.13__I_lsm.25_15 = (*pretmp.13_42)[j_46];
    goto <bb 4>;

<bb 8>:
    i_27 = i_45 + 1;
    if (i_27 <= 999)
        goto <bb 9>;
    else
        goto <bb 11>;

<bb 9>:

<bb 10>:
    # SMT.24_1 = PHI <SMT.24_40(D)(2), SMT.24_48(9)>
    # i_45 = PHI <0(2), i_27(9)>
    pretmp.12_30 = (long unsigned int) i_45;
    pretmp.12_39 = pretmp.12_30 * 4000;
    pretmp.13_42 = A_9(D) + pretmp.12_39;
    pretmp.15_31 = (long unsigned int) i_45;
    pretmp.15_3 = pretmp.15_31 * 4000;
    pretmp.16_32 = A_9(D) + pretmp.15_3;
    goto <bb 7>;

<bb 11>:
    // I/O and return code...
}

```

Control-flow before SCoP formation.

```

matmult (float[1000] * A)
{
    _Bool D.1619, D.1617, D.1615;
    unsigned int graphiteIV.28, graphiteIV.27, graphiteIV.26;
    float pretmp.13__I_lsm.25, prehitmp.18, pretmp.17;
    float[1000] * pretmp.16, * pretmp.13;
    long unsigned int pretmp.15, pretmp.12;
    unsigned int k, j, i;
    float C[1000][1000], B[1000][1000];
    float D.1588, D.1587, D.1586, D.1585, D.1584;

<bb 2>:

<bb 12>:
    # SMT.24_1 = PHI <SMT.24_40(D)(2), SMT.24_48(13)>
    # graphiteIV.26_13 = PHI <0(2), graphiteIV.26_14(13)>
    D.1615_16 = 999 < graphiteIV.26_13;
    goto <bb 10>;

<bb 14>:
    # SMT.24_2 = PHI <SMT.24_1(10), SMT.24_48(15)>
    # graphiteIV.27_43 = PHI <0(10), graphiteIV.27_49(15)>
    D.1617_50 = 999 < graphiteIV.27_43;
    goto <bb 7>;

<bb 16>:
    # prehitmp.18_41 = PHI <D.1587_24(7), pretmp.17_44(17)>
    # SMT.24_17 = PHI <SMT.24_17(7), SMT.24_2(17)>
    # graphiteIV.28_51 = PHI <0(7), graphiteIV.28_52(17)>
    D.1619_53 = 999 < graphiteIV.28_51;
    goto <bb 4>;

<bb 17>:
    graphiteIV.28_52 = graphiteIV.28_51 + 1;
    goto <bb 16>;

<bb 15>:
    graphiteIV.27_49 = graphiteIV.27_43 + 1;
    goto <bb 14>;

<bb 13>:
    graphiteIV.26_14 = graphiteIV.26_13 + 1;
    goto <bb 12>;

<bb 3>:
    if (D.1619_53)
        goto <bb 5>;
    else
        goto <bb 17>;

<bb 4>:
    # VUSE <B_34(D)>
    D.1584_19 = B[graphiteIV.26_13][graphiteIV.28_51];
    # VUSE <C_35(D)>
    D.1585_22 = C[graphiteIV.28_51][graphiteIV.27_43];
    D.1586_23 = D.1584_19 * D.1585_22;
    D.1587_24 = D.1586_23 + prehitmp.18_41;
    pretmp.13__I_lsm.25_37 = D.1587_24;
    k_25 = graphiteIV.28_51 + 1;
    goto <bb 3>;

<bb 5>:
    # SMT.24_48 = VDEF <SMT.24_17>
    (*pretmp.13_42)[graphiteIV.27_43] = pretmp.13__I_lsm.25_12;
    j_26 = graphiteIV.27_43 + 1;

<bb 6>:
    if (D.1617_50)
        goto <bb 8>;
    else
        goto <bb 15>;

<bb 7>:
    # VUSE <SMT.24_2>
    pretmp.17_44 = (*pretmp.16_32)[graphiteIV.27_43];
    # VUSE <SMT.24_2>
    pretmp.13__I_lsm.25_15 = (*pretmp.13_42)[graphiteIV.27_43];
    goto <bb 16>;

<bb 8>:
    i_27 = graphiteIV.26_13 + 1;

<bb 9>:
    if (D.1615_16)
        goto <bb 11>;
    else
        goto <bb 13>;

<bb 10>:
    pretmp.12_30 = (long unsigned int) graphiteIV.26_13;
    pretmp.12_39 = pretmp.12_30 * 4000;
    pretmp.13_42 = A_9(D) + pretmp.12_39;
    pretmp.15_31 = (long unsigned int) graphiteIV.26_13;
    pretmp.15_3 = pretmp.15_31 * 4000;
    pretmp.16_32 = A_9(D) + pretmp.15_3;
    goto <bb 14>;

<bb 11>:
    // I/O and return code...
}

```

Control-flow after polyhedral code generation.

4 Graphite Analyses, Transformations and Heuristics

Graphite is designed as a generic pass to integrate polyhedral compilation algorithms into GCC. To seamlessly plug in such algorithms, Graphite provides a set of dependence analyses, a standard representation of the program for transformation purposes, and way to interact with existing and future cost-modeling frameworks in GCC.

4.1 Features

Graphite also provides an exact and approximate instancewise data dependence analysis. It works on a full SCoP, collecting pairs of iterations of statements in dependence. It is being further enhanced to compute instancewise reaching definition (last-write) relations, allowing for more expressive optimization search spaces via array privatization and renaming.

4.2 Transformations

Graphite currently performs very few loop transformations by itself: namely, loop interchange and loop tiling (activated by additional compiler options). Achieving the full potential of Graphite will require further integration with the model-based or feedback-directed tools. For this purpose, we will connect Graphite to the LetSee exploration tool (see D3.4) in the third year of the project, providing GCC with the most advanced model-based and feedback-directed loop nest transformations.

LetSee can be downloaded at the following URL:

<http://www-rocq.inria.fr/~pouchet/software/letsee>

4.3 Heuristics

LetSee and Graphite use the same internal program representation; we do not expect any integration difficulty. The same should apply to other PolyLib-based compilation libraries designed in the project or by third-party developers.

The main research challenge remains to drive feedback-directed optimization tools like LetSee with a relevant and optimization model, compatible with the constraints (execution time, number of runs) of a production compiler. The modular cost modeling approach described in the upcoming deliverable D4.3 is a promising step in the right direction; it will provide key information to narrow the optimization search space when the integration of LetSee to Graphite will be effective.

References

- [1] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parelo, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3):261–317, June 2006. Special issue on Microgrids.
- [2] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. Graphite: Loop optimizations based on the polyhedral model for GCC. In *Proc. of the 4th GCC Developer's Summit*, Ottawa, Canada, June 2006.