Optimization opportunities based on the polyhedral model in GRAPHITE

How much impact has GRAPHITE already?

Tobias Grosser
University of Passau
grosser@fim.uni-passau.de

Abstract

The polytope model is used since many years to describe standard loop optimizations like blocking, interchange or fusion, but also advanced memory access optimizations and automatic parallelization. Its exact mathematical description of memory accesses and loop iterations allows to concentrate on the optimization problem and to take advantage of professional problem solving tools developed for operational research.
Up to today the polytope model was limited to research compilers or source to source transformations. Graphite generates a polytope description of all programs compiled by the gcc. Therefore polytope optimization techniques are not limited anymore to hand selected code pieces, but can actually be applied in large scale on real world programs. By showing the impact of GRAPHITE on important benchmarks - “How much runtime is actually spent in code, that can be optimized by polytope optimization techniques?" - we invite people to base their current polytope research on GRAPHITE to make these optimizations available to the large set of gcc compiled applications.

1 Motivation

The polytope model describes memory access optimizations based on an abstract mathematical representation. It can be used to describe traditional loop optimizations like blocking, tiling, or splitting in an exact way and to schedule them in arbitrary order. However there are also advanced auto parallelization[1] passes and new optimizations based on powerful operational research tools, that cannot be expressed easily with traditional loop transformations, possible.

[1] http://www.fmi.uni-passau.de/cl/loopo/

As the polytope model was always limited to source to source translation tools or hand selected code pieces, it was never used on regular base to optimize programs written in imperative languages. Therefore the big portion of C, C++ and Fortran programs was never accessible to optimizations based on the polytope model. GRAPHITE[2] is the first open source implementation of the polytope model for a low level imperative compiler. GCC 4.4 already includes a first implementation of GRAPHITE, which is still limited in several aspects. It is tightly connected to gcc and the polytope description is not yet complete. During the last six months the current graphite branch was reworked and a complete polytope description was introduced. The last but very important thing missing is a set of advanced optimizations based on GRAPHITE and the polytope model. Here we can take advantage of the research taking part in this area since a long time.
Before starting to write/port new optimizations it is time to take a step back and see what can be optimized by GRAPHITE and if it is actually worth to write optimizations for.
How much code can be translated into the polytope model? How many conditions, statements, and loops of the original program are covered by the polytope description? How much runtime is actually spent in code that can be described and therefore optimized? And finally: If some code is not handled at the moment, is this just not yet implemented in GRAPHITE, is it a limitation because of missing optimizations in GCC or a limitation of the polytope model?
To find an answer, valid for a large set of programs, the SPEC 2006 benchmark suite is used to analyze the impact of GRAPHITE on “real world" programs.

[2] http://gcc.gnu.org/wiki/Graphite

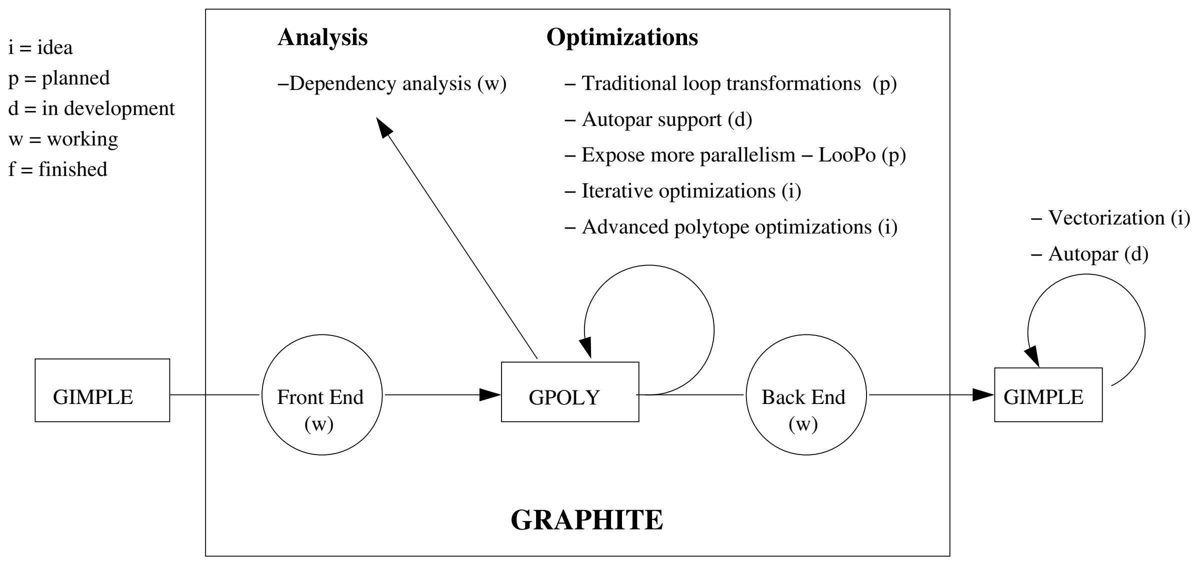

Figure 1: Status of GRAPHITE development - May 2009

2 Status of GRAPHITE development

GRAPHITE is developed since several years. The first analysis, the scalar evolution pass, on which it is heavily based was committed five years ago. Last year the first public visible code was committed to gcc 4.4, released in spring 2009. However until today the impressive gains are still missing. So what is the development status? What is still coming and what does already exits?

To start there are two different versions of GRAPHITE. The version as in gcc 4.4 allows a simple transformation - loop blocking - to give a first impression of GRAPHITE and to test integration of CLooG and PPL. However GRAPHITE 4.4 is still very limited. Loop blocking does not have any heuristics at all and is based on a non polytope dependency analyzes from the lambda framework. This limits the effects and prevents big performance improvements.

The second version is in GRAPHITE branch and already took the next step. During the last 6 month the polytope description was completed, so that all future optimizations can take advantage of it and are not limited by any leftover code. Also all non polytope code was removed. This means in GRAPHITE branch there are currently no optimizations at all. However it is ready to base optimizations on it.

This paper will describe the situation in GRAPHITE branch, as this is the first full polytope model in gcc.

In branch front end and back end, converting from GIMPLE to the GRAPHITE polytope description (GPOLY) and back from GPOLY to GIMPLE, work. This means we can extract the polytope description for interesting code regions and generate completely new loop nests from this description. This was tested by enabling the identity transformation "*GIMPLE* $\rightarrow$ *GPOLY* $\rightarrow$ *GIMPLE*" in gcc bootstrap and by testing large projects as the SPEC 2006 benchmark suite. Fortunately the missing part is the most interesting. Adding optimizations that work on the polytope representation.

There are two student projects about optimizations in GRAPHITE during Google Summer of Code™ 2009. The first one works on the already existing tree-autopar code for automatic loop parallelization with OpenMP. This code will be extended to handle all code graphite can optimize.

The other project aims to implement traditional loop transformations in GRAPHITE. Even if we already have loop-blocking in GCC 4.4, to actually take advantage of the polytope model and to get real performance improvements we need loop transformations that work completely on the polytope model and that are are accompanied by well tuned heuristics.

The last missing part blocking both projects, the polyhedral data dependency analysis, was recently finished. Beside these two projects there are several other open projects, whereas most of them can take advantage of research already done. The LooPo [5] project has a large set of tools to discover more parallelism, there is research on iterative compilation or the internal vectorizer

might want to take advantage of the exact dependency analysis. It is also possible to write new optimizations that use characteristics of the polytope model to optimize cost functions like the distance between memory access using the linear programming solver in PPL.
Another orthogonal project in GRAPHITE is PCP[3]. It was started half a year ago and will move GRAPHITE in a simple to handle package that includes a clean text and library interface for improved debugging and testing facilities. This should make development of GRAPHITE optimization passes even easier. For further information about the design of GRAPHITE and PCP is in the paper [7] available.

3 The polytope model

The polytope model is the model on which future GRAPHITE based optimizations will work. As there exists already a long research history there are several papers that talk about the polytope model [6], however most of them base the description on an source to source compiler.
In the context of modern compiler development this paper tries to offer a different introduction to the polytope model.

3.1 SSA on arrays - The polytope model?

Most modern optimizing compilers use an intermediate language based on static single assignment (SSA[4]) in their analysis and optimization passes. GIMPLE, the intermediate language used in GCC, is also based on SSA. The main reason for the use of SSA are the explicit use-def chains. Using SSA the lookup of the defining statement for a given use is in $O(1)$. Furthermore for every use there is only one defining statement, therefore algorithms on SSA are often a lot simpler than on non SSA code.
Unfortunately SSA is only well defined on scalar variables (scalar SSA), whereas it is difficult to define a pratical version of SSA on non scalar variables like arrays or memory references (array SSA). Even though GCC defines some kind of array SSA using "VUSE" statements, the implementation is limited compared to scalar SSA. In scalar SSA for every use of a scalar its current value depends on exactly one defining statement, as every write defines a scalar completely. In the array SSA implementation of gcc the content of an array can be defined by several statements, as a write does not define the complete array but can define single array cells. So the array may contain cells defined by different statements. Therefore there is a set of defining statements, not just one defining statement.
Making array SSA in the general case as useful as scalar SSA is a difficult problem, maybe impossible. However it is possible to represent a subset of programs using the polytope model, which can be seen as some kind of array SSA that extends the current "VUSE" statements.

3.1.1 Arrays as sets of scalar values

Even if it is difficult to define SSA on arrays, it is possible to extend scalar SSA. As arrays are defined as sets of scalar values, instead of tracking accesses to the complete array, accesses to every single array cell can be tracked. And for scalar values, SSA is already defined and useful. Unfortunately in the general case the array cells a statement accesses are not known at compile time, as the array subscript can be calculated by a function - in the worst case by a call to `rand()`. So tracking them is not always possible.
Nevertheless there is a subset of array accesses that can be tracked exactly. All array accesses with compile time constant subscripts like $A[5][8]$ or $B[3][2]$. In this subset memory accesses to scalars and arrays can be defined in the same way. A memory access is the access to a scalar cell in the memory M of the program, a vector space in $\mathbb{Z}^n$, whereas the first dimension defines the base element and further dimension define possible array subscripts. Scalar values like c are handled as arrays of dimension zero. To sort variables in this model we define a function f that assigns every scalar name and every array base an unique number $i \in \mathbb{Z}$.
Now a memory access is not defined by its name, but as access to a cell $c \in M$. With f defined as $\{A \rightarrow 0, B \rightarrow 1, c \rightarrow 2\}$, $A[5][8]$ is the access to (0,5,8), $B[3][2]$ is the access to (1,3,2), and c is (2, 0, 0) as in 3.
By accessing array cells with this notation they are handled like normal scalars. So every use has exactly one defining statement, as every write defines a cell completely. However this is just the first part of SSA. Finding the defining statement is still a problem, as accesses to array cells are not yet in SSA. There may be several statements that write into (0,5,8).

[3]http://gcc.gnu.org/wiki/PCP

```
A [5][8] = 10;
A [5][8] = c;

for (i = 0; i < 100; i++) {
  B[3][2] = A[5][8]
   for (j = 0; j < 50; j++)
     A[3][4] = B[3][2]
  B[5][8] = A[3][4]
}
```

Figure 2: Non SSA code

```
(0,5,8) = 10;
(0,5,8) = (2,0,0);

for (i = 0; i < 100; i++) {
   (1,3,2) = (0,5,8);
   for (j = 0; j < 50; j++)
     (0,3,4) = (1,3,2);
   (1,5,8) = (0,3,4);
}
```

Figure 3: Constant array accesses based on memory cells

3.2 Using a schedule to specify the defining statement

To identify a certain definition, the memory cell it accesses is not enough. Scalar SSA solves this by allowing only one definiton of every scalar variable. Further writes into the same variable are forbidden. By adding this restriction the name of a scalar variable is enough to identify the defining statement.
However for the scalar notation that was introduced on arrays a different approach is taken. Statements are referenced using a schedule. Every statement gets assigned an vector depending on its textual position in the program. The first statement is S:1, the second statement is S:2, ... For every loop level an additional dimension is added that is incremented instead of the outer dimensions. If a new loop is for example started at position 3 the first statement in this loop has the schedule S:3.1. To specify the last definition of a use a tuple of schedule and memory cell is used. For example to reference the definiton of memory cell (0,5,8) that took place in S:2, the tuple (2, (0,5,8)) is used as seen in 4.
Now every use can reference the single definition it is based on. This references can be used like nor-

```
(0,5,8) = 10; //S1
(0,5,8) = (0,(2,0,0)); //S2

for (i = 0; i < 100; i++) {
  (1,3,2) = (2,(0,5,8)); //S3.1
  for (j = 0; j < 50; j++)
    (0,3,4) = (3.1,(1,3,2)); //S3.2.1
  (1,5,8) = (3.2.1,(0,3,4)); //S3.3
}
```

Figure 4: Constant array SSA using the statement schedule

mal SSA, but instead of a single assignment for every name there is a single assignment for every tuple $(schedule, memorycell)$.

3.3 Extending array SSA for loops

The subset we used to define SSA on arrays is still very limited and is not able to represent a lot of real world code. Especially in loops it is uncommon to find constant array subscripts, as loops are often used to rework the content of a complete array. Fortunately it is possible to extend the defined array SSA.
To be able to represent arrays as set of scalars it was necessary to specify for every array access the accessed cell. This is easy for constant subscripts, but in general it is possible for all access functions that can be expressed and analyzed at compile time. Therefore array accesses like $A[(i*M)^2]$ can be analyzed, as long as there is a way to store and analyze the function $(i*M)^2$ reasonable fast.
Unfortunately analyzing arbitrary functions is complex. However there is a subset of functions that can be analyzed with limited complexity. The set of functions, that are affine in virtual loop iterators (VLI) and parameters (P). Parameters are defined as scalar integer variables, that are constant during the execution of a code region expressed using polyhedral array SSA. Virtual loop iterators count the number of loop iterations. Therefore valid array accesses are all accesses, where the subscripts accessed are defined by an affine function like $5i+8j+k+12$ as with $i, j \in VLI$ and $k \in P$ as shown in 5.
However by using this extension specifying the defining statement for an use is more complex. Scalar SSA and constant array SSA have the property, that in every loop the defining statement iteration for a memory

cell is always the previous iteration, as in every iteration the same memory cell is overwritten. If affine access functions are used, a statement can write in a different array cell on every loop iteration. Therefore a definition can only be referenced by the combination of the static schedule referencing a statement and a dynamic schedule specifying the loop iteration. In 5 e.g. the last definition of memory cell (1,25+k) was in statement S3.1 at iteration (i=5).

Therefore use-def chains are defined in between single statement iterations, instead of statements. As the number of statement iterations is prohibitive high, it is impossible to save all use-def relations for every single iteration. However it is possible to calulate these use-def chains as long as the expressions in loop boundaries and conditions are affine functions in VLI and P. The usage of polytope libraries enables us to express the detailed use-def information in a highly compressed form. For S4.1 the affine function $j = 5i + k$ is used to identify the defs for every iteration of loop j. For iteration $j = 5 + k$ the corresponding def is for example $i = 1$.

```
void foo (int k)
  A [5] = 10;
  A [k] = c;

   for (i = 0; i < 100; i++)
     B[5i+k] = A[5]

   for (j = 0; j < 100; j++)
     ... = B[j]

void foo (int k)
   (0,5) = 10; // S:1
   (0,k) = (0,(2,0)); // S:2

   for (i = 0; i < 100; i++){
     (1,5i+k) = (2,(0,5); // S:3.1

   for (j = 0; j < 100; j++)
     ... = (3.1 (j=5i+k),j)); // S:4.1
```

Figure 5: Affine array SSA

3.3.1 Why generate a new model and not extend GIMPLE?

It was shown that it is possible to extend SSA to arrays, so why is GRAPHITE not implemented as GIMPLE extension? One reason is that polytope array SSA duplicates a lot of information available in GIMPLE. Access functions, loop boundaries and conditional expressions are saved in the polytopes. Therefore all scalar code, except some reductions, is duplicated. Any transformation on the polytopes or the gimple code would require an expensive update of the other data structure.

Also the polytope model is not expressive enough to be used for all GIMPLE memory accesses, so there would be two data structures to represent memory accesses. This complicates writing general algorithms.

However the most important point is, that the polytope model is not imperative any more. Or at least it does not have to be imperative. A complete dependency description of the different statements and their iterations is enough to calculate the result of a SCoP. There is no need for control flow any more. So working on control flow based data structures like GIMPLE is too low level.

3.3.2 How to optimize polytope array SSA

There are two different ways to optimize in the polytope model. This first one is to change the order in which statements and loop iterations are executed. Most traditional loop optimizations try to optimize the use of the CPU cache, by either creating more cache locality using e.g. loop blocking or loop fusion or by reducing the memory footprint using loop splitting. All these transformations can be expressed by changing the execution order. However there are other use case to change the execution oder. The two most interesting ones are transformations enabling parallel execution or vectorization.

Another ortogonal approach is optimizing the data layout. It is possible to reorder arrays to improve cache locality or to privatize data to allow further optimizations. It might even be possible to reduce the memory footprint by removing unused array regions or by removing temporary results.

3.4 Code GRAPHITE can represent

GRAPHITE calls a part of a program that can be represented in the polytope model a static control part

```
/* Start SCoP.  */
start:
  A[i] = 1;
  i++;

  if (i <= 100)
    goto start;
/* End SCoP.  */
```

Figure 6: Valid SCoP with goto-loop

(SCoP). SCoPs are detected on the gcc intermediate representation GIMPLE, therefore all properties that qualify a code region as SCoP are independent of a specific language representation. Hence GRAPHITE is able to optimize programs written in any of the gcc front end languages (C, C++, Fortran, Ada, Java).

A SCoPs is a single entry single exit region containing an arbitrary set of loop nests, straight line code and/or conditions, whereas the control flow has to be structured. However it is possible that the code in a SCoP is written using explicit `for` loops or implicit `while`, `do..while` or even `goto` loops. Loops that contain multiple exits are not supported.

Scalar variables that do not changed inside a SCoP are called parameters. Together with the loop induction variables the parameters span a vector space that is used to define affine expressions, whereas loops with multiple induction variables are supported.

In a SCoP only loops are allowed that have constant strides and which bounds can be expressed by affine expressions. Therefore unlimited infinite loops are not allowed. Conditions are limited to comparisons (`<`, `<=`, `>`, `>=`, `==`, `!=`) between two affine expressions. However the same generality applies as for the control flow. Affine expressions do not have to be written explicitly in the source code, but it is sufficient that the expressions used can be analyzed to an affine expressions.

GRAPHITE handles all side effect free calculations in a SCoP. Thus even function calls are allowed if they are `pure` or `const`.

To be able to analyze dependencies efficiently data references to arrays are only allowed, if the subscripts are affine expressions, whereas graphite depends a lot on working alias analysis to be able to detect independent pointer sets. A SCoP can also contain scalar reductions as payload, they are handled like all data references. Structures can not be part of a SCoP.

```
void foo (int M, int N) {
  int i;

  /* Start SCoP.  */
  for (i = 0; i <= M, i+=2)
    a = 10 * i + 12 + M;
    b = 5 * i + a;

    if (5 * i + 10 * M != a)
      A[b] = 20;
    else
      A[b] = b;
  /* End SCoP.  */
}
```

Figure 7: Valid affine expressions in SCoP

SCoPs that meet these properties can be represented by the polytope model and can be optimized using generic polytope optimizations.

4 The polytope description in GRAPHITE

GRAPHITE implements a plain and simple interface[4] to optimize memory accesses based on the polytope model. As this model has been used for many years there exists already a well established description heavily influenced by CLooG[2], an open source polytope code generator.
The interface in GRAPHITE is close to the established descriptions used in polytope research, but contains some small differences and adjustments. Therefore it should be easy to port and try existing optimizations.
As GRAPHITE's polytope interface establishes a strict boundary to the GCC internal state. Analysis and optimization passes read, analyze and optimize an abstract mathematical problem description.
The small and independent interface allows people that are not yet part of the gcc community to try their polytope optimizations, without being forced to get used to gcc internals. A small header file with three data structures is enough to port a polytope optimization to the

[4]Source in "gcc/graphite-poly.h"

```
void foo (int N, int M) {
  int i, j, r;
  int A[100];

  // Start SCoP
  A[N+1] = 0; // bb0

  for (i = 0; i <= N+10; i++) {
    A [i+N] = A [i]; // bb1

    for (j = i; j <= i+M; j++)
      if (j > N)
        r = A [i-N] + r; // bb2
  }
  // End SCoP
}
```

Figure 8: Example SCoP

GCC.
And if polytope optimizations once prove to be useful in compiler development, the simple interface is portable to a large set of open source compilers, so that a productive exchange and comparison of optimizations can take place.

4.1 The polytope library in GRAPHITE

To describe polytopes the Parma Polyhedra Library (PPL [1]) is used. It is the leading open source polytope library. For the import of GRAPHITE into gcc 4.4 PPL's platform support was tested on and extended to all important gcc platforms.
All polytopes in GRAPHITE use the type `ppl_Pointset_Powerset_NNC_Polyhedron_t`. This type describes a union of non necessary closed convex polyhedra in the vector space $\mathbb{R}^n$. This is wrong as loop iterations and array accesses are elements of $\mathbb{Z}$, but works for the current uses. However to write more sophisticated optimizations, that require counting the points in the polyhedra, polyhedra in the vector space $\mathbb{Z}^n$ are required. So a switch to $\mathbb{Z}$-polyhedra will be necessary.

4.2 SCoP

A SCoP is a program region described in the polytope model. Therefore it only contains information concerning the polytope model. Additional information describing the actual calculations is hidden. All optimizations and analysis take a SCoP, work on it, and return an optimized or analyzed version of it.
The SCoP $S = (p, bbs)$ is defined by the number of external parameters p and a set of black boxes bbs.
A parameter is an integer variable that is unknow during compile time, but constant during SCoP execution. In the example M and N are parameters, as they are used in the SCoP, but not modified. i and r are no parameters as i is an induction variable and r is modified inside the SCoP. For this example $p = 2$.

4.3 Black Box

A black box describes a calculation that will be executed in the SCoP. As the name black box suggests the details of the calculation are hidden. In our example every statement is a black box on its own. Therefore a black box can contain a larger set of statements, some hidden control flow, or function calls. The only part exposed are the data references a black box contains.
The black box $B = (domain, drs, scattering)$ is defined by the iteration domain *domain*, a set of data references *drs* and the scattering polytope *scattering*.
Every black box might be executed several times, whereas the loop induction variables are different for every iteration. A specific iteration of a black box is therefore referenced by the values of the loop induction variables for this iteration, whereas the possible values for the induction variables are the points in *domain*. *domain* contains one dimension for every parameter of the SCoP and a special dimension representing constant offsets. In addition it contains one dimension for every loop iterator. E.g. $bb1$ has one loop dimension representing i, whereas $bb2$ has two dimensions for i and j and $bb0$ has no loop dimension at all. The domain for bb looks like this 4.3
In contrast to other polytope optimizations packages the domain does not define the order in which different iterations are executed. Therefore it is not necessary for any optimization to change the domain.
The order in which the different iterations of a black box are executed is defined in the *scattering* polyhedron. This polyhedron contains the same dimensions as *domain*, but is extended by several new dimensions $(t_1, t_2, ..., t_n)$ called scattering dimensions. The scattering polytope now maps every point of the domain to a point defined by the new scattering dimensions. For ex-

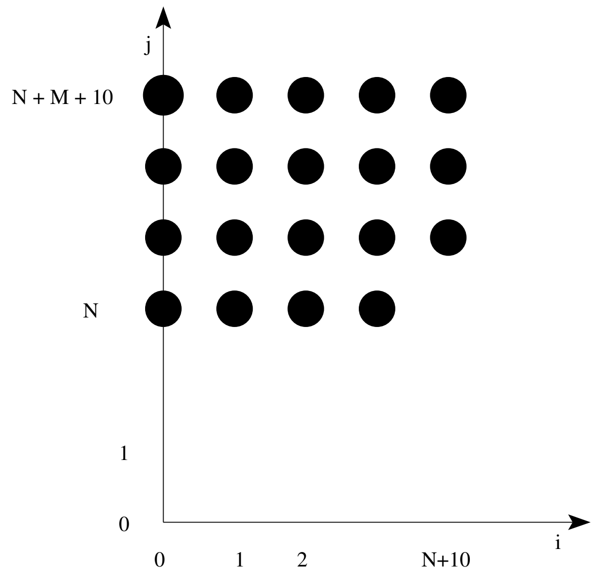

Figure 9: Domain for black box bb2

ample for $bb1$ the scattering function may define a mapping $\{t_1 = 0, t_2 = i, t_3 = 0\}$ and for $bb2$ we may define a mapping $\{t_1 = 2, t_2 = i, t_3 = j\}$. To get the global execution order of the iterations of all black boxes the scattering dimensions are ordered lexicographically and the corresponding black box iterations are executed in this order. Therefore all iterations of $bb1$ will be executed before $bb2$ as $t_1 < t_2$ for all iterations. The individual loop iterations are executed in the original order, as for iteration $i = 4$ and $i' = 5$ there is $t_2 < t'_2$.
Additional information about scattering functions can be found in the cloog documentation [5]. However in contrast to cloog the scattering functions in GPOLY are fully defined unions of non necessarily closed convex polyhedron. This allows to map subsets of the domain with different scattering functions.

4.4 Data references

A data reference $DR = (accesses, type)$, is defined by the accessed space *accesses* and the type of its access *type*. The type of data reference can either be read, write or may-write. Read means a data reference reads or may read any of the values marked in accesses. Write means a data reference must overwrite all the values marked in accesses. May-write means that the values marked in accesses can be but do not need to be overwritten.

[5] http://www.bastoul.net/cloog/manual.php#SEC8

Data references in GPOLY use an unified data model to describe the accessed memory. In classical GIMPLE there exist scalar values and arrays, whereas arrays are defined as a matrix of scalar cells. GPOLY on the other hand talks about a multi dimensional space $M \subseteq \mathbb{Z}^n$ of scalar values.
An access of an array cell $A[s_1]$ is mapped to the point $(d_1 = b, d_2 = s_1, d_3 = 0, ...)$, whereas the value b is defined by the alias set the array is pointing to. Every alias set is mapped to an unique value. If the array is part of more than one alias set every array cell is mapped to one point for every alias set the array is part of. The points only differ in the first dimension. Scalar values are handled like arrays of dimension 0.
In our example the read access of $bb2$ is defined as an access to $(2,0)$ for the scalar access `r`, and $(1, i - N)$ for the array access $A[i - N]$. This can represented like in 4.4 as union of two polyhedra each containing one element.
At the moment data references are read only for optimizations, therefore the optimization of the data layout is not yet possible.

5 Coverage of GRAPHITE

After having seen which regions GRAPHITE can optimize it is time to ask how many SCoPs can be found in “real world” software. Is there a significant impact of GRAPHITE on code coverage or are there still restrictions left that limit coverage. In this analysis the SPEC

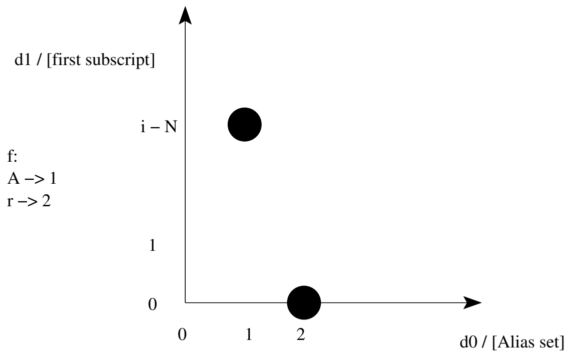

Figure 10: Access space for read data referance of bb2

CPU 2006[6] [7] benchmark suite was used to provide an accepted set of test programs. Nevertheless the reader is asked to try GRAPHITE on its own programs.

5.1 How was the code coverage analyzed?

5.1.1 Compile time coverage

The coverage analysis in this paper is based on the idea of the "gcov" tool. The number of interesting items in the complete program is counted and compared to the number of touched parts of the program. However in contrast to "gcov" the analyzis does not work on source code files, but on the gcc internal GIMPLE representation.
As the analyzis is based on GIMPLE, not the lines of code are counted, but loops, conditions and statements in the GIMPLE control flow tree of the program. This simplifies comparing different fronted languages. Also the GIMPLE language is closer to the GRAPHITE optimizations, so a coverage analysis on it seems to be more related to the "real world" impact as an analysis based on source code.
Every item (loop, condition, statement) is interesting as soon as it can be detected as a part of a SCoP during GRAPHITEs SCoP detection, because future optimizations in GRAPHITE will be able to optimize it.

5.1.2 Hot spot coverage

As normally programs spend a lot of time in small parts of their code base, to foresee the performance impact of GRAPHITE a measurement that can model these hotspots is needed.
The model that is used is an extension to the GIMPLE based compile time coverage. It takes advantage of gccs feedback analysis "`-fprofile-arcs -ftest-coverage -fprofile-use`". All benchmarks are run once with profiling instrumentation to get the execution count of every basic block. With the help of "`-fprofile-use`" this execution count is available for every basic block in gcc and was used to scale the loop, condition and statement counts depending on the number of times they where executed in the profiled test run.
These scaled counts will *not* give an approximation of the run time spent in a part of a program, as unoptimized hotspots can slow down program execution a lot. Nevertheless they give a better impression of the hot spot coverage and as they are independent of the processor arquitecture they can be used during GRAPHITE development to track progress in code coverage.
To get a accurate performance analysis for a specific test case on a specific hardware the usage of low impact performance measurement technics like PAPI might be usefule. However the SCoPs graphite detects are often very small so even little side effects might influence the measurement significantly.

5.2 Coverage of GRAPHITE branch - May 2009

At the moment (May 2009) GRAPHITE branch can handle most of the basic loop nests it was ment to handle, however it was never optimized for code coverage. Nevertheless it is important to get a start point to be able to see future improvements in code coverage.

[6] http://www.spec.org/cpu2006/

[7] Except '447.dealII', '465.tonto', '482.sphinx3', '433.milc' as they had still miscompiles in GRAPHITE.

	Static			Hotspot		
Benchmark	**loops**	**conds**	**stmts**	**loops**	**conds**	**stmts**
459: GemsFDTD	2.24	0.71	0.57	0.28	0.27	0.07
473: astar	3.31	0.79	0.47	0.00	0.00	0.00
410: bwaves	3.45	1.35	1.46	0.19	0.16	0.09
401: bzip2	3.37	1.20	0.90	0.53	0.18	0.23
436: cactusADM	**2.50**	**0.96**	**3.54**	**81.83**	**68.83**	**99.41**
454: calculix	**10.68**	**4.10**	**5.28**	**61.79**	**55.66**	**76.09**
403: gcc	2.87	0.40	0.48	0.51	0.25	0.48
445: gobmk	2.54	0.41	0.51	0.30	0.08	0.08
435: gromacs	9.08	3.49	2.77	7.75	2.94	2.20
464: h264ref	**6.24**	**2.05**	**2.21**	**20.42**	**12.58**	**7.96**
456: hmmer	0.79	0.26	0.22	0.00	0.00	0.00
470: lbm	**19.23**	**7.35**	**35.79**	**0.07**	**0.02**	**0.02**
437: leslie3d	5.76	2.49	1.23	4.36	4.34	0.72
429: mcf	1.54	0.42	0.41	0.00	0.00	0.00
444: namd	0.16	0.03	0.05	0.00	0.00	0.00
471: omnetpp	0.96	0.06	0.08	0.00	0.00	0.00
400: perlbench	2.51	0.25	0.70	1.01	0.16	0.79
453: povray	3.57	0.75	0.69	0.01	0.00	0.00
458: sjeng	2.60	0.29	0.27	2.40	0.67	0.69
450: soplex	0.58	0.12	0.10	0.61	0.32	0.38
481: wrf	**18.57**	**6.36**	**7.70**	**34.45**	**29.72**	**54.86**
483: xalancbmk	1.18	0.19	0.15	0.00	0.00	0.00
434: zeusmp	3.41	1.03	1.11	0.84	0.43	0.11
average	**4.66**	**1.52**	**2.90**	**9.45**	**7.68**	**10.62**

Figure 11: Coverage of GRAPHITE branch - May 2009 [in %]

Beside from this it is also good to see where are the low hanging fruits to improve the impact of GRAPHITE. As shown in 11 GRAPHITE achieves in average 4.66 % coverage of all loops[8]and about 2.90 of all GIMPLE statements. As expected the hot spot analysis shows that most of the loops are part of hotspots as 9.45 % of all loop iterations are covered by graphite and 10.62 % of all statement executions.
Very interesting is the high divergency between the different results. There exist only four interesting benchmarks looking at the hot spot coverage. These are cactusADM with 99.41 %, calculix with 76.09 %, and wrt with 54.86 % statement coverage and h264ref with 20.47 % loop coverage. All other benchmarks are under 5 % most of them even less.
Looking at the compile time coverage the diversity is still big, but not that extreme. All benchmarks are in in between 0.10 and 36.38 % statement coverage.
What is interesting is that a high value in compile time coverage does not imply high runtime coverage. Looking at cactusAMD it seems 3.64 % of all statements seem to be sufficient to get 99.41 % of the statement iterations. Whereas the 36.38 % statement coverage of lbm covers just 0.02 % of the executed statement iterations.
Taking a look at the first coverage report for GRAPHITE it shows two things. On the one hand we already got some very interesting test cases for which it is worth to write optimizations for. On the otherhand we learned that there are still a lot of benchmarks where GRAPHITE does not have much impact. However for a first shot the coverage seems promising enough to try to analyse the reasons for the small coverage in some of the benchmarks.

[8]loops, conds, stmts = simple coverage, whereas p-loops, p-conds, p-stmts is hot spot coverage

5.3 Ways to achieve better coverage

Until now focus in GRAPHITE development was to complete the polytope model by gathering necessary information from GIMPLE. As GPOLY is already complete enought to write optimizations on it, now it is time to focus on coverage. Good optimizations do not give any gains in performance, if they can not be applied.

5.3.1 What is possible in structured code ?

There are different ways to improve coverage of GRAPHITE. To get an impression how much coverage can be achived SCoP detection is run without any restrictions beside the structured control flow graph. Therefore it detects single entry single exit regions, without stopping on side effects, non affine loop bounds, structures, conditions that can not be handled or any other restrictions. They are only restricted to structured code containing only single exit loops and conditions with branches that can be joined easily. The analysis gives an idea of how much structured code can easily be accessed by GRAPHITE. As shown in 12 almost 50 % of the statements and 64.31 % of the statement iterations are in structured code.
However there is again a high diversity that shows e.g. statement coverage in between 11.25 % and 90.50%. Certainly it is theoretically impossible to handle all of this, but GRAPHITE can move closer to these numbers.

5.3.2 Implemening missing features

The first step to extend coverage is to look inside GRAPHITE. There are several interesting features that are not yet implemented and can be added without extending GPOLY.
The first feature would be to handle arbitrary boolean expressions of affine functions in loop boundaries and conditions. At the moment we just support code like “SCoP 1" in 13, whereas it is possible to add all conditions shown in “SCoP 2" in the polytope model. Fortunately most of the work to simplify the conditions will be done by PPL. For normal conditions the only part left to be done is to the conditions into PPL and to see how fast PPL can simply them. For loop bounderies it may be necessary to extend the analysis that detects the number of iterations for a loop.

```
void foo (int N, int M) {
  int i, r;
  int A[100];

  // Start SCoP 1
  for (i=0; i<=N+10; i++) {
      if (5 * i > i + N)
        r = A [i-N] + r;
  }
  // End SCoP 1

  // Start SCoP 2
  for (i=0; i<=N+10 || i<=M; i++) {
      if (5*i>i+N
           && (i!=12 && N < M))
        r = A [i-N] + r;
  }
  // End SCoP 2

}
```

Figure 13: Boolean expressions in bounds and conditions

It is also possible to support “may-write" for array accesses. Currently a memory access is rejected if the access function is not affine. However even it is not possible to represent the access function exactly, we can mark the complete region that they may be accessed as “may-write" or “may-read".
Another way to extend coverage is to allow unstructured code. At the moment conditions are not allowed to be nested in complicated ways, loops with multiple exits can not be handled. A SCoP detection working on unstructured code should be able to handle this. Even if we can not represent unstructured parts of a SCoP in the polytope model, it might be possible to hide these parts inside a black box and treat them like an atomic operation. While working on more complex control flow graphs it would also possible to support `switch` statements.

5.3.3 Improve gcc analysis passes

GRAPHITE relies heavily on several gcc analysis passes. The most important ones are loop detection, scalar evolution and alias analysis.

	Static			Hotspot		
Benchmark	**loops**	**conds**	**stmts**	**loops**	**conds**	**stmts**
459: GemsFDTD	96.55	60.80	68.24	100.00	99.98	100.00
473: astar	72.73	52.95	62.04	88.55	54.14	62.52
410: bwaves	97.70	82.88	81.07	100.00	100.00	100.00
401: bzip2	53.37	37.75	43.26	54.82	27.05	30.74
436: cactusADM	72.81	48.08	62.92	99.33	98.48	99.96
454: calculix	77.31	52.97	59.11	97.34	92.89	97.67
403: gcc	43.29	15.63	19.66	63.17	39.57	46.28
445: gobmk	43.09	23.76	26.07	53.78	31.87	36.52
435: gromacs	83.91	61.00	73.57	57.96	36.38	83.44
464: h264ref	87.01	52.59	56.61	94.54	82.30	89.50
456: hmmer	66.40	42.89	53.68	96.97	98.81	98.89
470: lbm	65.38	54.41	90.50	99.93	99.91	100.00
437: leslie3d	96.64	56.09	74.40	99.93	99.79	99.98
429: mcf	27.69	18.99	19.44	14.76	12.53	10.69
444: namd	53.88	24.16	41.04	20.23	12.34	54.91
471: omnetpp	38.70	5.47	11.25	3.78	1.96	2.03
400: perlbench	40.26	13.17	16.49	4.57	11.81	18.33
453: povray	64.42	36.39	39.78	17.56	11.86	17.33
458: sjeng	61.69	21.91	23.60	58.75	36.39	35.44
450: soplex	61.57	37.53	44.37	86.91	78.86	78.96
481: wrf	92.05	67.45	63.42	99.00	97.53	98.43
483: xalancbmk	48.88	17.39	24.80	37.23	14.64	17.67
434: zeusmp	79.21	60.82	68.74	99.52	99.00	99.77
average	**66.28**	**41.09**	**48.87**	**67.33**	**58.18**	**64.31**

Figure 12: Coverage on structured code without restrictions [in %]

As we have already seen in 12 loop detection does a decent job.
However looking into the scalar evolution analysis there might still be room for improvement. Running SCoP detection without stopping for scalar evolutions that returned "scev-unknown", shows that the executed statements increase from 10.62% to 22.63 % as well as the other metrics. Therefore improving scalar evolution could have significant impact on GRAPHITEs code coverage. Another interesting pass is the alias and data reference analysis. This pass is not only interesting to achieve hight SCoP coverage, but also limits how much optimization is possible as less aliasing removes dependencies.
Beside these main passes all kind of interprocedural analysis are beneficial to GRAPHITE. Better constant propagation for example improves coverage as functions that contain a product of a parameter and another dimension become affine as soon as the parameter is know.

5.3.4 Extend the polyhedral model

It is also possible to extend the polyhedral model that is currently defined in GRAPHITE. One possibility is to allow conditions that can contain arbitrary expressions in their condition as long as the expressions do not touch any global state. Therefore the conditions are only allowed to read the values of loop induction variables or parameters. However allowing these conditions would require to keep track of them beside the polytope representation. Another possiblility might be to extend the polytope model to handle parametric strides or array access functions containing non-linear parameters as described in [3]. However at the moment complexity seems to be prohibitive high. Therefore it seems to be necessary to wait for some optimizations that lower complexitiy at least for some special cases.
But even without extensions there is still enought space for improvement in GRAPHITE. So there is time for research and library development to work on tools and

algorithms for future GRAPHITE extensions.

6 Conclusion

After more than 30 years of research on polytope model based optimizations, with gcc 4.4 the first mainstream compiler was released that was able to apply polytope optimizations on low level imperative code. And this not only on small hand selected examples, but on all programs that can be parsed by the gcc. However the implementation is still very limited.

During the last six months the remaining limitations were removed in GRAPHITE branch and by adding polytope data references the polytope model was completed. So the fundaments are set to integrate polytope optimizations into gcc. Nevertheless there was a small part missing. It was not known how much impact GRAPHITE has. Does it actually make sense to write optimizations for GRAPHITE?

This paper shows that there are already 4 benchmarks in the set of 22 analysed SPEC benchmarks where GRAPHITE can extract enough SCoPs to have an noticeable impact without even optimizing it for high coverage. Therefore it seems possible to apply polytope optimizations in an automatic way on “real world” programs. The basis to start porting optimizations to GRAPHITE.

Furthermore several possibilities to extend code coverage in GRAPHITE were shown. On the one hand by extending GRAPHITE itself and on the other hand by improving the gcc analysis passes that are essential for GRAPHITE.

Taking this into account it seems realistic to expect future optimizations based on the polytope model in GRAPHITE to have notable impact. Therefore it is time to start working on both, improving GRAPHITEs coverage and writing optimizations that can take advantage of the model GRAPHITE exports.

References

[1] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.

[2] Cédric Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT’13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[3] Philipp Classen. Code generation in the polytope model with non-linear parameters, 2007.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph, 1991.

[5] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo. In Michael Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers*, volume 21 of *Konferenzen des Forschungszentrums Jülich*, pages 311–320. Forschungszentrum Jülich, 1996.

[6] Martin Griebl, Christian Lengauer, and Sabine Wetzel. Code generation in the polytope model. In *In IEEE PACT*, pages 106–111. IEEE Computer Society Press, 1998.

[7] Jan Sjoedin, Sebastian Pop, Harsha Jagasia, Tobias Grosser, and Antoniu Pop. The design of graphite. 2009.