LetSee: the Legal Transformation SpacE Explorator

Louis-Noël Pouchet, Cédric Bastoul and Albert Cohen
ALCHEMY group, INRIA Futurs and LRI / University of Paris-Sud 11, France

Motivation

The situation

- Usual static models fail to capture the real complexity of modern architectures
- Compiler optimization interactions are hard to interpret

A solution to address these issues

- Use an iterative compilation framework
- Focus on Loop Nest Optimization
- Consider only the result of the application of sequences of transformation
- Model a set of distinct, semantically equivalent program versions

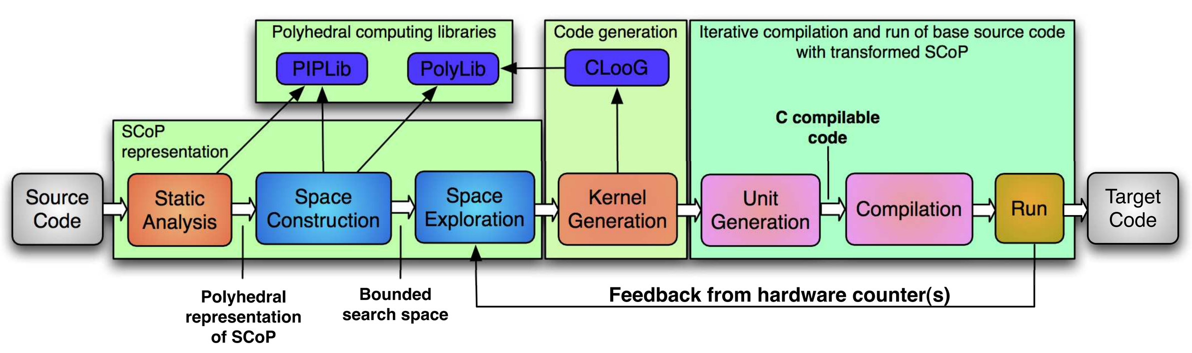
Classical iterative optimization scheme: build a set of sequences of transformations

- Drawbacks:
  - A sequence may not be applied: it can break the semantics of the transformed program
  - Various sequences of transformations may lead to the same target program

Affine schedule-based iterative optimization scheme: build a set of different program versions

- Advantages:
  - Can do an upstream characterization of program candidates preserving the semantics
  - Drastic reduction of the number of possible candidates
    For ex: h264 benchmark: $\approx 1,8 \times 10^8$ distinct versions, only 360 preserve the semantics!
  - On small kernels, exhaustive traversal is possible

An Iterative Tool Chain



1. *Static Analysis*: isolate SCoPs and represent its information using a mathematical algebraic abstraction. The remainder of the program and the actual statement operations are kept apart.
2. *Space Construction*: build a search space encompassing legal and distinct program versions, thanks to its algebraic representation
3. *Space Exploration*: traverse the search space, where each point represent a different program version where the semantics is preserved.
4. *Kernel Generation*: generate the target kernel code corresponding to a point in the search space
5. *Unit Generation*: reinsert all the remainder of the original program plus the instrumentation for performance feedback (`LetSee` uses hardware counters to collect the most accurate information on the program behavior)
6. *Compilation*: compile with a given optimizing compiler targeting a given architecture
7. *Run*: run the program candidate on the target architecture, and gather information about its behavior
8. Use the information collected to drive the exploration according to user objectives (optimize speed, memory footprint, number of cache misses, etc.)

References

- **CLooG**: `http://www.cloog.org`
- **PiPLib**: `http://www.piplib.org`
- **PolyLib**: `http://icps.u-strasbg.fr/polylib`
- **LetSee**: `http://www-rocq.inria.fr/~pouchet`

References

[1] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *International Journal of Parallel Programming*, 21(5):389–420, 1992.

[2] T. Kisuki, P. M. W. Knijnenburg, and M. F. P. O'Boyle. Combined selection of tile sizes and unroll factors using iterative compilation. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 237, Washington, DC, USA, 2000. IEEE Computer Society.

[3] Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and Nicolas Vasilache. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *International Symposium on Code Generation and Optimization*, pages 144–156, San Jose, California, March 2007. IEEE Computer Society.

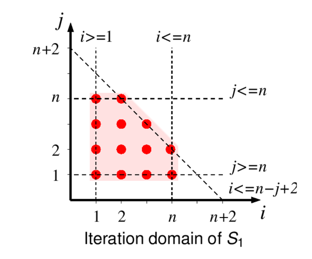
Polyhedral Representation of Programs

Static Control Parts (SCoP)

- Loops and conditionals can be described using affine forms
- Iteration domain: represented as integer polyhedra

```
for (i=1; i<=n; ++i)
. for (j=1; j<=n; ++j)
. . if (i<=n-j+2)
. . . s[i] = ...
```

$$\mathcal{D}_{S1} = \begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & 0 \\ -1 & -1 & 1 & 2 \end{bmatrix} . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \geq \vec{0}$$



Iteration domain of S_1

- Memory accesses: static references, represented as affine functions of the iterators and the parameters

```
for (i=0; i<n; ++i) {
. s[i] = 0;
. for (j=0; j<n; ++j)
. . s[i] = s[i]+a[i][j]*x[j];
}
```

$$f_{\mathbf{s}}^{S2} = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix} . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \quad f_{\mathbf{a}}^{S2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix} \quad f_{\mathbf{x}}^{S2} = \begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} . \begin{pmatrix} i \\ j \\ n \\ 1 \end{pmatrix}$$

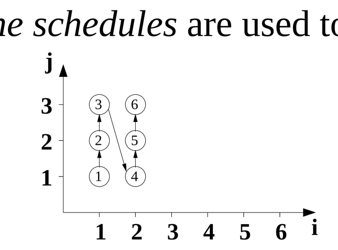
- Data dependence between S1 and S2: a subset of the Cartesian product of $\mathcal{D}_{S1}$ and $\mathcal{D}_{S2}$ (**exact analysis**)

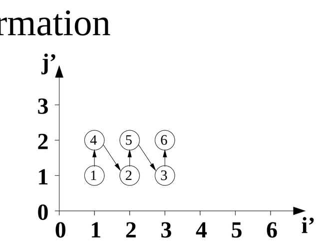
```
for (i=1; i<=3; ++i) {
. s[i] = 0;
. for (j=1; j<=3; ++j)
. . s[i] = s[i] + 1;
}
```

$$\mathcal{D}_{S1\delta S2} : \begin{bmatrix} 1 & 0 & 0 & -1 \\ -1 & 0 & 0 & 3 \\ 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 3 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & -1 & 3 \\ \hline 1 & -1 & 0 & 0 \end{bmatrix} . \begin{pmatrix} i_{S1} \\ i_{S2} \\ j_{S2} \\ 1 \end{pmatrix} \begin{matrix} \geq \vec{0} \\ = 0 \end{matrix}$$

The dependence $S1\delta S2$ only exists between instances of $S1$ and $S2$ such that $i_{S1} = i_{S2}$, and not between all instances of $S1$ and $S2$.

- Reduced dependence graph labeled by dependence polyhedra
- *Affine schedules* are used to drive any composition of loop transformation






```
for (i=1; i<=3; ++i) {
. for (j=1; j<=2; ++j)
. . a[i][j] = a[i][j] * 0.2;
}
```

$$\Longrightarrow \quad \theta_R \begin{pmatrix} i \\ j \end{pmatrix} = \begin{pmatrix} j \\ i \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$$

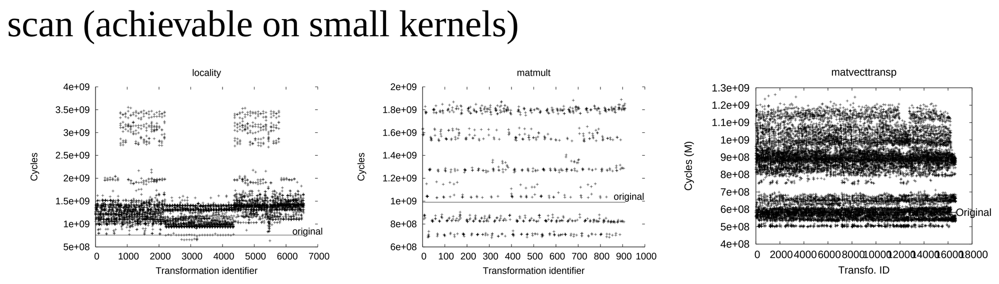
```
for (j=1; j<=2; ++j) {
. for (i=1; i<=2; ++i)
. . a[i][j] = a[i][j] * 0.2;
}
```

Some Experimental Results

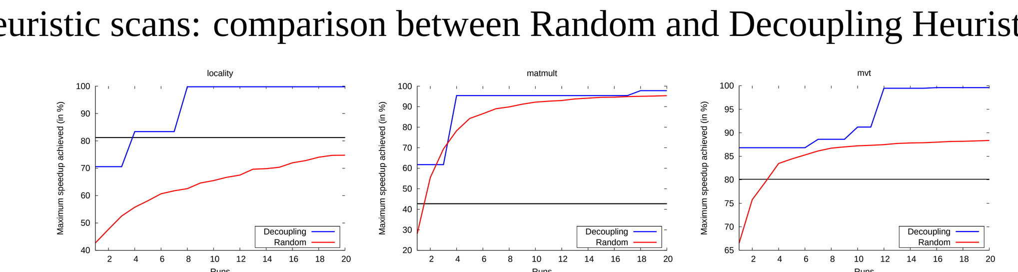
- Dramatic narrowing of the search space, and encouraging speedups: from 10% to 368% on UTDSP kernels, on an AMD Athlon64 machine

Benchmark	Statements	Dependences	Dimension	All	Legal	Iterators	Speedup
`locality`	1	2	1	5.9×10^4	6561	9	19%
`matmult-250`	2	7	1	1.9×10^4	912	76	243%
`h264`	5	15	1	1.2×10^8	360	32	36%
`edge-2048`	3	30	3	1.7×10^{24}	3.1×10^7	1467	40%
`compress-1024`	6	56	2	6.2×10^{24}	6480	9	368%
`latnrm-256`	11	75	2	4.1×10^{18}	1.9×10^9	678	32%
`lmsfir-256`	9	112	2	1.2×10^{19}	2.6×10^9	19962	22%

- Several traversal methods
  - Exhaustive scan (achievable on small kernels)



  - Various heuristic scans: comparison between Random and Decoupling Heuristic



What's Next?

- Using machine learning for space exploration
  - accelerate the space traversal
  - capture the relationship between space variables, and ultimately learn new heuristics
- Integration of `LetSee` in GRAPHITE, the new polyhedral-aware branch of GCC
- Improve scalability and applicability
  - search space exploration problem: equivalent to the dynamic scan of a large integer polytope
  - non static control parts can be amenable to polyhedral modeling with conservative approximation
- Computing search space for multidimensional schedules: a highly combinatorial problem
  - at the moment, we only compute the space corresponding to maximal fine-grain parallelism
  - do we need a new formulation of the space of legal program versions?