Noise: User-Defined Optimization Strategies

Ralf Karrenberg, Marcel Köster, Roland Leißa, Yevgeniya Kovalenko, Sebastian Hack

www.cdl.uni-saarland.de/projects/noise

Setting

Automatic optimization strategies (e.g. "-O3") often do not produce the code that the programmer desires. This can be due to:

- Too imprecise static analysis results
- Cost function deficiencies
- Detrimental optimization effects
- Suboptimal optimization order ("phase ordering problem")

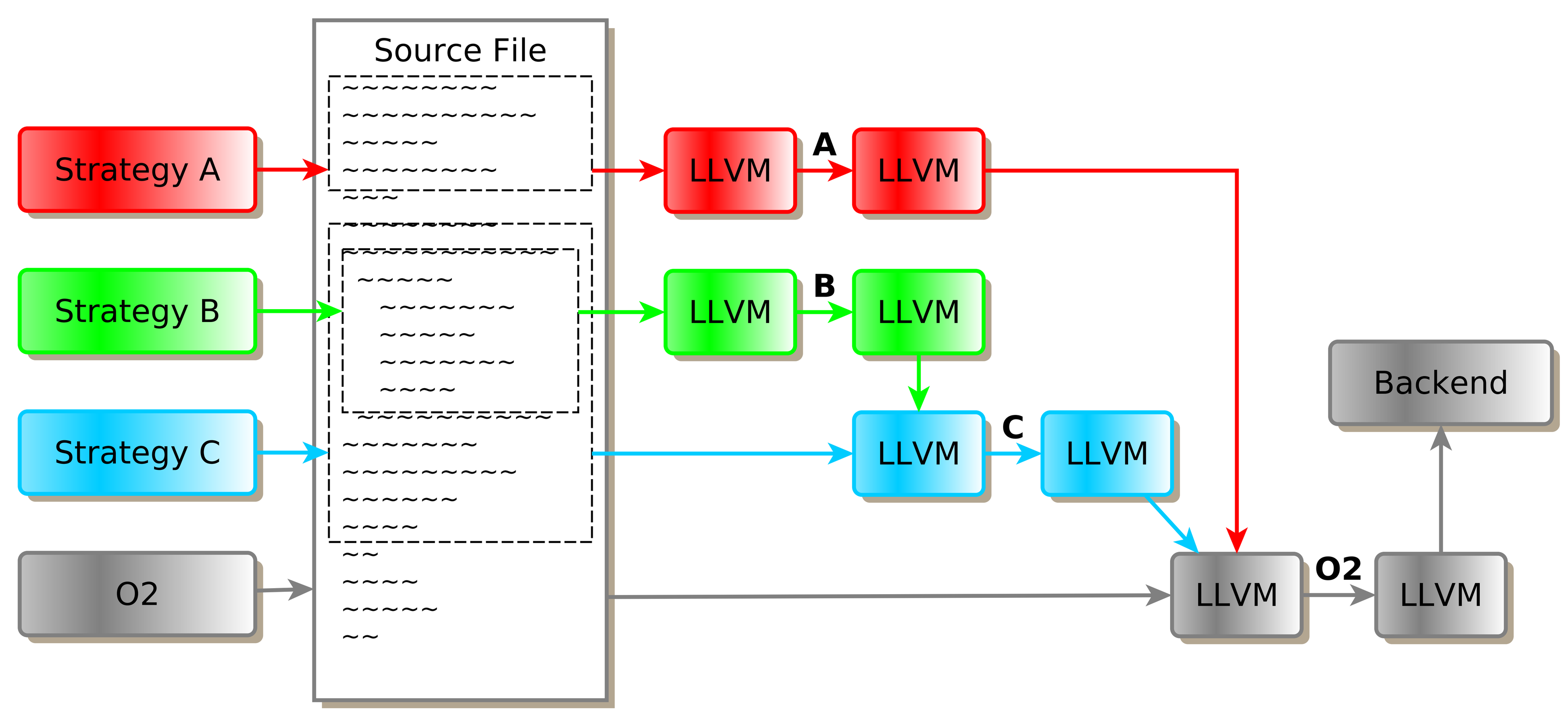
Therefore, programmers often try to outsmart the compiler by manually "optimizing" the code. However, this has a number of disadvantages:

- Time cost
- Error proneness
- Illegible/unmaintainable code
- Does not scale with #target architectures

This is especially important for legacy code in the High-Performance Computing (HPC) environment, but is also relevant in other performance-sensitive fields such as computer graphics.

Noise

- **Language extension for Clang**
- **Create user-defined optimization strategies for code segments**
- **Fine-grained control over applied optimizations**
- **Conveniently tune code without actually rewriting it**
- **Other parts of the program are optimized as before**



Example

```
float g(float x) { return x + 42.f; }

void testNoiseWFV(float x, float* in, float* out) {
  NOISE("loopfusion inline(g) vectorize(8) unroll(4)")
  {
    for (int i=0; i<32; ++i) {
        float lic = x * g(x);
        out[i] = in[i] + lic;
    }
    for (int i=0; i<32; ++i) {
        out[i] *= x;
    }
  }
}
```

Transformations

The current implementation allows to employ all transformations available in LLVM under the LLVM-internal names (e.g. dead code elimination [*dce*] and loop invariant code motion [*licm*]). Additionally, we implemented the following special-purpose transformations:

Function Inlining

Force inlining of specific function calls without relying on the compiler's heuristics. This possibly allows additional optimization opportunities afterwards, e.g. transformations that would have to be inter-procedural before now can be applied locally.

Explicit Loop Unrolling

We provide the possibility to both rely on LLVM's heuristics for unrolling or to force it explicitly with *unroll(N)*. If N is not supplied, the phase itself decides whether and how the loop should be unrolled.

Loop Vectorization

In addition to the LLVM-internal phases *bb-vectorize* and *loop-vectorize* we provide *wfv-vectorize*, a wrapper around libWFV that can be used to vectorize data-parallel loops.

Loop Fusion

Fuse multiple loops into a single one by merging their bodies. Annotated loops are not required to directly succeed each other. This enables complex combinations of loop fusion and code motion.

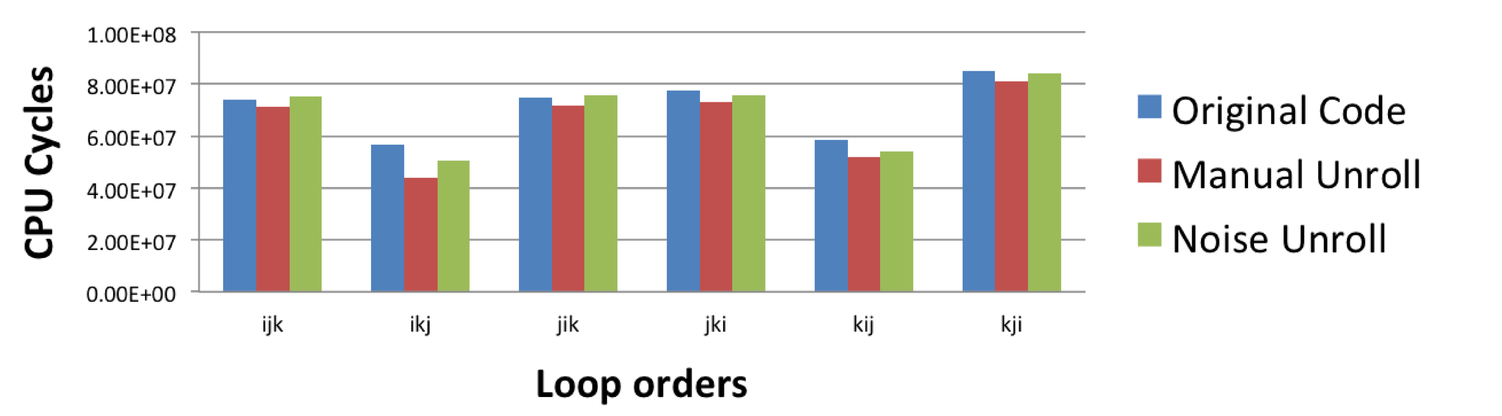
Specialized Loop Dispatching

Create specialized variants of the annotated loop and introduce a dynamic dispatcher (case distinction on the specialized variable). Uncover further optimization potential by exploiting knowledge about runtime values of a variable.

Preliminary Results

We are currently evaluating Noise in an HPC environment:

- Performance-critical regions of molecular dynamics legacy code.
- First results confirm applicability, usability, and improved work-flow.
- Phase-ordering still a problem, but now transparent to programmer.



Result (Pseudo Code)

```
void testNoiseWFV(float x, float* in, float* out) {
  <8 x float>* inv  = (<8 x float>*)in;
  <8 x float>* outv = (<8 x float>*)out;
  float lic = x * (x + 42.f);
  outv[0] = SIMD_mul(SIMD_add(inv[0], lic), x);
  outv[1] = SIMD_mul(SIMD_add(inv[1], lic), x);
  outv[2] = SIMD_mul(SIMD_add(inv[2], lic), x);
  outv[3] = SIMD_mul(SIMD_add(inv[3], lic), x);
}
```

Acknowledgements

UNIVERSITÄT DES SAARLANDES · intel · VISUAL COMPUTING INSTITUTE · HLRS · ECOUSS · Bundesministerium für Bildung und Forschung