Transparent Parallelization of Binary Code

Benoît Pradelle Alain Ketterlin Philippe Clauss

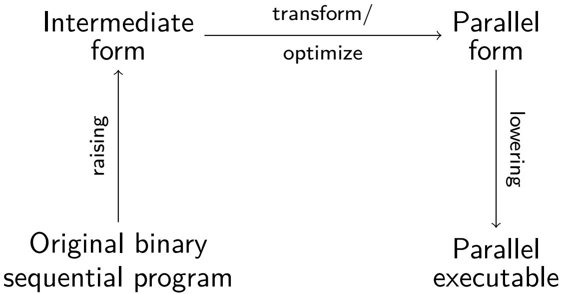
Université de Strasbourg

INRIA (CAMUS team, Centre Nancy Grand-Est)

CNRS (LSIIT, UMR 7005)

First International Workshop on
Polyhedral Compilation Techniques (IMPACT 2011)
Sunday April 3, 2011

Overview



Intermediate form

transform/ optimize

Parallel form

raising

lowering

Original binary sequential program

Parallel executable

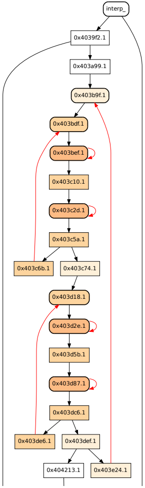
1. Bring the (x86-64) code into “something usable”
2. Apply parallelizing transformations
3. Translate back into “something executable”
4. Empirical evaluation

Raising / Decompiling x86-64

1. Rebuild CFG
2. Natural loops
3. Points-to to discriminate
    - current stack frame
    - “outer” memory

   → track stack slots
4. SSA
5. Slicing/symbolic analysis
    - memory addresses
    - branch conditions
6. Induction variables

   → normalized counters
7. Control dependence
    - trip-counts
    - block constraints
8. Loop selection



$\times 2$

Raising / Decompiling x86-64

1. Rebuild CFG
2. Natural loops
3. Points-to to discriminate
    - current stack frame
    - “outer” memory

   → track stack slots
4. SSA
5. Slicing/symbolic analysis
    - memory addresses
    - branch conditions
6. Induction variables

   → normalized counters
7. Control dependence
    - trip-counts
    - block constraints
8. Loop selection

```
mov [rsp+0xe8], 0x2  ; -> _V_42.0
L1:
_V_42.1 = ϕ(_V_42.0,_V_42.2)
        ; @ [rsp_{1} - 0x2e0]
        = 2 + I
...
mov rax^{29}, [rsp+0xf0]

    L2:
    rax^{30} = ϕ(rax_{29},rax_{31})
             = ... + J*0x8
    ...
    addsd xmm1, [rax_{30}]
              ; @ ... + 8192*I + 8*J
    ...
    add rax^{31}_{30}, 0x8
    jmp L2

add [rsp+0xe8], 0x1  ; -> _V_42.2
...
jmp L1
```

Raising / Decompiling x86-64

→ affine loop nests over a single array `M`

```
xor ebp, ebp
mov r11, rbp
for (t1 = 0; -1023 + t1 <= 0; t1++)
    for (t2 = 0; -1023 + t2 <= 0; t2++) {
        mov M[23371872 + 8536*t1 + 8*t2], 0x0
        mov M[rsp.1-0x30], r11
        movsd xmm1, M[rsp.1-0x30] // <- 0.
        for (t3 = 0; -1023 + t3 <= 0; t3++) {
            movsd xmm0, M[6299744 + 8296*t1 + 8*t3]
            mulsd xmm0, M[14794848 + 8*t2 + 8376*t3]
            addsd xmm1, xmm0
        }
        movsd M[23371872 + 8536*t1 + 8*t2], xmm1
    }
```

→ *almost* directly usable

Parallelizing / Adapting to the tools...

- Outlining: exact instructions do not matter, shown as ⊙
- ~~Array reconstruction: split memory into disjoint pieces~~
  ~~Note: parametric bounds would lead to runtime checks~~
  (not really needed anymore...)
- Forward substitution of scalars
- The previous example becomes

```
for (t1 = 0; -1023 + t1 <= 0; t1++)
    for (t2 = 0; -1023 + t2 <= 0; t2++) {
        A2[t1][8*t2] = 0
        xmm1 = 0
        for (t3 = 0; -1023 + t3 <= 0; t3++)
            xmm1 = xmm1 ⊙ ( A1[t1][8*t3]  ⊙ A3[t3][8*t2] )
        A2[t1][8*t2] = xmm1
    }
```

Parallelizing / Removing scalars

- Scalar expansion, then transformation?
- We don't want this!

```
for (t1 = 0; t1 <= 1023; t1++)
  for (t2 = 0; t2 <= 1023; t2++)
    xmm1[t1][t2] = 0;
for (t1 = 0; t1 <= 1023; t1++)
  for (t2 = 0; t2 <= 1023; t2++)
    for (t3 = 0; t3 <= 1023; t3++)
      xmm1[t1][t2] = xmm1[t1][t2] ⊙ (A1[t1][8*t3] ⊙ A3[t3][8*t2]);
for (t1 = 0; t1 <= 1023; t1++)
  for (t2 = 0; t2 <= 1023; t2++)
    A2[t1][8*t2] = xmm1[t1][t2];
```

Parallelizing / Removing scalars

- Instead we do "backward substitution":

```
A2[t1][8*t2] = 0
xmm1 = 0
for (t3 = 0; -1023 + t3 <= 0; t3++)
    xmm1 = xmm1 ⊙ (A1[t1][8*t3]⊙A3[t3][8*t2])
A2[t1][8*t2] = xmm1
```

becomes

```
A2[t1][8*t2] = 0
for (t3 = 0; -1023 + t3 <= 0; t3++)
    A2[t1][8*t2] = A2[t1][8*t2] ⊙ (A1[t1][8*t3]⊙A3[t3][8*t2])
[ xmm1 = A2[t1][8*t2] ]
```

- Restrictions:
  - no data dependence (we use `isl`)
  - no complex mixing with other registers
- If we can't back-substitute, we need to "freeze" the fragment

Parallelizing / PLUTO

run PLUTO

Lowering / Restoring semantics

- Identifying statements (note: some have been moved, some duplicated... — we do not tolerate fusion/splitting)
  - Thanks PLUTO for providing stable numbering
- The resulting nest(s) is(are) made of abstract statements
  - acting on memory cells, with address expressions
  - using registers for intermediate results

→ generating C is simpler than reusing the original code

Lowering / Restoring semantics

- Identifying statements (note: some have been moved, some duplicated... — we do not tolerate fusion/splitting)
    - Thanks PLUTO for providing stable numbering
- The resulting nest(s) is(are) made of abstract statements
    - acting on memory cells, with address expressions
    - using registers for intermediate results

→ generating C is simpler than reusing the original code

- Memory addresses are cast into pointers:

```
(void*)(23371872+8536*t4+8*t5)
```

- Loads and stores use intrinsic functions

```
xmm0 = _mm_load_sd((double*)(6299744+8296*t4+8*t7));
_mm_store_sd((double*)(23371872+8536*t4+8*t5), xmm1);
```

- Basic operations use intrinsics as well:

```
xmm1 = _mm_add_sd(xmm1, xmm0);
```

Lowering / Restoring semantics

```
#pragma omp parallel for private(t2,t3,t4,t5)
for (t2=0; t2<=1023/32; t2++)
  for (t3=0; t3<=1023/32; t3++)
    for (t4=32*t2; t4<=min(1023,32*t2+31); t4++)
      for (t5=32*t3; t5<=min(1023,32*t3+31); t5++) {
        void *tmp0 = (void*)(23371872 + 8536*t4 + 8*t5);
        asm volatile("movq $0, (%0)":: "r"(tmp0));
      }
#pragma omp parallel for private(t2,t3,t4,t5,xmm0,xmm1)
for (t2=0; t2<=1023/32; t2++)
  for (t3=0; t3<=1023/32; t3++)
    for (t4=32*t2; t4<=min(1023,32*t2+31);t4++)
      for (t5=32*t3;t5<=min(1023,32*t3+31);t5++) {
        double tmp1 = 0.;
        xmm1 = _mm_load_sd(&tmp1);
        for (t7=0; t7<=1023; t7++) {
          xmm0 = _mm_load_sd((double*)(6299744 + 8296*t4 + 8*t7));
          __m128d tmp2 = _mm_load_sd((double*)(14794848 + 8*t5 + 8376*t7));
          xmm0 = _mm_mul_sd(xmm0, tmp2);
          xmm1 = _mm_add_sd(xmm1, xmm0);
        }
        _mm_store_sd((double*)(23371872 + 8536*t4 + 8*t5), xmm1);
      }
```

Lowering / Monitoring execution

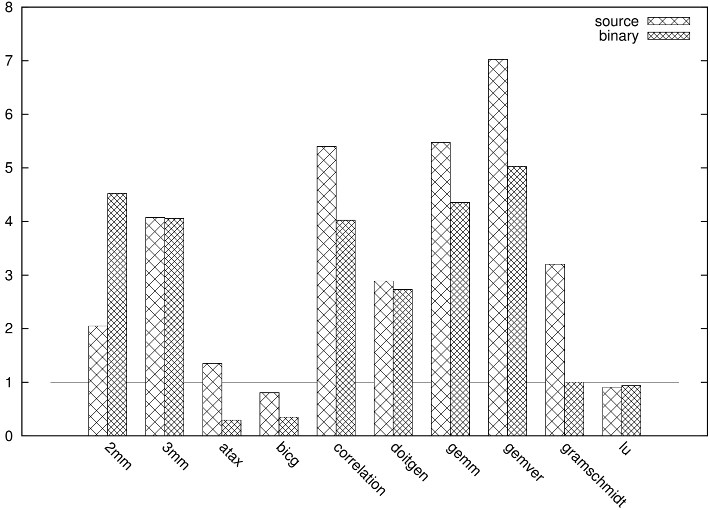
- Transformed/parallelized loop nests
  - are compiled as functions with `gcc`
  - and placed in a shared library
- We use run-time monitoring to replace a loop nest
  - the monitoring process `ptrace`-s the child
  - the child process runs the original executable
  - breakpoints are set at loop entry
  - and loop exit
  - the monitor redirects (parallelized) loop executions
- If you think this is too complex... you're right
  (we have a hidden agenda)

Results / Coverage

- On `polybench 1.0`, compiled with `gcc -O2` (4.4.5)

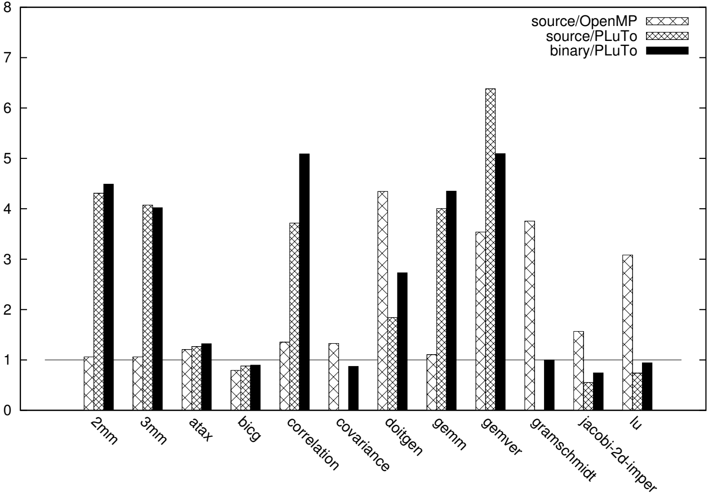
Benchmark	Parallelized	In source	Rate
`2mm`	7	7	100%
`3mm`	10	10	100%
`atax`	2	2	100%
`bicg`	2	2	100%
`correlation`	3	5	60%
`doitgen`	3	3	100%
`gemm`	4	4	100%
`gemver`	3	4	75%
`gramschmidt`	1	2	50%
`lu`	1	2	50%
Sum	36	41	87.8%

Results / Speedup



Results / Speedup

- Intel Xeon W 3520, 4 cores



Conclusion

What about “real” programs?

- Parameters everywhere:
  - loop bounds
  - access functions
  - block constraints

  → conservative dependence analysis, and runtime-tests
- Address expressions like:

  `0x8 + 8*rbx.3 + 8*rdx.3 + 1*rsi.1`
  `+ 8*K + 8*rbx.3*J + 8*rdx.3*I`
- “Fake” non-static control (on floating point values)
  → what exactly is a statement?

Shameless advertisement!

Benoît PRADELLE (`b.pradelle@gmail.com`)
Expert in:

- Runtime selection of parallel schedules
- Parallelization of binary code
- and more

Will graduate December 2011

Available January 2012 for a post-doc position