Practical fine-grained static slicing of optimized code

Michael Ernst
University of Washington
(work done at Microsoft Research)
Simple example

```plaintext
a = foo();
b = bar();
c = a * b;
d = a - b;
e = b + c;
f = c * e;
g = e - c;
h = f / g;
i = g * h;
j = g / 2;
```
Example: function calls and pointers

```
main(int a, int b)
{
    int sum = swapsum(&a, &b);
    printf("%d %d %d", a, b, sum);
}

swapsum(int *x, int *y)
{
    int z;
    z = *x;
    *x = *y;
    *y = z;
    z = (*x) + (*y);
    return(z);
}
```

```
main(int a, int b)
{
    int sum = swapsum(&a, &b);
    printf("%d %d %d", a, b, sum);
}

swapsum(int *x, int *y)
{
    int z;
    z = *x;
    *x = *y;
    *y = z;
    z = (*x) + (*y);
    return(z);
}
```
Applications of slicing

Closure slicing: visualize dependences
  - program understanding
  - maintenance
  - test coverage
  - debugging

Executable slicing: produce binary
  - specialization
  - parallelization
  - testing
  - integration of program versions

Dynamic slicing: execution tracing
Outline

Motivation

Basic slicing algorithm

Interprocedural

Pointers and aggregate values

Optimized code

Executable slicing

Conclusion
Simple slicing algorithm

For graph-based program representation:

Just graph reachability!
The value dependence graph (VDG)

Sparse, functional, parallel representation for imperative programs

Insights:
- Only values matter
- Original names and control flow are incidental

Consequences:
- All values are explicit
- Select values, not control paths
- Control flow represented by function calls
Components of the value dependence graph

- Operation nodes
- Selectors
- Closures (functions)
- Function calls
- Memory lookups (load instructions)
- Memory assignments (store instructions)
void sum_product(int n) {
    int sum = 0;
    int product = 0;
    int i = 0;

    while (i++ < n) {
        sum = sum + i;
        product = product * i;
    }
    printf("Sum %d, product %d",
            sum, product);
}
Example VDG, sliced

```c
void sum_product(int n)
{
    int sum = 0;
    int product = 0;
    int i = 0;

    while (i++ < n)
    {
        sum = sum + i;
        product = product * i;
    }

    printf("Sum %d, product %d", sum, product);
}
```
The VDG is good for slicing

- Simple, fast algorithm
- Fine granularity
- One graph directly links producers and consumers
- All values are explicit
Expression results, not whole computations
Any expression
Any result, including unnameable ones
Unreferenced variables
Outline

Motivation
Basic slicing algorithm
Interprocedural
Pointers and aggregate values
Optimized code
Executable slicing
Conclusion
Interprocedural slicing

Goals:

- sensitive to calling context
- do not include entire procedure or all calls
- efficient
- omit irrelevant procedures

Obvious solution: graph reachability

- call results \( \supseteq \) procedure returns
- formal parameters \( \supseteq \) actual parameters

This does not satisfy our goals.
Interprocedural slicing (solution)

- Summarize dependences of returns on formals
- Separately include appropriate portions of body
- When slicing criterion is within a procedure, include all calls

```plaintext
int increment(int i); { return (i+1); }

a = foo();
b = bar();
x = increment(a);
y = increment(b);
```

This meets our goals.
Procedure summary dependences

Optimistic forward dataflow problem
- operation result depends on what the operands depend on
- for calls, use current approximation as transfer function
- reprocess calls when new approximation becomes available
- iterate until fixpoint is reached

Complexity
Outline

Motivation
Basic slicing algorithm
Interprocedural
Pointers and aggregate values
Optimized code
Executable slicing
Conclusion
Pointers

Support arbitrary pointer manipulations
Points-to analysis gives possibly referenced locations

For lookup, slice on location and (part of) store

\[ \ldots x \ldots = \text{store} \]

\[ \ldots \ast x \ldots \]

\[ \text{store} \]

\[ \text{Lookup} \]
**Treat store as a collection**

Slice on every possibly referenced location.

Slice continues at
- killing def: location, value (use pointer equality)
- preserving def: location, value, store
- other: store

---

**Lookup**

store

\[ x \]

**Update**

store

\[ a \]

\[ 22 \]
Aggregate values

Just like the store.

Complexity
Outline

Motivation
Basic slicing algorithm
Interprocedural
Pointers and aggregate values
Optimized code
Executable slicing
Conclusion
Slicing optimized code

Improve:
- dependences
- liveness
- computations
- overhead

The hardware runs optimized code

Slice reflects intermediate representation's semantics
Correspondence with source code

Need many-to-many mappings
- VDG node ➔ source text
- source text ➔ VDG node

Maintain a separate source graph
- initially isomorphic to VDG
- transformations modify VDG and source correspondences
- slicing traverses both graphs in tandem
- slice display defaultly highlights only appropriate sources
**History mechanism**

Source graphs are never side-effected or removed
Transformations add to source graph
Mappings no longer necessarily inverses

This enables
- undoing transformations
- explaining changes
- slice according to naive interpretation
- slice dead code
- statement-oriented display
Outline

Motivation
Basic slicing algorithm
Interprocedural
Pointers and aggregate values
Optimized code
Executable slicing
Conclusion
Executable slicing

Goal: executable binary

Applications:
- specialization
- parallelization
- testing
- integration of program versions
Traditionally, slicing is achieved by:

- Subsetting the original program.
- Compiling the subset using a standard compiler.
Compilable slicing tradeoffs

Retains context, comments, formatting, names

Include undemanded portions of the program to satisfy syntactic constraints
- multi-valued computation
- function call parameters
- variable assignments
- control flow: goto, continue, break

Limits optimization
Hard to undo
Our executable slicing solution

Generate code directly from slice
For debugging, use original program
Outline

Motivation
Basic slicing algorithm
Interprocedural
Pointers and aggregate values
Optimized code
Executable slicing
Conclusion
Status

Part of a programming environment
Handles C (except `longjmp`)
Written in Scheme
Integrated with Emacs
Future work

Quasi-static slicing
Dynamic slicing
History mechanism
User interface alternatives
Tests on real users
Debugging optimized code
Ambiguity in program points

\[
\begin{align*}
  z &= \ldots; \\
  a &= b + c; \\
  z &= x \times y;
\end{align*}
\]
Comparing the VDG and PDG

Which PDG?

VDG:
- no implicit quantities
- better integration with programming environment
- easier analysis, transformation
- cleaner interprocedural representation
- single graph
- finer granularity