DCG: Efficient Retargetable Dynamic Code Generation

Dawson Engler
MIT Lab for Computer Science

Todd Proebsting
University of Arizona

What & Why

Executable code:

- Sequence of bits corresponding to machine instructions

Dynamic code generation:

- Generation of executable code at runtime

Why?

- Optimizations can exploit runtime information
- Order of magnitude improvements in performance

Examples

Interpreters can dynamically compile code:

- Smalltalk-80 [Deutsch et al., 1984]
- Self [Chambers et al., 1989]

Runtime constants can be exploited:

- Strength-reduce multiplication by runtime constants

OS specialization:

- Synthesis [Massalin et al., 1988]

Concrete example: scaling a matrix

Compile time (k = ???): $\begin{bmatrix} x & y \\ w & z \end{bmatrix} \times k$

k is a *runtime constant*:

- Exploit using dynamic code generation

Runtime (k = 3): $\begin{bmatrix} x & y \\ w & z \end{bmatrix} \times 3$

After strength-reduction: $\begin{bmatrix} (x{<<}1+x) & (y{<<}1+y) \\ (w{<<}1+w) & (z{<<}1+z) \end{bmatrix}$

Available solutions are not adequate

Method 1: Generate C code, farm out to compiler and link

- Reasonably portable, high-level specification
- Slow: ~30,000 instructions/generated instruction

Method 2: Generate executable code "by hand"

- Very fast: ~10 instructions/generated instruction
- Very low-level and machine (or even chip) specific

Problem: machine independence + efficiency

Our solution: the DCG system

Portability:

- **Machine-independent code specification**

Efficiency:

- **1-pass code generation (350 insn/generated insn)**

Retargetability:

- **RISC retarget takes approximately 1 week**
- **Runs on MIPS Rx000s and ABI compliant SPARCS**

Operational overview (recipe for bits)

(1) Application uses DCG's libraries to construct IR

(2) Application gives IR to DCG's retargetable backend

(3) DCG consumes IR and emits code (1 pass)

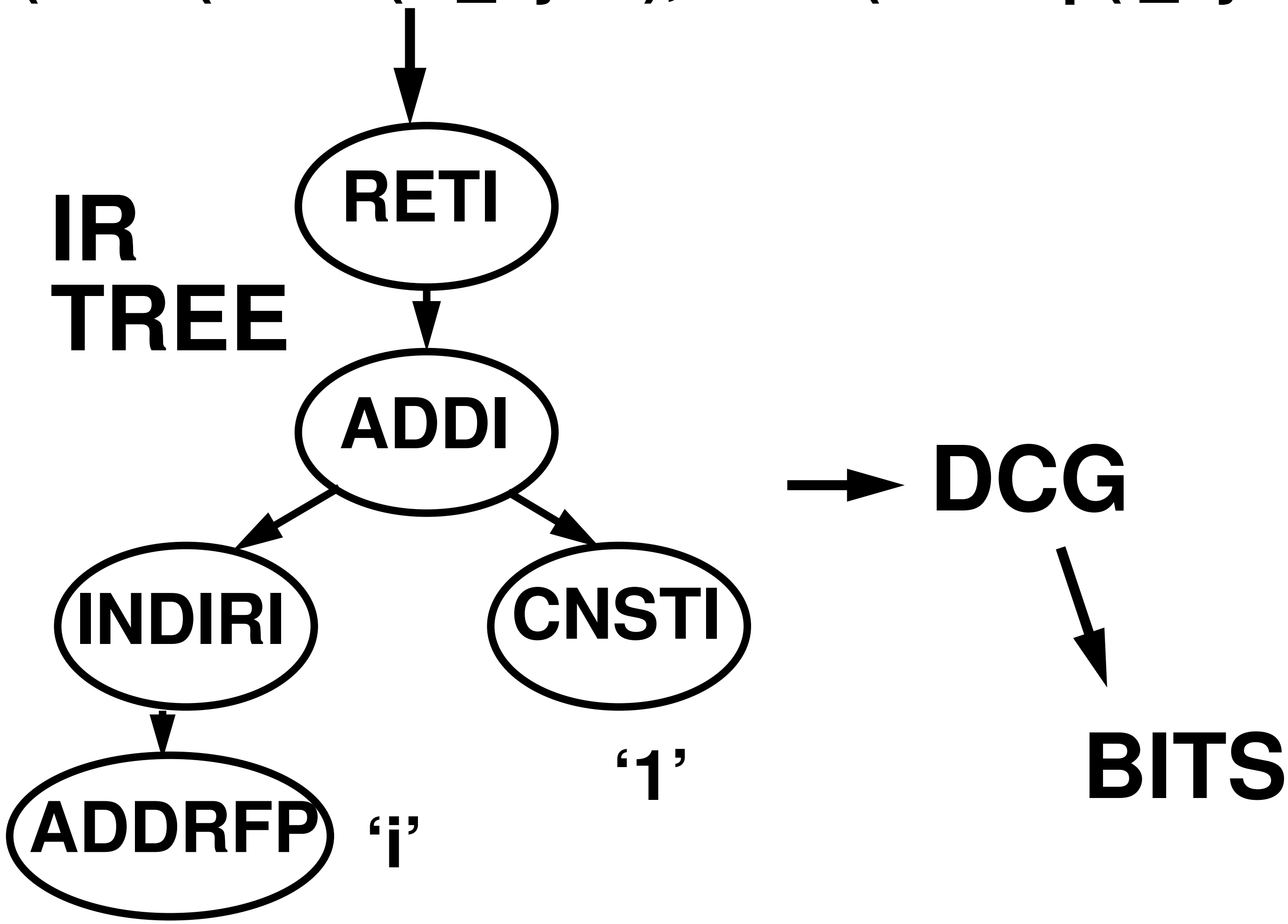
(4) DCG does cleanup (e.g., flushing I/D caches)

(5) DCG returns pointer to new code to application

(6) Application can then call the new function

IR tree for: int plus1(int i) {return i+1;}

reti(addi(cnsti(1_sym), indiri(addrfp(i_sym))))



Efficiency: 1-pass code generation

Code generation:

- Allocate registers (Sethi-Ullman)
- Select instructions using BURS pattern matching
- Emit executable code (the actual bits)
- Backpatch unresolved jumps (linking)

Cost:

- Measured in "bad case": 350 instructions/generated instruction
- ~ 100 times faster than using C compiler

Retargetability: lcc's IR

lcc: ANSI C compiler [Fraser & Hansen, 1991]

Code specified using lcc's IR trees:

- Proven IR (retargets to MIPS/SPARC/x86/Vax)
- Simple and clean (109 total nodes)

Use machine spec and preprocessors to retarget

Example 1: Scaling of matrices

Treat scaling factor as a *runtime constant*

A * k (multiply) is strength-reduced to shifts and adds:

- ~1.3 x faster on MIPS R3000
- ~4 x faster on SPARC10

A / k (divide) is transformed into multiplication [Granlund et al., 1994]:

- ~2 x faster on R3000
- ~4 x faster on SPARC 10

Example 2: “Compiling” interpreter

Small C-like language for interactive use

Code is entered, compiled directly and then run

Speed from elimination of interpretation overhead

Example, fibonacci:

- Within factor of 2 of “GCC -O2”
- 50 x faster than interpreter on SPARC 10
- 20 x faster than interpreter on MIPS R3000

Example 3: Mogul's packet-filter

Packet-filter:

- **Used for packet demultiplexing**
- **Interpreted predicate that outputs {accept, reject}**

Compiling interpreter:

- **~10 times faster than a threaded interpreter using indirect gotos (GNU C extension)**

Example 3: Mogul's packet-filter

Packet-filter packet demultiplexer:

- Interpreted predicate that outputs {accept, reject}

Implementation 1 (threaded interpreter):

- Indirect goto on packet-filter opcode

Implementation 2 (dynamic code generation)

- Translate packet-filter into DCG IR; generate code

Dynamic code generation approximately 10 x faster

Future Work

For DCG:

- Instruction scheduling
- More aggressive peephole optimization
- Eliminate the reliance on C calling conventions

In general:

- Currently adding dynamic code generation extensions to ANSI C

Conclusions

Portable efficient dynamic code generation is possible:

- Current technology allows simple implementation
- Retargeting can be done with only modest effort
- IR interface can be used to generate code efficiently
- Application performance can improve by an order of magnitude