‘C and tcc: A Language and Compiler for Fast, High-Level Dynamic Code Generation

Massimiliano Poletto Dawson Engler

Frans Kaashoek Eddie Kohler

MIT Lab for Computer Science

Wilson Hsieh

University of Washington

Dynamic Code Generation: What & Why

Generate code while the application is running

Exploit information about run-time invariants

Improve performance by orders of magnitude

- Image processing algorithms
- Compiling interpreters (*i.e.*, Java bytecode)
- Network packet filters

Other Dynamic Code Generation Designs

- Implicit

  Smalltalk-80 [Deutsch et al., 1984]

  Self [Chambers et al., 1989]

- Programmer-supplied hints

  Fabius [Leone, Lee, 1996]

  C Run-Time Code Generation [Consel et al., 1996]

  SPIN [Chambers et al., 1996]

Explicit Dynamic Code Generation

Fast but hard:

Synthesis [Massalin et al., 1988]

DCG [Engler, Proebsting, 1994]

VCODE [Engler, 1996]

Easy but slow:

LISP 'eval'

‘C: Effective Explicit Dynamic Code Generation

Language: Programming must be convenient

- Expressive
- Portable
- Easy to use

Implementation: Dynamic code generation must be efficient

- Fast dynamic code generation (DCG)
- Good code quality

Outline

The 'C language

- Operational overview
- Extensions to C
- Some examples

The tcc compiler

- Design and implementation
- Some results

Current status and future work

‘C: Operational Overview

- **Edit time:** Programmer uses ‘C to specify dynamic code
- **Compile time:** Compiler emits static code and code to generate dynamic code
- **Run time:**

  Code specifications are composed and compiled to executable form

  Dynamic code can be called as a function

‘C: Phases of Dynamic Compilation

`b = ‘{ printf("hello world"); };`	Specification
⋮	
`f = compile(b);`	Instantiation
⋮	
`(*f)();` ⟶ hello world	Execution

‘C Extensions to C

		2 type constructors		3 operands		3 special forms
‘C = ANSI C	+	cspec vspec	+	‘ @ $	+	local param compile

Minimizing changes:

- Eases learning ‘C
- Simplifies and encourages conversion of existing C code
- Simplifies the implementation

‘C Extensions: Type Constructors

- **cspec** (code specification): type of dynamic code

```
void cspec c = `{ printf("hi"); };
```

- **vspec** (variable specification): type of “dynamic lvalues” (storage class determined at run time)

```
double vspec d;
```

‘C Extensions: Special Forms

- **local**: returns a vspec for a local variable of the given type

```
int vspec y = local(int);
```

- **param**: returns a vspec for an argument of the given type

```
float vspec arg3 = param(float,3);
```

- **compile**: compiles a cspec, returning a pointer to a function

```
int (*f)() = compile(‘{return 2; }, int);
(*f)(); ⟶ 2
```

‘C Extensions: the ’@’ Operator

@ (at) denotes composition of vspecs and cspecs

```
// Generate a function which adds 1 to its input

int vspec arg0 = param(int, 0);

int cspec sum = `(1+@arg0);

void cspec mesg = `{ printf("The result is: %d", @sum); };

int (*plus1)(int) = compile(`{ @mesg; return @sum;}, int);
```

‘C Extensions: the '$' Operator

$ (dollar) denotes run-time constants

```
int cspec c1, cspec c2;
void cspec c;
void (*f)();

int x = 1;
c1 = `$x;              // Bind x at specification time: x = 1
c2 = `x;               // Bind x at execution time: x = 3
c = `{ printf("spec: x= %d, exec: x = %d\n", @c1, @c2); };

x = 2;
f = compile(c, void);   // Compile

x = 3;
(*f)();                 // Run: will print "spec: x = 1, exec: x = 3"
```

Example: Dot Product

A fixed vector, v0:

17	0	0	4

Another vector, v1:

45	8	19	5

Traditionally:

```
return v0[0]*v1[0]+v0[1]*v1[1]+v0[2]*v1[2]+v0[3]*v1[3];
```

With dynamic code generation:

```
return v1[0]*17+v1[3]*4;
```

With dynamic code generation and strength reduction:

```
t = v1[0]; return t+t<<4+v1[3]<<2;
```

Dot Product Using DCG

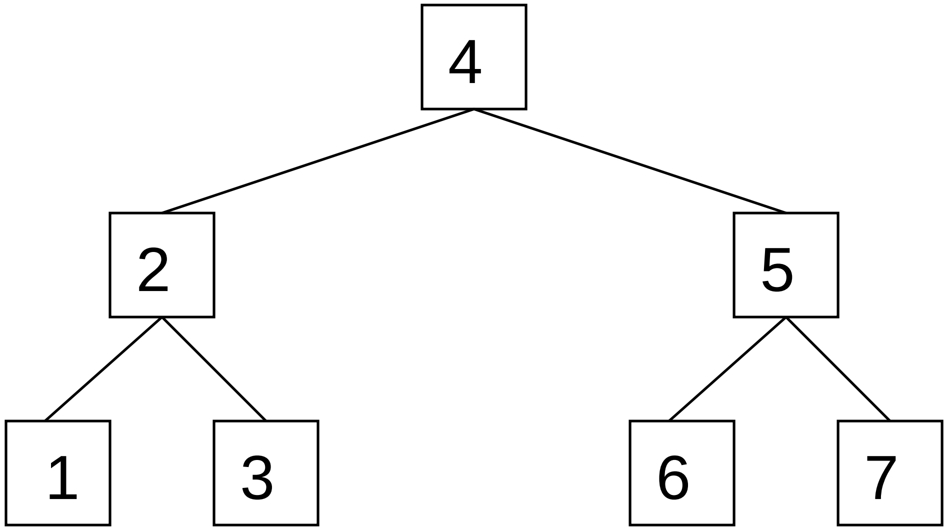
```
// return function that computes the dot product (v0 dot v1)

int (*)(int *) mkdot(int *v0, int n) {
      int k;
      int * vspec v1 = param(int *, 0);
      int cspec sum = `0;

    for (k = 0; k < n; k++)
          if (v0[k])
            sum = `(@sum + (@v1)[$k] * $v0[k]);

    return compile(`{ return @sum; }, int);
}
```

Example: Binary Search

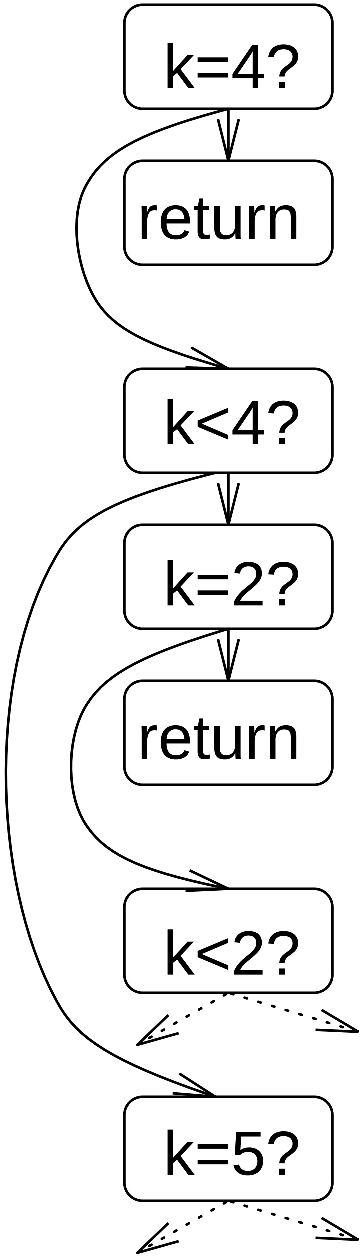
A sorted list:

1	2	3	4	5	6	7

Binary search tree:



DCG code for binary search:



Binary Search

```
void            bin(int *x, int       key, int l,  int u, int r) {
   int p;
   if(l > u)
             return -1;           // failed;           return -1
   p = u - r;                     // current position


    if( x[p] ==  key) return  p;    // found it! return index
    else if( x[p] <  key)
          bin(x, key, p+1, u, r/2);  // too small; search right
    else
          bin(x, key, l, p-1, r/2);  // too big; search left

}
```

```
  cspec gen                    vspec

  return '{               };                    generate "          "

// generate binary comparison tree
return ' {
    $           @             $
          $        @
  @gen

  @gen
};
```

Outline

The ‘C language

- Operational overview
- Extensions to C
- Some examples

The tcc compiler

- Design and implementation
- Some results

Current status and future work

Implementation Issues

Dynamic code generation must be fast

Dynamic code quality must be good

How?

- Do as much as possible at static compile time
- Efficient dynamic register allocation
- Inexpensive optimizations at instantiation time

Language Features for Efficiency

Static typing

Do type checking and most instruction selection statically

Lexical scoping

Find free variables of dynamic code statically

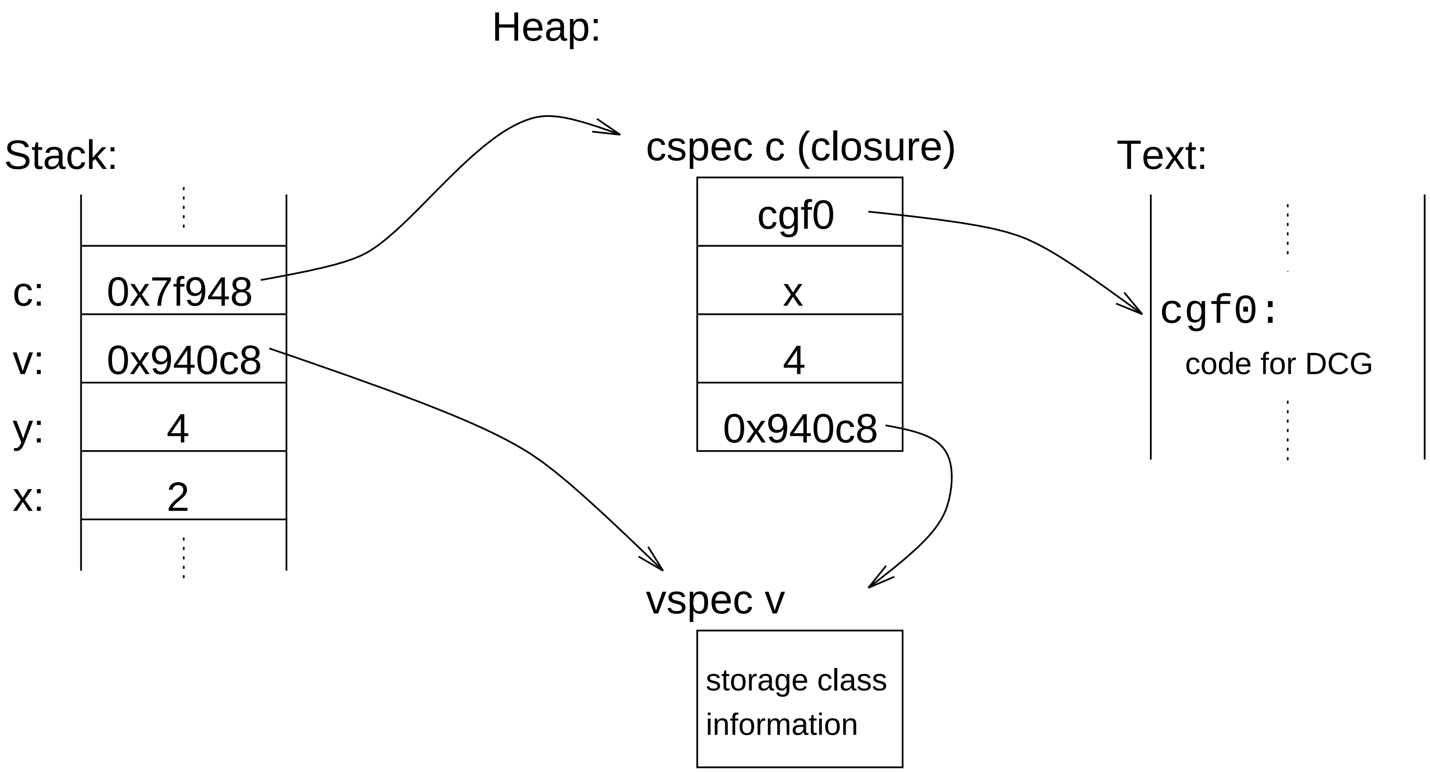
Do less at instantiation time:

- Some register allocation
- Dynamic optimizations
- Emitting and linking dynamic code

Compiling Dynamic Code

Parse and check cspec
Emit code to construct closure for specification-time info
Emit code-generating function for cspec

```
int x = 2, y = 4; int vspec v;
  ⋮
void cspec c = `{ return x+$y+@v; };
```



Code-Generating Functions

Each cspec compiles to a code-generating function (CGF) invoked at dynamic compile time

Contents of a CGF:

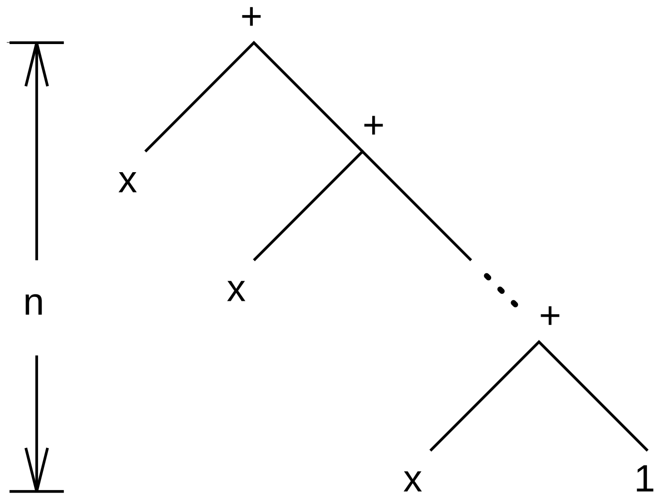
- VCODE macros and register allocation directives
- Code to perform optimizations
- Calls to other CGFs (composition of cspecs)

Dynamic Register Allocation

Efficiently allocating registers across cspecs is hard

Example:

```
for (c=`1, i=0;i<n;i++) { c = `(x + @c); }
```



Our heuristic is fast and usually produces good code

- Restructure expression trees
- Minimize registers live across a call to a CGF

Optimizing Run-Time Constants

Current transformations are at the expression level

- Constant folding and strength reduction

  Example: if at specification time `a[i]==4`, then
  `` `{ x = y * $a[i] } `` compiles to `sll $rx, $ry, 2`

- Dead code elimination (checks on branch conditions)

```
if ($x)
  a = b;
else
  a = c;
```

$\xrightarrow{\$x=1}$

```
a = b;
```

More aggressive strategies are possible but costly

Performance Results

Speed-up:

$$\frac{\text{Run time of static code}}{\text{Run time of dynamic code}}$$

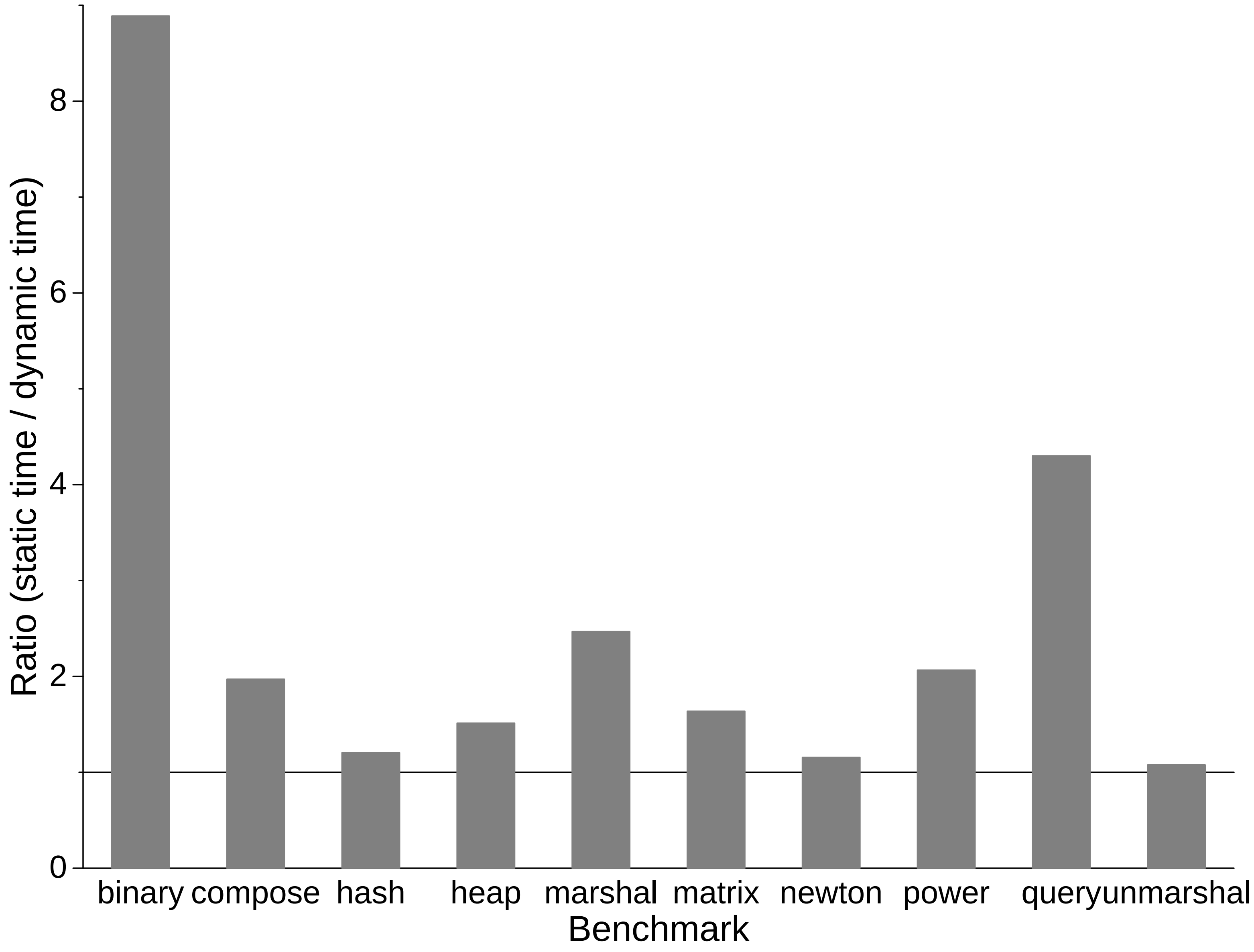
Does *not* include cost of creating dynamic code

Break-even point:

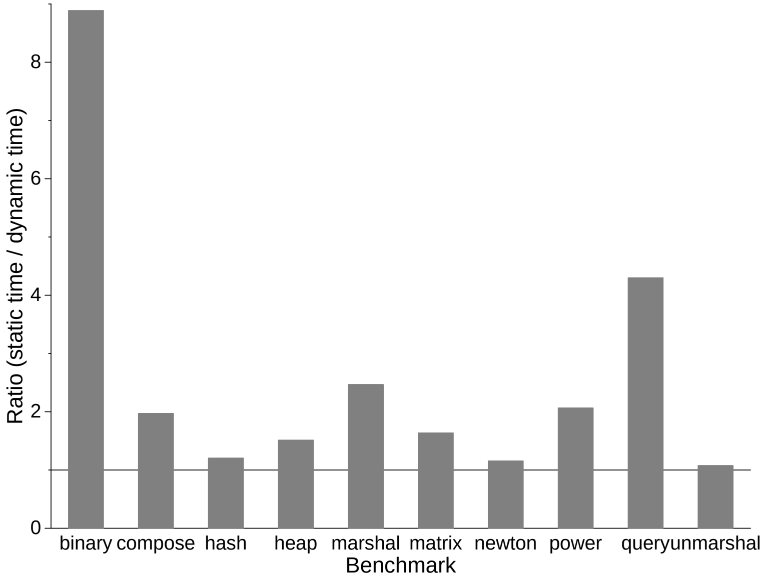
$$\frac{\text{Run time of dynamic code generation}}{(\text{Run time of static code}) - (\text{Run time of dynamic code})}$$

When is the cost of creating dynamic code recouped?

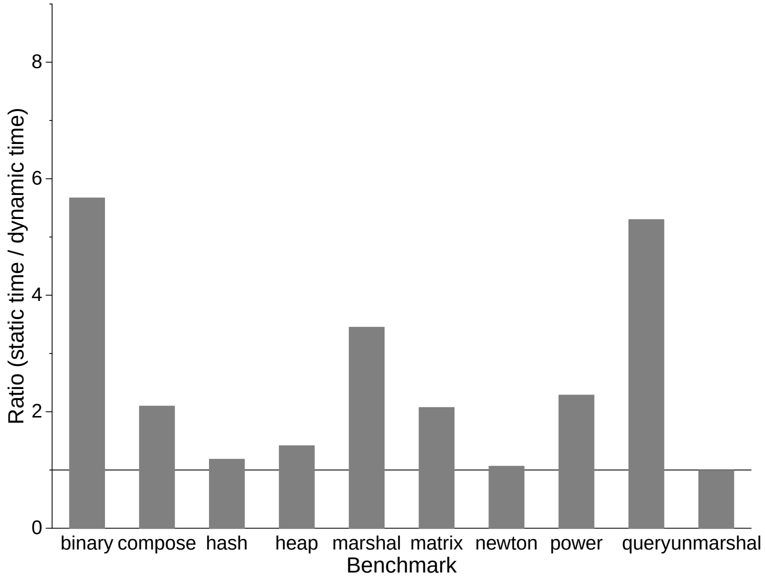
Results: Speed-Up on MIPS R4400



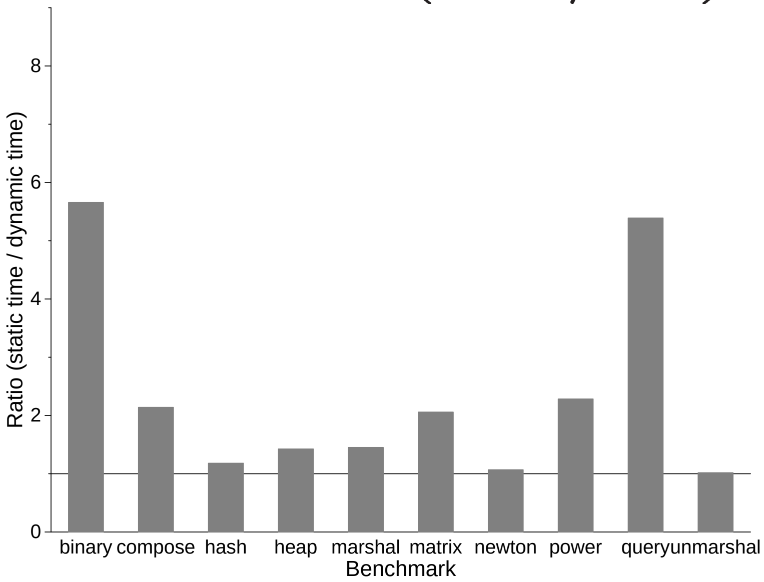
Results: Speed-Ups



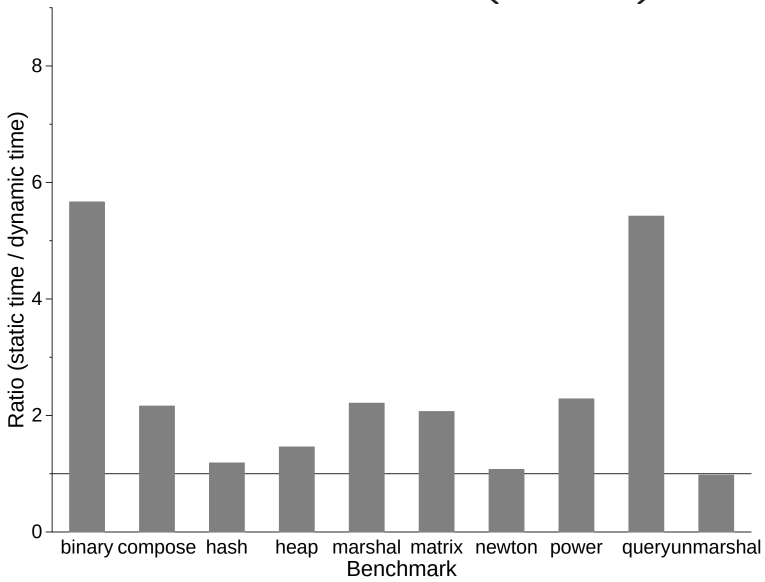
MIPS R4400 (5000/260)



MIPS R2000 (2100)

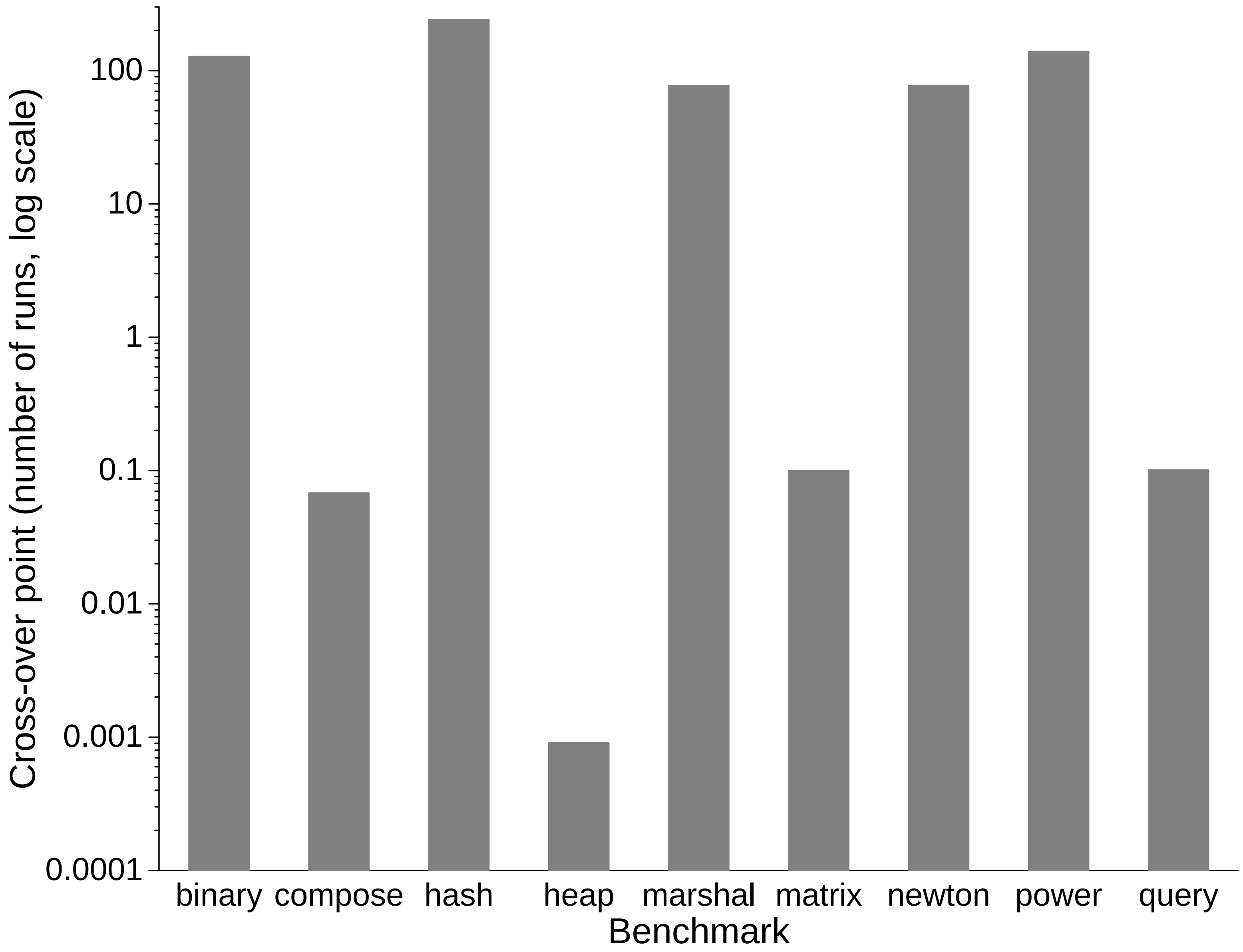


MIPS R3000 (5000/133)

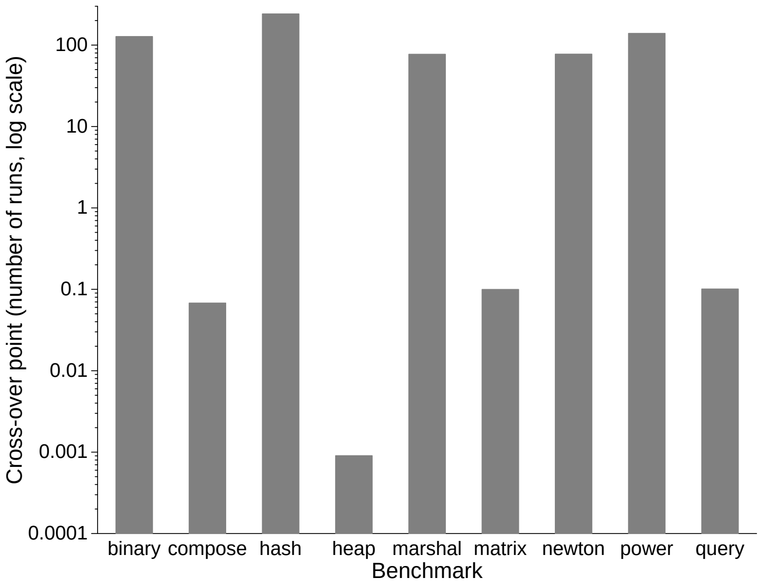


MIPS R3000 (5000/200)

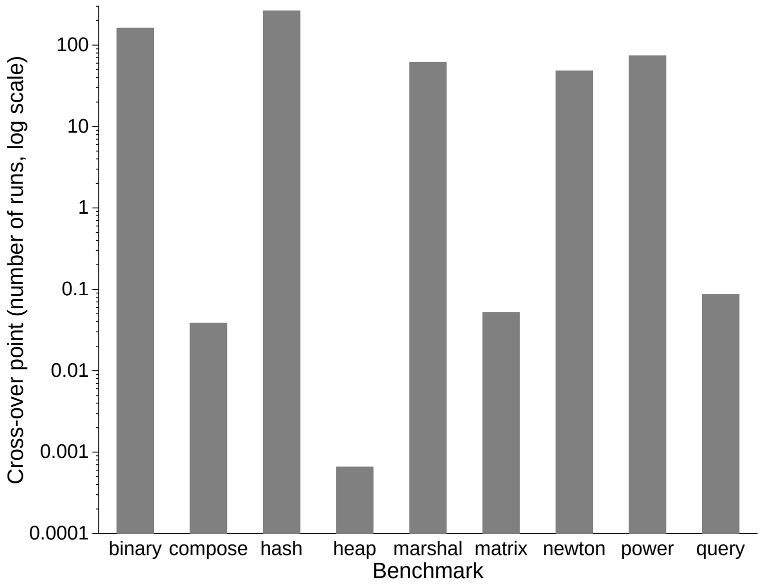
Results: Break-Even Point on MIPS R4400



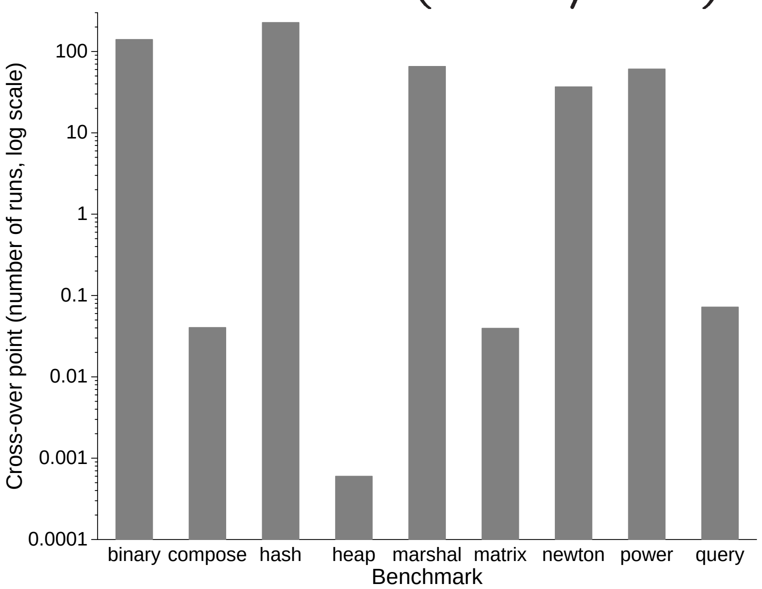
Results: Break-Even Points



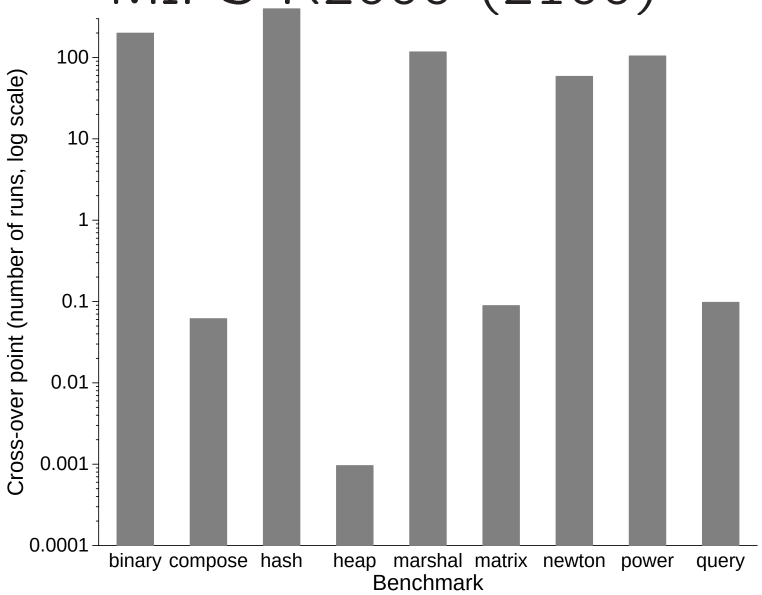
MIPS R4400 (5000/260)



MIPS R2000 (2100)



MIPS R3000 (5000/133)



MIPS R3000 (5000/200)

Summary of Performance Results

DCG provides substantial benefits (up to an order of magnitude)

Cost of DCG is low and easily recoupable

Results hold across:

- generations of a processor family
- two different processor families (MIPS and SPARC)

Future Work: Improving Performance

Better dynamic register allocation and run-time constant optimizations

Fast dynamic peephole optimizations

Higher-quality compilation of CGFs

Future Work: Extending the ‘C System

Language extensions

- Expression statements: stronger type-checking of dynamically generated statements
- Support for partial evaluation of functions: quickly adapt large pieces of code to exploit dynamic code generation

Programmer support

- Debugging
- Profiling

Summary

First design and real implementation of a high-level language for efficient and explicit dynamic code generation

Significant performance improvements across several benchmarks on 2 major architectures

A release of the ‘C compiler is being made available

gcc versus lcc

Using a better compiler:

- Improves quality of dynamic code
- Increases speed of dynamic code generation

Optimizations based on DCG cannot be performed by a traditional compiler, no matter how good