Automatic Program Specialization for Interactive Media

Scott Draves

Context and Thesis

- Interactive sample-level media programs
  - like Photoshop
- Flexible *or* fast
  - trade-off
- Run-time code generation
- Specialization
  - Compiler generation
- Treat interface as language

Outline

- RGB to mono example
  - Motivation
- Specialization
  - Cyclic integers, sharing, late normalization
- Implementation
  - Compiler generation and Nitrous
- Benchmarks
- Conclusion

RGB to Mono

```
fun rgb_to_mono f r g b m =
  if (r s_endp) then () else
    ((m s_put) (f (r s_get) (g s_get)
                  (b s_get));
     rgb_to_mono f (r s_next) (g s_next)
                   (b s_next) (m s_next))
```

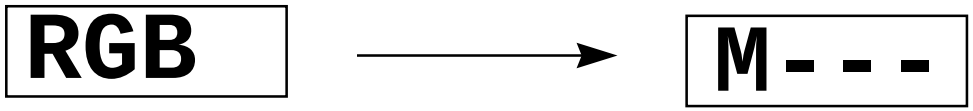
object-oriented
streams

easy to write,
modular,
slow to run

Specialized One Way

rgb0 points to beginning, rgb1 to end

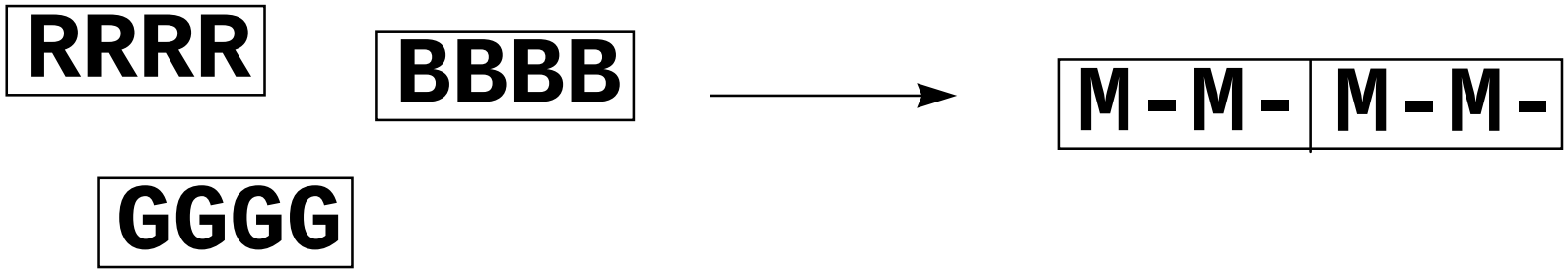
```
fun rgb_to_mono_1
            (rgb0 rgb1 m0 m1) =
  if (rgb0 = rgb1) then () else
  let w = load_word rgb0
  (store_word m0 ( 9 * w&0xff +
                  20 * (w>>8)&0xff +
                   3 * (w>>16)&0xff);
   rgb_to_mono_1 (rgb0+1 rgb1 m0+1 m1))
```



assumptions

Specialized Another Way

```
fun rgb_to_mono_2 (r0 r1 b0 b1 g0 g1 m0 m1) =
  if (r0 = r1) then () else
  let rw = load_word(r0)
      gw = load_word(g0)
      bw = load_word(b0)
   (store_word m0  (rw&0xff + gw&0xff + bw&0xff) |
      ((rw>>8)&0xff + (gw>>8)&0xff + (bw>>8)&0xff)<<16;
    store_word (m0+1)
      ((rw>>16)&0xff + (gw>>16)&0xff + (bw>>16)&0xff) |
      ((rw>>24)&0xff + (gw>>24)&0xff + (bw>>24)&0xff)<<16;
    rgb_to_mono_2 (r0+1 r1 b0+1 b1 g0+1 g1 m0+2 m1))
```



Yet Another

what about 12-bit channels?

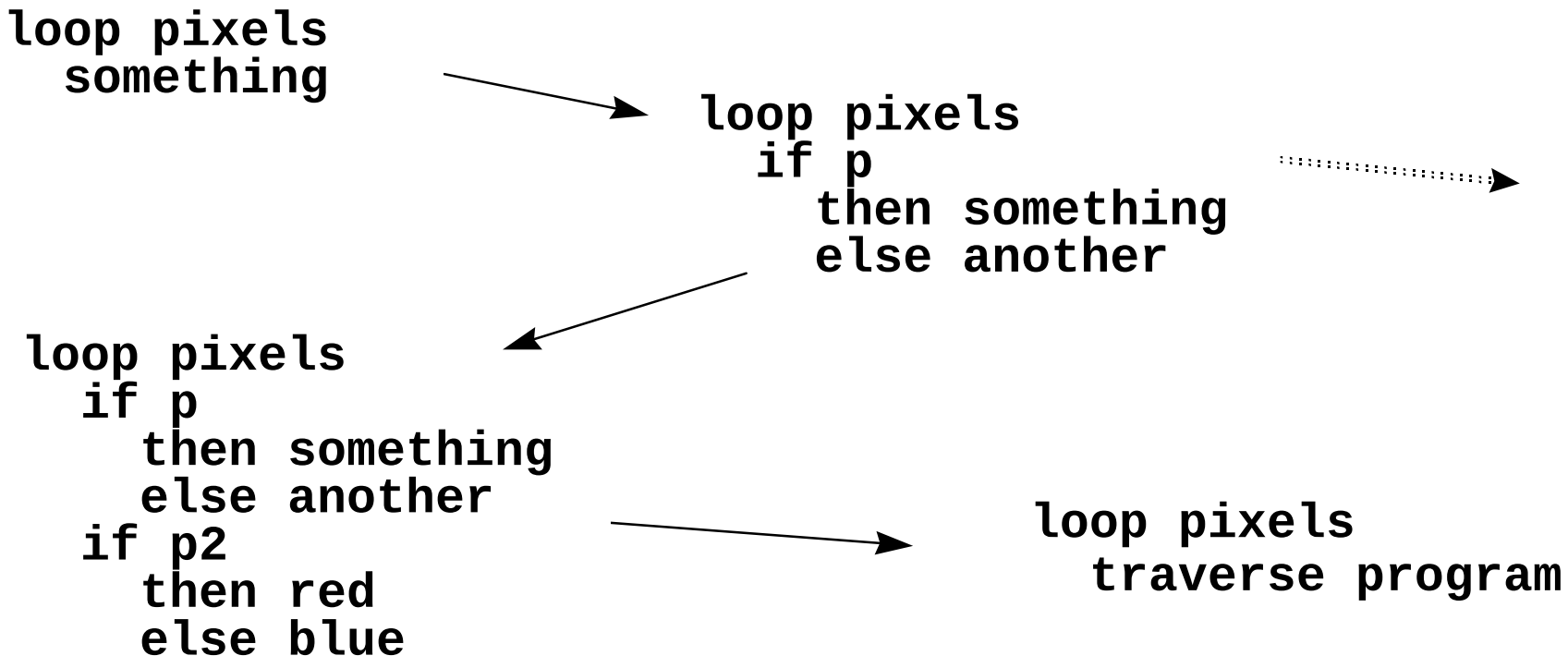
what about run-length encoding?

what about other combination functions?

The Problem

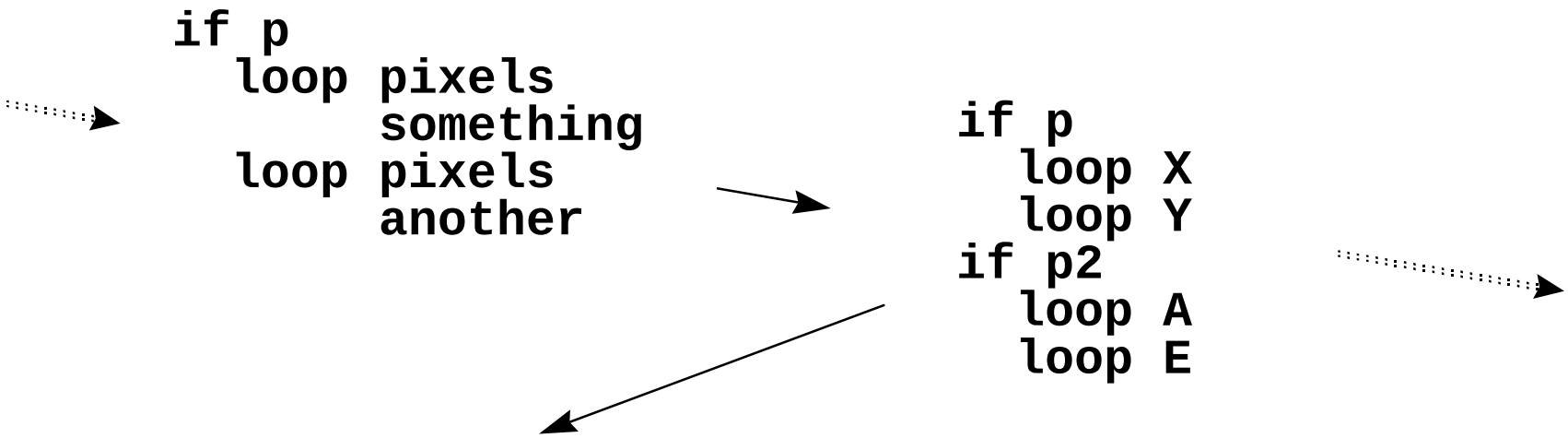
- These routines are large
  - Self-consistency
- Generality/performance tradeoff
  - Lots of operators, lots of data
- Three approaches
  - Interpreter
  - Buffering/batches
  - Special cases

Interpreter

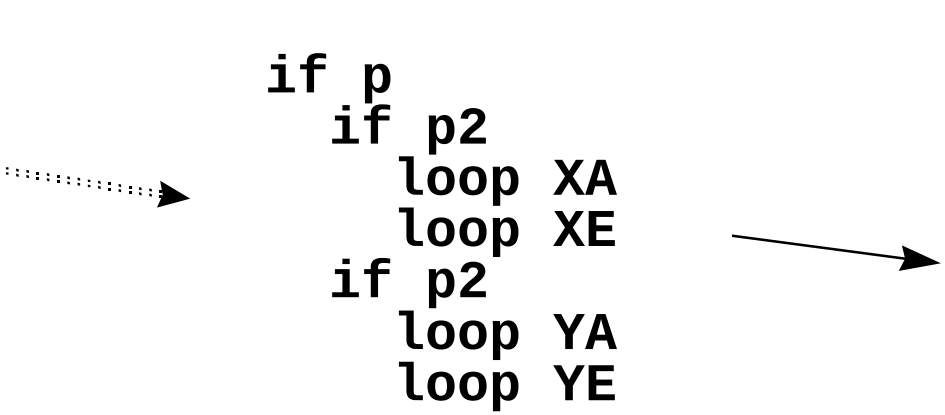


sequence of generalizations

Buffers or Batching



Special Cases



```
if p
  if p2
    loop XA
    loop XE
  if p2
    loop YA
    loop YE
```



```
if p
  if p2
    if p3
      loop XAJ
      loop XAK
    if p3
      loop XEJ
      loop XEK
  if p2
    if p3
      loop YAJ
      loop YAK
    if p3
      loop YEJ
      loop YEK
```

optimal performance,
exponential blow-up

The Solution

- Write programs that write programs
  - Macros, lex/yacc, templates, …
- Run-time code generation
  - Laziness avoids blow-up
- Synthesis, Blit
- Common LISP
  - compile/eval quasi-quote/unquote

What can my system do?

- Transform slow to fast via assumptions
- Lazy vector library
  - Construct, transform, reduce
- Smart unrolling
  - Alignment against words of memory
  - Constant shift offsets
- Cache-hits to registers
- Irregular data

Bit-addressing

Outline

- RGB to mono example
  - Motivation
- Specialization
  - Cyclic integers, sharing, late normalization
- Implementation
  - Compiler generation and Nitrous
- Benchmarks
- Conclusion

Specialization

- Partial evaluation
- One interface to metaprogramming
  - **`((spec f x) y) = (f x y)`**
  - Constant prop, inlining, and memoization

```
fun power e b =
  if (e = 0) then 1 else
    b * (power e-1 b)
```

```
fun power5 b =
  b * b * b
    * b * b
```

Language and Domains

$$t \in \mathsf{Type} ::= \texttt{int} \mid \mathsf{Type} \texttt{ -> } \mathsf{Type}$$

$$\begin{aligned} d, e \in \mathsf{Exp} ::= {} & \mathsf{Atom} \mid \mathsf{Var} \\ & \mid \texttt{lambda } \mathsf{Var} \texttt{:} \mathsf{Type} \texttt{.} \mathsf{Exp} \\ & \mid \mathsf{Exp}\ \mathsf{Exp} \\ & \mid \texttt{lift } \mathsf{Exp} \mid \mathsf{Exp} \diamond \mathsf{Exp} \mid \ldots \end{aligned}$$

$$\begin{aligned} s, k &\in \mathsf{Atom} = \mathbb{Z} \\ f_t &\in \mathsf{F} = (\mathsf{M} \to \mathsf{M}) \times \mathsf{Type} \\ m &\in \mathsf{M} = \mathsf{Exp} + \mathsf{Atom}_\perp + \mathsf{F} \\ \rho &\in \mathsf{Env} = \mathsf{Var} \to \mathsf{M} \end{aligned}$$

Evaluation

$$\mathcal{S} : \mathsf{Exp} \to \mathsf{Env} \to \mathsf{M}$$

$$\mathcal{S}\ \boxed{e_0 \diamond e_1}\ \rho = \mathrm{match}\ (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(s_0, s_1) \to s_0 \diamond s_1$$
$$(d_0, d_1) \to \boxed{d_0 \diamond d_1}$$

$$\mathcal{S}\ \boxed{v}\ \rho = \rho\ v$$

$$\mathcal{S}\ \boxed{k}\ \rho = k$$

$$\mathcal{S}\ \boxed{\texttt{lambda}\ v{:}t.e}\ \rho = (\lambda v'.\mathcal{S}\ e\ ([v \mapsto v']\rho))_t$$

$$\mathcal{S}\ \boxed{e_0\ e_1}\ \rho = \mathrm{match}\ (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(f, m) \to f\ m$$
$$(d_0, d_1) \to \boxed{d_0\ d_1}$$

$$\mathcal{S}\ \boxed{\texttt{lift}\ e}\ \rho = D\ (\mathcal{S}\ e\ \rho)$$

boxes denote syntax

unsafe

metavariables in patterns must match domain

Reification and Reflection

$$D : \mathsf{M} \to \mathsf{Exp}$$

$$\begin{aligned}
D\ d &= d \\
D\ s &= \boxed{s} \\
D\ f_t &= \text{let } v' = \text{gensym} \\
&\qquad\quad e' = D(f(Utv')) \\
&\qquad \text{in } \boxed{\texttt{lambda } v'\texttt{:}t\texttt{.}e'}
\end{aligned}$$

Danvy96

$$U : \mathsf{Type} \to \mathsf{Exp} \to \mathsf{M}$$

$$\begin{aligned}
U\ \boxed{t\texttt{->}t'}\ e_0 &= (\lambda\ v\ .\ \text{let } e_1 = D\ v \text{ in } U\ t\ \boxed{e_0\ e_1})_t \\
U\ \texttt{int}\ \boxed{e} &= e
\end{aligned}$$

Example

```
fun sum (from, to, size, stride) r =
  if from = to then r else
  sum ((from + stride), to, size, stride)
      (r + (load_sample from size))

fun load_sample p b =
   ((1 << b) - 1) &
   ((load_word (p / 32)) >> (p % 32))
```

overlap not handled here

Want vs Get

```
fun sum_0088 fromq toq r =
   if fromq = toq then r else
     let val w = load_word fromq
     in sum_0088 (fromq + 1) toq
           (r + (w & 255) + ((w >> 8) & 255) +
            ((w >> 16) & 255) + (w >> 24))
     end
```

```
fun sum_88 from to r =
  if from = to then r else
  sum_88 (from + 8) to
      (r + (255 & ((load_word (from / 32))
                   >> (from % 32))))
```

Cyclic Integers

a kind of partially-static integer

$$\langle b\ q\ r\rangle \in \mathsf{Cyclic} = \mathbb{Z} \times \mathsf{Exp} \times \mathbb{Z}$$

$$m \in \mathsf{M} = \ldots + \mathsf{Cyclic}$$

$$D\ \langle b\ q\ r\rangle = \boxed{b{*}q{+}r}$$

breaks int into dynamic quotient, static base, and static remainder

normal form

Plus and Times

$$\begin{aligned}\mathcal{S}\ \boxed{e_0{+}e_1}\ \rho = \text{match}\ &(\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)\\ &(\langle b\ q\ r\rangle, s) \to \text{let}\ r' = (r+s)\ \%\ b\\ &\qquad\qquad q' = (r+s)\ /\ b\\ &\qquad\qquad \text{in}\ \langle b\ \boxed{q+q'}\ r'\rangle\end{aligned}$$

$$\begin{aligned}\mathcal{S}\ \boxed{e_0{*}e_1}\ \rho = \text{match}\ &(\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)\\ &(\langle b\ q\ r\rangle, s) \to \text{if}\ s > 0\ \text{then}\ \langle sb\ q\ sr\rangle\\ &\qquad\qquad \text{else if}\ s < 0\ \text{then error}\\ &\qquad\qquad \text{else}\ 0\end{aligned}$$

excess additions

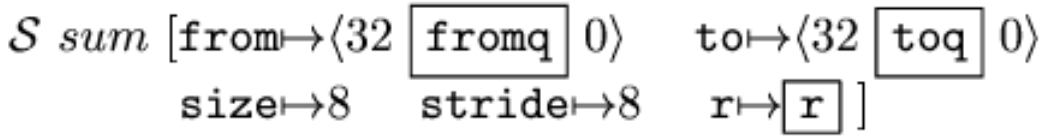
Zero-test and Division

$$\mathcal{S}\ \boxed{\texttt{zero?}\quad e}\ \rho = \text{match}\ \mathcal{S}\ e\ \rho$$
$$\langle b\ q\ r\rangle \rightarrow \text{if}\ (0 = (r\ \%\ b))$$
$$\text{then let}\ t = (r\ /\ b)$$
$$\text{in}\ \boxed{\texttt{zero?}\quad q\texttt{+}t}$$
$$\text{else false}$$

$$\mathcal{S}\ \boxed{e_0 / e_1}\ \rho = \text{match}\ (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle, s) \rightarrow \text{if}\ (0 = (b\ \%\ s))$$
$$\text{then}\ \langle (b\ /\ s)\ q\ (r\ /\ s)\rangle$$

$$\mathcal{S}\ \boxed{e_0 \% e_1}\ \rho = \text{match}\ (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle, s) \rightarrow \text{if}\ b = s\ \text{then}\ r$$

Residual Code




```
fun sum_0088 fromq toq r =
   if fromq = toq then r else
     sum_0088 (fromq + 1) toq
      (r + (((load_word fromq) >> 0) & 255) +
           (((load_word fromq) >> 8) & 255) +
           (((load_word fromq) >> 16) & 255) +
           (((load_word fromq) >> 24) & 255))
```

Example

```
fun sum (from, to, size, stride) r =
  if from = to then r else
  sum ((from + stride), to, size, stride)
      (r + (load_sample from size))

fun load_sample p b =
   ((1 << b) - 1) &
   ((load_word (p / 32)) >> (p % 32))
```

Cache and Sharing

- Repeated loads, duplicate arguments
- Common subexpression elimination
- Stores are harder
  - aliasing
  - loads for partial stores
- Software cache
- Conservative early equality
  - extend PE with sharing

negative alias info needed

Late Normalization

- Problems with plus and times
  - Residual addition confuses sharing
- Normalize only at memoization points
  - Maintain sharing
  - Trace all static state

$$\mathcal{S}\ \boxed{e_0 + e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho,\ \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle,\ s) \rightarrow \text{let } r' = (r+s)\ \%\ b$$
$$q' = (r+s)\ /\ b$$
$$\text{in } \langle b\ \boxed{q+q'}\ r'\rangle$$

$$\mathcal{S}\ \boxed{e_0 * e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho,\ \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle,\ s) \rightarrow \text{if } s > 0 \text{ then } \langle sb\ q\ sr\rangle$$
$$\text{else if } s < 0 \text{ then error}$$
$$\text{else } 0$$

$$\mathcal{S}\ \boxed{e_0 + e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle,\ s) \rightarrow \langle b\ q\ r+s\ \rangle$$

$$\mathcal{S}\ \boxed{e_0 * e_1}\ \rho = \text{match } (\mathcal{S}\ e_0\ \rho, \mathcal{S}\ e_1\ \rho)$$
$$(\langle b\ q\ r\rangle,\ s) \rightarrow \text{if } s > 0 \text{ then } \langle sb\ q\ sr\rangle$$
$$\text{else if } s < 0 \text{ then } \langle \text{-}sb\ \boxed{-q}\ sr\rangle$$
$$\text{else } 0$$

Outline

- RGB to mono example
- Specialization
  - Cyclic integers, sharing, late normalization
- Implementation
  - Compiler generation and Nitrous
- Benchmarks
- Conclusion

Compiler Generation

- Another stage
  - **`(((cogen f) x) y) = (f x y)`**
  - Avoids redundant syntax traversal
- Binding Times
  - *When* will we know a value?
- Temporal Types

handles Turing-complete languages

cogen = (spec spec spec)

Nitrous Implementation

- Compiler generator *PE96*
- Intermediate language
  - Three-address code (CPS-CPS)
- Multiple levels (feedback)
- Supports Bit-addressing
  - Partially-static integers
  - Sharing, late normalization

Other Implementations

- Simple
  - no dynamic calls
  - no dynamic conditionals
    - except loop primitive
  - no late normalization

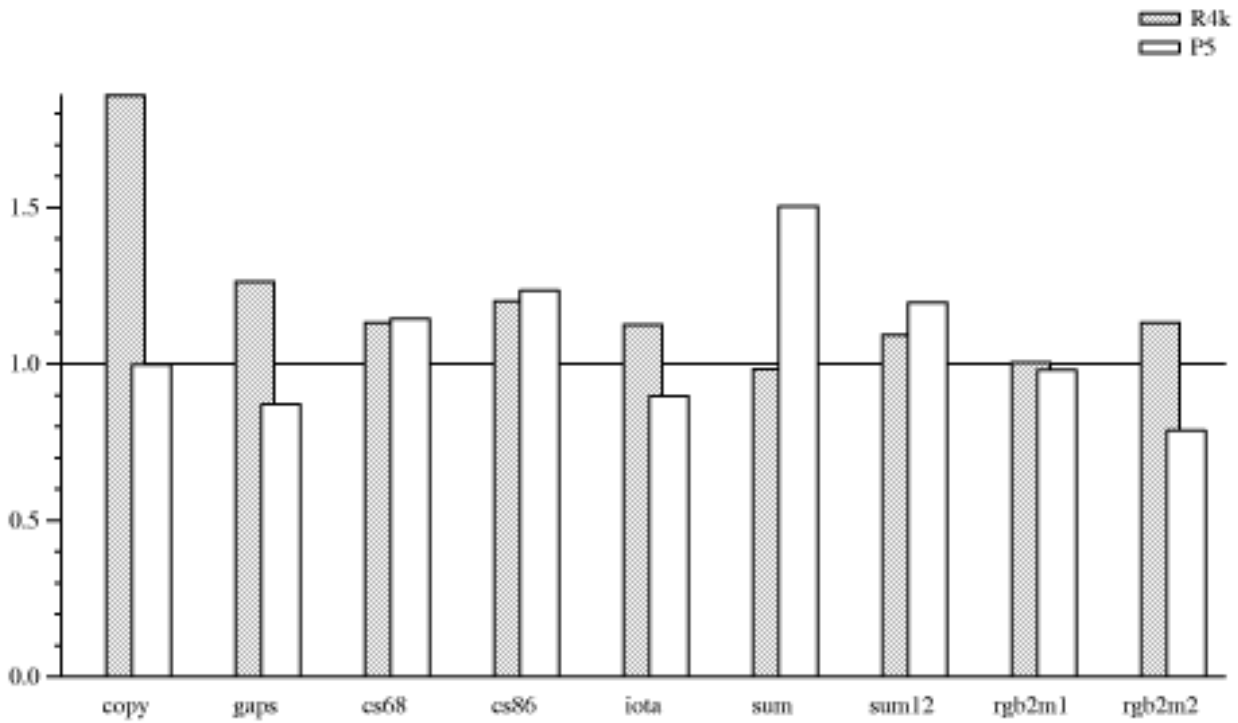
ICFP97

- Similix
  - manual binding time improvement
  - no sharing

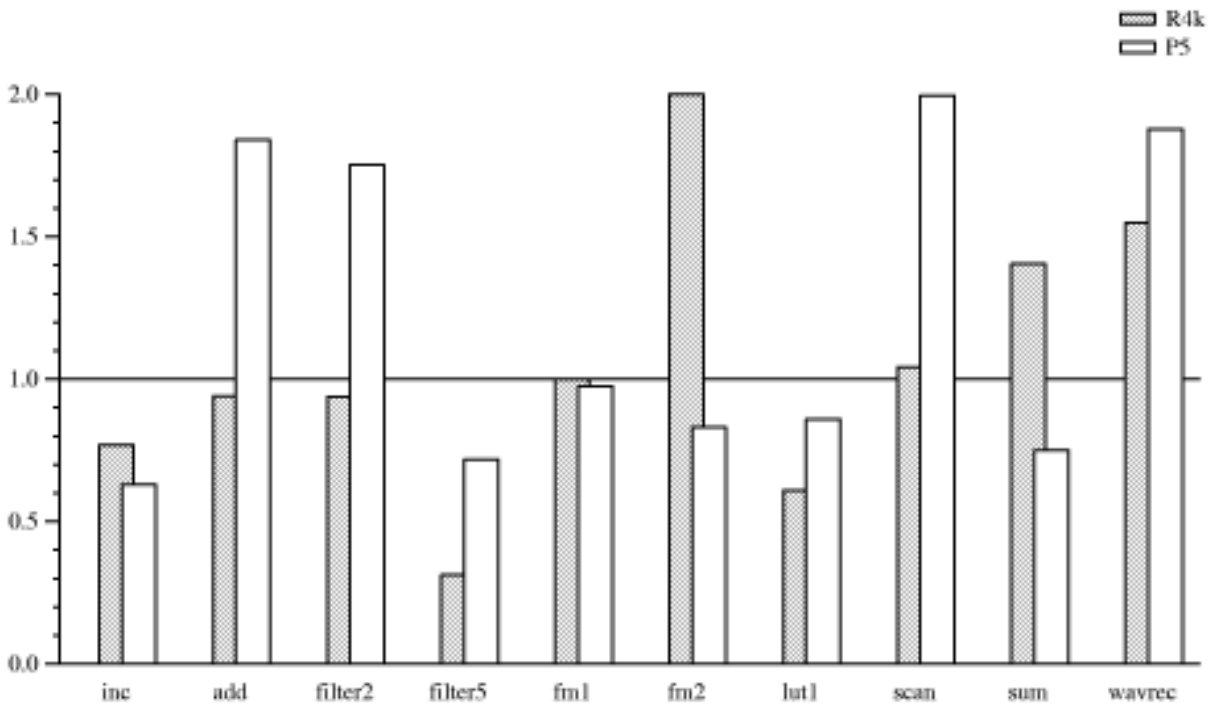
Video Benchmarks

compile to C,
then with GCC

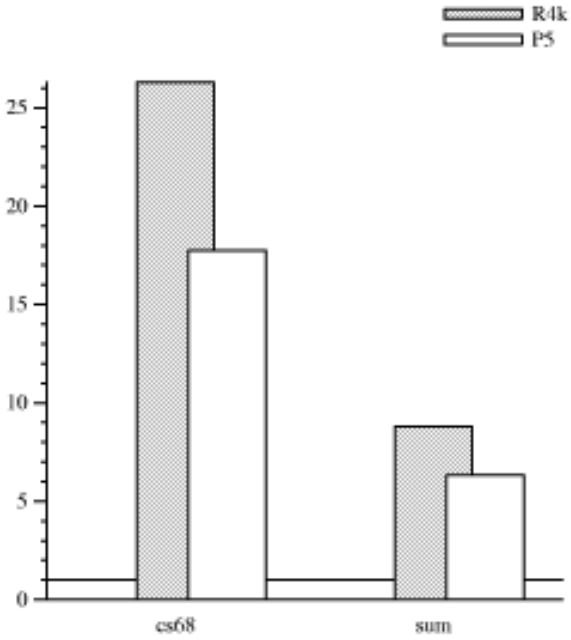
ratio of times,
automatic to manual

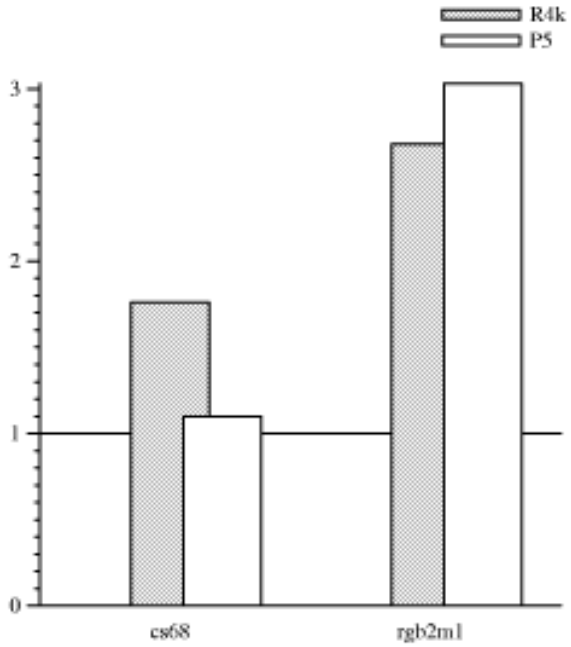


Audio Benchmarks



Interpretation and Buffering





Related Work

- Bit-fields in C, VHDL
- Ptolemy, Lustre, NESL
- Filter Fusion, Horus
- Tick-C
- Tempo, Fabius

Future Applications

- Decode and Blit
  - Brush language
- Audio Synthesis
- 3D Graphics
  - Particle Systems
  - Image-based Rendering
- Genetic Programming
  - Cellular Automata

FPGAs

Lightweight Languages

Contributions

- Cogen for run-time code generation
- Extend Partial Evaluation with
  - Cyclic integers
  - Sharing
- Bit-addressing