Implementing Bit-addressing with Specialization

Scott Draves[*]
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA

Abstract

General media-processing programs are easily expressed with bit-addressing and variable-sized bit-fields. But the natural implementation of bit-addressing relies on dynamic shift offsets and repeated loads, resulting in slow execution. If the code is specialized to the alignment of the data against word boundaries, the offsets become static and many repeated loads can be removed. We show how introducing modular arithmetic into an automatic compiler generator enables the transformation of a program that uses bit-addressing into a synthesizer of fast specialized programs.

In partial-evaluation jargon we say: modular arithmetic is supported by extending the binding time lattice used by the static analysis in a polyvariant compiler generator. The new binding time `Cyclic` functions like a partially static integer.

A software cache combined with a fast, optimistic sharing analysis built into the compilers eliminates repeated loads and stores. The utility of the transformation is demonstrated with a collection of examples and benchmark data. The examples include vector arithmetic, audio synthesis, image processing, and a base-64 codec.

1 Introduction

Media such as audio, images, and video are increasingly common in computer systems. Such data are represented by large arrays of small integers known as samples. Rather than wasting bits, samples are packed into memory. Figure 1 illustrates three examples: monaural sound stored as an array of 16-bit values, a grayscale image stored as an array of 8-bit values, and a color image stored as interleaved 8-bit arrays of red, green, and blue samples. Such arrays are called *signals*.

Say we specify a signal's representation with four integers: `from` and `to` are bit addresses; `size` and `stride` are numbers of bits. We use 'little-endian' addressing so the least significant bit of each word (LSB) has the least address of the bits in that word.

[*]This research was sponsored in part by the Defense Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract No. F19628-95-C-0050. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

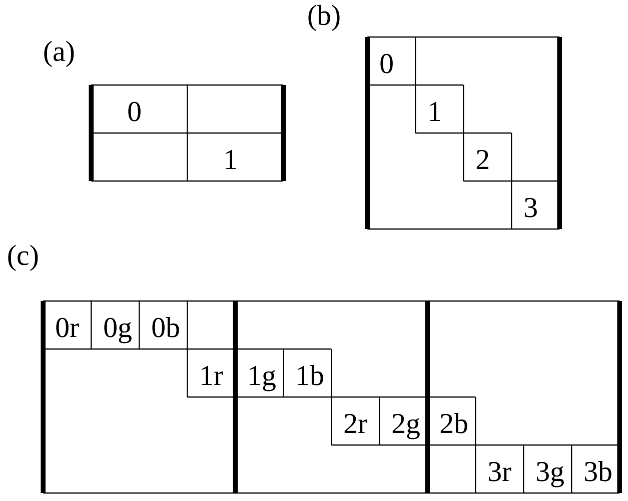

Figure 1: Layout of (a) 16-bit monaural sound, (b) an 8-bit grayscale image, and (c) a 24-bit color image. The heavy lines indicate 32-bit word boundaries.

```
type signal = int * int * int * int
          (* from  to    size  stride *)
```

Figure 2 gives the code to sum the elements of a signal. This and other examples use ML syntax extended with infix bit operations as found in the C programming language (`<< >> & |`). The `load_word` primitive accesses a memory location. This paper assumes 32-bit words, but any other size could just as easily be substituted even at run-time. The integer division (`/`) rounds toward minus infinity; integer remainder (`%`) has positive base and result. To simplify this presentation, `load_sample` does not handle samples that cross word boundaries.

If we fix the layout by assuming `stride = size = 8` and `(from % 32) = (to % 32) = 0` then the implementation in Figure 3 computes the same value, but runs more than five times faster (see Figure 21). There are several reasons: the loop is unrolled four times, resulting in fewer conditionals and more instruction level parallelism; the shift offsets and masks are known statically, allowing immediate-mode instruction selection; the division and remainder computations in `load_sample` are avoided; redundant loads are eliminated.

Different assumptions result in different code. For example, sequential 12-bit samples result in unrolling 8=lcm(12,32)/12 times

```
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))
```

Figure 2: Summing a signal using bit addressing.

```
fun sum_0088 from to r =
  if from = to then r else
  let val v = load_word from
  in sum_0088 (from + 1) to
    (r + (v & 255) + ((v >> 8) & 255) +
    ((v >> 16) & 255)+ ((v >> 24) & 255))
  end
```

Figure 3: Summing a signal assuming packed, aligned 8-bit samples as in Figure 1(b).

so that three whole words are loaded each iteration (see Figure 4). Handling samples that cross word boundaries requires adding a conditional to `load_sample` that loads an additional word, then does a shift-mask-shift-or sequence of operations.

As such, the programmer is faced with a familiar trade-off: write one slow, easy-to-read, general-purpose routine; or write many fast special cases. We pursue an alternative: write general-purpose code and automatically derive fast special cases. The techniques presented here are designed to be fast enough to generate special cases lazily at run-time, thus providing an interface to run-time code generation (RTCG). It is not strictly necessary that specialization occur at run-time, but because the number of special cases is exponential in the number of static arguments, code space quickly becomes a problem if the specialization is all done at compile time, as with macro and C++ template expansion.

As a concrete example consider the screen position of a window. The horizontal coordinate affects the alignment of its pixels against the words of memory, so special-purpose graphics operations may be created each time a window is opened or moved. As another example, consider an interactive audio designer. A particular 'voice' is defined by a small program; Figure 5 is a typical example of an FM synthesizer. Most systems allow the user to pick from several predefined voices and adjust their scalar parameters. With RTCG, the user may define voices with their own wiring diagrams.

Other interfaces to run-time code generation have been explored in a variety of places: there have been manual systems such as Common Lisp [Steele90] with `eval`, macros with backquote/comma syntax, and slow code generation. Fast manual systems such as Synthesis [Massalin92] and the Blit terminal [PiLoRei85] confirmed the performance benefits of RTCG in operating systems and bit-mapped graphics, respectively. 'C [EnHsKa95]

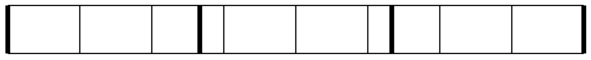

Figure 4: 12-bit signal against 32-bit words shown with abbreviated vertical axis.

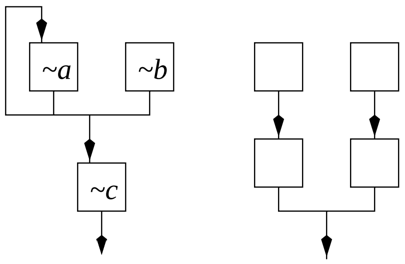

Figure 5: Two voices. On the left is a simple 2-in-1 FM synthesizer. Oscillators *a* and *b* sum to modulate *c* as well as feeding back into *a*. On the right is another possibility.

adds a Lisp-style interface to RTCG to the C programming language. Fabius [LeLe96] uses fast automatic specialization for run-time code generation of a subset of ML, but cannot handle bit-addressing. Tempo [CoHoNoNoVo96] attempts to automate the kind of RTCG used by Synthesis. Self takes an automatic but less general approach to run-time code generation [ChaUng91], as do recent just-in-time (JIT) implementations of Java [GoJoSte96].

Past work in bit-level processing has not emphasized implementation on word-machines. VHDL [IEEE91] allows this level of specification, but lacks an efficient compiler. Synchronous real-time languages like Signal [GuBoGaMa91] support programming with streams, but not at the bit level.

This paper shows how to implement bit-addressing with a partial evaluator.

Section 2 presents a polyvariant, direct-style specializer and briefly describes how to derive a compiler generator from it. Section 3 extends the specializer with cyclic integers, resulting in an analysis similar to [Granger89]. Section 3.2 shows how irregular (data-dependent) layouts are handled. Section 4 shows how extending of partial evaluation allows fast elimination of redundant loads and stores. Section 5 describes two implementations of these ideas; Section 6 presents example source programs and compares the performance of the generated code with hand-written C programs.

2 Specialization

We begin our discussion of specialization with a definition, then we introduce our notation and give a simple polyvariant specializer for a λ-language. Section 2.1 discusses efficient implementation via compiler generaton and introduces the concept of binding times. Section 2 is generally a review of partial evaluation practice; [JoGoSe93] is the standard text of the field and may be considered a reference of first resort if you can find it. [WeiCoRuSe91] is a more widely available description of an advanced on-line specializer. The system described here is a polyvariant version of type-directed partial evaluation [Danvy96], much like [Sheard96].

A specializer mix satisfies the following equation where italic names denote program texts and Quine quotes $[\![\cdot]\!]$ denote ordinary evaluation:

$$[\![f]\!]\ x\ y = [\![[\![mix]\!]\ f\quad x]\!]\ y$$

There are many ways to implement $[\![mix]\!]$; a simple curry function suffices. Our intension is that $[\![mix]\!]$ will do as much work of f as is possible knowing only its first argument and return a *residual* program that finishes the computation. Because we expect to use this residual function many times, this gives us a way of ‘factoring’ or ‘staging’ computations as in [JoSche86].

Figure 6 gives the grammar of our object language, and defines some domains and their metavariables. The language is the λ-calculus extended with explicit types on abstractions, constants, primitives, a conditional, and a lift annotation.

We say the lift is an ‘annotation’ because in the ‘ordinary’ semantics of the λ-calculus, lift has no meaning; it becomes the identity function. The ordinary semantics can be useful for debugging.

Figure 7 gives a specializer $\mathcal{S}$. The notation $[v \mapsto k]\rho$ denotes updating the environment ρ with a binding from the variable v to the value k; $\diamond$ denotes a generic, ‘black box’ binary primitive operation; $\boxed{\text{frames}}$ mark manipulation of the terms of the λ-language’s syntax (like Lisp’s backquote); `match` e $pat \to e$ ... denotes pattern matching where the metavariables only match the appropriate domain.

Figure 8 defines the reification and reflection functions R and L. They operate as coercions between code and data; understanding them is not essential to this work.

$\mathcal{S}$ is a *partial-evaluation* function; it assigns a meaning from M to a source text with environment. The difference from an ordinary semantics is that M contains Exp, whose members represent computations dependent on unknown values, i.e. are residual code. We say the specializer *emits* residual code.

We say $\mathcal{S}$ is *polyvariant* because a given piece of syntax may be both executed by $\mathcal{S}$ and emitted as residual. This happens to `f` in this example:

```
let fun f x = x + 1
    fun g s d = (f s) + (f d)
in (g 1 (lift 1))
end
```

Creating general code and a special case of the same source text corresponds to the standard ‘fast-path’ optimization technique.

Note that the `if` clause requires that when a conditional has dynamic predicate, then both arms are also dynamic.

$\mathcal{S}$ is similar to the λ-mix of [GoJo91], but because λ-mix is monovariant, it uses a two-level input language where source `lambda` terms have been labeled either for execution or immediate residualization. $\mathcal{S}$ reserves judgement until the λ is applied; $\mathcal{S}$ depends on lift annotations to emit functions.

Note that many cases are missing from $\mathcal{S}$. We assume that all input programs are type-correct and lift annotations appear as necessary. Placement of the lifts is crucial to successful staging: too many lifts and $\mathcal{S}$ degenerates into the curry function; too few and $\mathcal{S}$ fails to terminate. Typically *binding time analysis* (BTA) is combined with programmer annotations to insert the lifts. For example, if ρ=[a$\to$6 b$\to\boxed{\texttt{c}}$] then $\mathcal{S}$ requires `((lift a)`$\diamond$`b)` rather than `(a`$\diamond$`b)`. This kind of lift is obvious, and is easily handled by BTA. As an example of the kind of lift that cannot be easily automated, consider the following tail-recursive function:

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

where `e` is in Exp and `b` is in Val. Unless we manually lift `r` to dynamic, $\mathcal{S}$ will diverge.

Monovariant BTA is well-understood and can be efficiently implemented with type-inference [Henglein91]. Polyvariant BTA is usually implemented with abstract interpretation [Consel93].

$[\![mix]\!]$ can be defined with $\mathcal{S}$ like this:

```
t ∈ Type ::= atom | Type -> Type

d, e ∈ Exp ::= Val | Var | Exp Exp
           | if Exp Exp Exp
           | lambda Var:Type . Exp
           | lift Exp| Exp ◇ Exp | ...

s, k ∈ Val = Bool + ℤ
$f_t$ ∈ F = (M → M) × Type
m ∈ M = Exp + Val + F
ρ ∈ Env = Var → M
```

Figure 6: The λ-language, domains and metavariables. $\diamond$ is a primitive; Var is the set of variables.

```
𝒮 : Exp × Env → M

𝒮 $\boxed{e_0 \diamond e_1}$ ρ = match (𝒮 $e_0$ ρ, 𝒮 $e_1$ ρ)
                  ($s_0$, $s_1$) → $s_0$ ◇ $s_1$
                  ($d_0$, $d_1$) → $\boxed{d_0 \diamond d_1}$
𝒮 $\boxed{v}$ ρ = ρ v
𝒮 $\boxed{k}$ ρ = k
𝒮 $\boxed{\texttt{lambda}\ v\texttt{:}t\texttt{.}e}$ ρ =
             $(\lambda v' . \mathcal{S}\ e\ ([v \mapsto v']\rho))_t$
𝒮 $\boxed{e_0\ e_1}$ ρ = match (𝒮 $e_0$ ρ, 𝒮 $e_1$ ρ)
                 (f, m) → f m
                 ($d_0$, $d_1$) → $\boxed{d_0\ d_1}$
𝒮 $\boxed{\texttt{lift}\ e}$ ρ = R (𝒮 e ρ)
𝒮 $\boxed{\texttt{if}\ e_0\ e_1\ e_2}$ ρ =
  match (𝒮 $e_0$ ρ)
    $s_0$ → if $s_0$ then (𝒮 $e_1$ ρ)
              else (𝒮 $e_2$ ρ)
    $d_0$ → let $d_1$ = 𝒮 $e_1$ ρ
                $d_2$ = 𝒮 $e_2$ ρ
            in $\boxed{\texttt{if}\ d_0\ \texttt{then}\ d_1\ \texttt{else}\ d_2}$
            end
```

Figure 7: A direct-style polyvariant specializer.

```
R : M → Exp
R d = d
R s = $\boxed{s}$
R $f_t$ = let v′ = gensym
              e′ = R(f (L t v′))
          in $\boxed{\texttt{lambda}\ v'\texttt{:}t\texttt{.}e'}$

L : Type × Exp → M
L (t->t′) $e_0$ = (λ v . let $e_1$ = R v in
                        L t $\boxed{e_0\ e_1}$$)_t$
L atom $\boxed{e}$ = e
```

Figure 8: Reification function R and reflection function L.

$$[\![mix]\!]\ e\ x = R([\![R([\![R(\mathcal{S}\ e\ [\,])]\!]\ x)]\!]\ \boxed{\mathtt{y}})$$

but this is just a hypothetical and rather limited way to access $\mathcal{S}$.

Now we return to the sum example to see the result of specializing it without cyclic values. Conceptually[1], we specialize the text of sum to its size and stride like this:

$$\mathcal{S}\ sum\ [\mathtt{from}\to\boxed{\mathtt{from}}\ \ \mathtt{to}\to\boxed{\mathtt{to}}\ \ \mathtt{size}\to 8\ \ \mathtt{stride}\to 8\ \ \mathtt{r}\to\boxed{\mathtt{r}}]$$

In the residual code, the mask computation `((1 << b) - 1)` becomes constant, but all other operations are unaffected.

2.1 Compiler Generation

If we use a literal implementation of $\mathcal{S}$ to specialize programs, then every time we generate a residual program, we also traverse and dispatch on the source text. The standard way to avoid this repeated work is to introduce another stage of computation, that is, to use a compiler generator *cogen* instead of a specializer *mix*. The compiler generator converts f into a synthesizer of specialized versions of f:

$$[\![f]\!]\ x\ y = [\![[\![[\![cogen]\!]\ f]\!]\ x]\!]\ y$$

These systems are called compiler generators because if f is an interpreter, then $[\![cogen]\!] f$ is a compiler; the part of the execution of f we call 'interpretation overhead' is only performed once. Although a procedure like sum is not what we normally think of as an interpreter, the idea is the same: factoring-out the overhead of using a general representation.

The standard way of implementing a compiler generator begins with a static analysis of the program text, then produces the synthesizer by syntax-directed traversal of the text annotated with the results of the analysis. Cogen knows what will be constant but not the constants themselves. We call such information *binding times*; they correspond to the injection tags on a members of M. We say members of Val are *static* and members of Exp are *dynamic*. The binding times form a lattice because they represent partial information: it is always safe for the compiler to throw away information; this is called *lifting* and is the meaning of the `lift` annotation in the λ-language.

[BoDu93] shows how to derive a cogen from λ-mix in two steps. The first step converts a specializer into a compiler generator by adding an extra level of quoting to $\mathcal{S}$ so static statements are copied into the compiler and dynamic ones are emitted. The second step involves adding a continuation argument to $\mathcal{S}$ to allow propagation of a static context into the arms of a conditional with a dynamic test. One of the interesting results of [Danvy96] is how this property (the handling of sum-types) can be achieved while remaining in direct style by using the shift/reset control operators ([DaFi92] Section 5.2).

Making a working implementation of a compiler generator in a call-by-value language requires handling of memoization, inlining, and code duplication as well. Practical systems usually supply heuristics and syntax to control these features. Many systems (including ours) use the dynamic-conditional heuristic, which inlines calls to procedures that do not contain a conditional with dynamic predicate.

A remarkably pleasing though less practical way of implementing $[\![cogen]\!]$ is by self-application of a specializer $[\![[\![mix]\!]\ mix\ mix]\!]$, as suggested in [Futamura71] and first implemented in [JoSeSo85].

[1]Not formally because our λ-language is not the ML of the example.

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

$$m \in \mathsf{M} = \mathsf{Exp} + \mathsf{Val} + \mathsf{Cyclic} + \mathsf{F}$$

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

Figure 9: Extending domains and R for cyclic values.

```
S [e_0+e_1] ρ = match (S e_0 ρ, S e_1 ρ)
      (⟨b q r⟩, s) →
            let r' = (r + s) % b
                q' = (r + s) / b
            in ⟨b [q + q'] r'⟩
            end

S [e_0*e_1] ρ = match (S e_0 ρ, S e_1 ρ)
      (⟨b q r⟩, s) →
          if s > 0 then ⟨sb q sr⟩
          else if s < 0 then error
          else 0
```

Figure 10: First attempt at extending $\mathcal{S}$ to cyclic values; normal form is maintained.

3 Cyclic Integers

This section shows how adding some rules of modular arithmetic to the compiler generator can unroll loops, make shift offsets static, and eliminate the division and remainder operations inside the `load_sample` procedure.

Figure 9 defines the Cyclic domain, redefines M to include Cyclic as a possible meaning, and extends R to handle cyclic values. Whereas previously an integer value was either static or dynamic (either known or unknown), a cyclic value has known base and remainder but unknown quotient. The base must be positive. Initially we assume the remainder is 'normal', ie non-negative and less than the base.

Figure 10 gives an initial version of the addition and multiplication cases for $\mathcal{S}$ on cyclic values. Again we assume cases not given are avoided by lifting, treating the primitives as unknown (allowing $\diamond$ to match any primitive), or by using the commutivity of the primitives. The multiplication rule doesn't handle negative scales. A case for adding two cyclic values by taking the GCD of the bases is straightforward, but has so far proven unnecessary. Such multiplication is also possible, though more complicated and less useful.

Note that this addition rule contains a dynamic addition to the quotient. But in many cases q' is zero; so the addition may be omitted up by the backend (GCC handles this fine). But the allocation of a new dynamic location would confuse the sharing analysis (see Section 4). Furthermore, The multiplication rule has its own defect: in order to maintain normal form we must dissallow negative scales.

The rules used by Nitrous appear in Figure 11. They are simpler and more general because Nitrous imposes normal form only at memoization points.

Figure 12 gives rules for `zero?`, division, and remainder. These rules are interesting because the binding time of the results depends on the static value rather than just the binding times of the arguments as in the previous rules. In the case of `zero?`, if the remainder is non-zero, then we can statically conclude that the original value is non-zero. But if the remainder is zero, then we need a dynamic test of the quotient. This is a conjunction short-circuiting

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

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

Figure 11: Rules for addition and multiplication.

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

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

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

Figure 12: More rules for cyclic values.

across stages, and is why we require a polyvariant system. If we constrain such tests to be immediately consumed by a conditional, then one could probably incorporate these techniques into a monovariant system.

Division and remainder could also use polyvariance, but experience indicates this is expensive and is not essential, so our systems just raise an error.

Instead of adding rules to the specializer, we could get some of the same functionality by defining (in the object language) a new type which is just a partially static structure with three members. The rules in Figures 10 and 12 become procedures operating on this type. This has the advantage of working with an ordinary specializer, but the disadvanage of not interacting well with sharing.

Now we explain the impact of cyclic values on the `sum` example. The result of

```
$\mathcal{S}$ $sum$ [from$\rightarrow\langle$32 $\boxed{\text{fromq}}$ 0$\rangle$ to$\rightarrow\langle$32 $\boxed{\text{toq}}$ 0$\rangle$
      size$\rightarrow$8 stride$\rightarrow$8 r$\rightarrow\boxed{\text{r}}$]
```

appears in Figure 13. Because the loop index is cyclic three equality tests are done in the compiler before it reaches an even word boundary. At this point, the specializer emits a dynamic test and forms the loop. Note that `fromq` and `toq` are word-pointers.

If the alignments of `from` and `to` had differed, then the 'odd' iterations would have been handled specially before entering the loop. The generation of this prelude code is a natural and automatic result of using cyclic values; normally it is generated by hand or by special-purpose code in a compiler.

If we want to apply this optimization to a dynamic value, then we can use case analysis to convert it to cyclic before the loop, resulting in one prelude for each possible remainder, followed by a single loop.

Arbitrary arithmetic on pointers could result in values with any base, but once we are in a loop like `sum` we want a particular base. `set-base` gives the programmer control:

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

Figure 13: Residual code automatically generated with cyclic values.

```
fun binop (from, to, size, stride)
          (from', to', size', stride') =
   if ((from = to) andalso (from' = to'))
   then ()
   else (... ; binop( ... ))
```

Figure 14: Looping over two signals.

```
(set-base $m$ $b$) $\rightarrow$ $\langle b\ d\ r\rangle$
```

Since m may be dynamic, `set-base` can be used to perform case analysis. While we currently rely on manual placement of `set-base`, we believe automation is possible.

3.1 Multiple Signals

If a loop reads from multiple signals simultaneously then it must be unrolled until all the signals return to their original alignment. The ordinary way of implementing a pair-wise operation on same-length signals uses one conditional in the loop because when one vector ends, so does the other. Since our unrolling depends on the conditional, this would result in the alignments of one of the vectors being ignored.

To solve this, we perform such operations with what normally would be a redundant conjunction of the end-tests. In both implementations the residual loop has only one conditional, though after it exits it makes one redundant test[2]. Figure 14 illustrates this kind of loop.

Because 32 has only one prime factor (2), on 32-bit machines this conjunction amounts to taking the worst case of all of the signals. If the word-size were composite then more complex cases could occur, for example, 24-bit words with signals of stride 8 and 12 results in unrolling 6 times.

3.2 Irregular Data Layout

The `sum` example shows how signals represented as simple arrays can be handled. The situation is more complex when the data layout depends on dynamic values. Examples of this include sparse matrix representations, run length encoding, and strings with escape sequences. Figure 15 shows how 15-bit values might be encoded into an 8-bit stream while keeping the shift offsets static. It works because both sides of the conditional of `v` are specialized.

`Read_esc` is a good example of the failure of the dynamic-conditional heuristic. Unless we mark the recursive call as dynamic (so it is not inlined), specialization would diverge because some strings are never aligned, as illustrated in Figure 16.

[2]Nitrous does this because it uses continuations; Simple does because its compiler to C translates `while(E&&F)S` to `while(E)while(F)S`.

```
fun read_esc from to r =
  if from = to then r
  else let val v = load_sample from 8
    in if (v < 128)
      then read_esc (from + 8) to (next v r)
       else d@ read_esc (from+16) to
       (next (((v & 127) << 8) |
         (load_sample (from + 8) 8)) r)
    end
```

Figure 15: Reading a string of 8-bit characters with escape sequences. `d@` indicates a dynamic call.

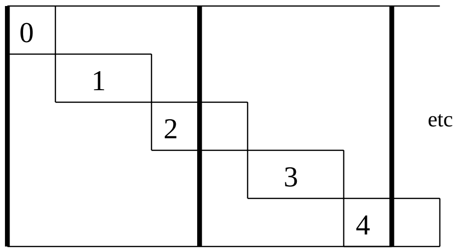

Figure 16: A string with escapes illustrating need for dynamic call annotation in `read_esc`.

4 Sharing and Caching

The remaining inefficiency of the code in Figure 13 stems from the repeated loads. The standard approach to eliminating them is to apply common subexpression elimination (CSE) and aliasing analysis (see Chapter 10.8 of [ASeUl86]) to residual programs. Efficient handling of stores is beyond traditional techniques, however. We propose fast, optimistic sharing and static caching as an alternative.

We implement the cache with a monad [Wadler92]. Uses of the `load_word` primitive are replaced by calls to a cached load procedure `load_word_c`. The last several addresses and memory values are stored in a table in the monad; when `load_word_c` is called the table is checked. If a matching address is found, the previously loaded value is returned, otherwise memory is referenced, a new table entry is created, and the least recently used table entry is discarded. Part of the implementation appears in Appendix A. In fact, any cache strategy could be used as long as it does not depend on the values themselves.

Note that safely eliminating loads in the presence of stores requires negative may-alias information (knowing that values will not be equal) [Deutsch94]. We have not yet implemented anything to guarantee this.

The prime variable is the size of the cache. How many previous loads should be stored? Though this is currently left to a manual setting, automation appears feasible because requirements combine simply.

How does the cache work? Since the addresses are dynamic any kind of equality test of the addresses will be dynamic. Yet these tests must be static if the cache is to be eliminated. Our solution is to use a conservative early equality operator for the cache-hit tests:

$$\begin{aligned}
&\mathcal{S}\ \boxed{\texttt{early=}\ e_0\ e_1}\ \rho\ \texttt{= match (}\mathcal{S}\ e_0\ \rho\texttt{,}\ \mathcal{S}\ e_1\ \rho\texttt{)}\\
&\quad (d_0\texttt{,}\ d_1\texttt{)}\ \rightarrow\ \texttt{aliases?(}d_0\texttt{,}\ d_1\texttt{)}\\
&\quad (\langle b_0\ q_0\ r_0\rangle\texttt{,}\ \langle b_1\ q_1\ r_1\rangle\texttt{)}\ \rightarrow\\
&\qquad\qquad b_0\ \texttt{=}\ b_1\ \texttt{and aliases?(}q_0\texttt{,}\ q_1\texttt{)}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{and}\ r_0\ \texttt{=}\ r_1
\end{aligned}$$

This operator takes two dynamic values and returns a static value; the compiler returns true only if it can prove the values will be equal, this is positive alias (sharing) information. The aliasing information becomes part of the static information given to compilers, stored in the memo tables, etc. Details appear in [Draves96].

In Nitrous the generated compilers keep track of the names of the dynamic values; the `aliases?` function merely tests these names for equality. Thus at compile time a cached load operation requires only a set-membership (`memq`) operation. These names are also used for inlining without a postpass (among other things), so no additional work is required to support `early=`. Simple uses textual equality of the terms.

The cache functions like a CSE routine specialized to examine only loads, so we expect a cache-based compiler to run faster than a CSE-based one. But since CSE subsumes the use of a cache and is probably essential to good performance anyway, why do we consider the cache? Because CSE cannot handle stores, but the cache does, as explained below.

Like the optimizations of the previous section, these load optimizations have been achieved by making the compiler generator more powerful (supporting `early=`). Even more so than the previous section, the source program had to be written to take advantage of this. Fortunately, with the possible exception of cache size, the modifications can be hidden behind ordinary abstraction barriers.

4.1 Store Caching

So far we have only considered reading from memory, not writing to it. Storing samples is more complicated than loading for two reasons: an isolated store requires a load as well as a store, and optimizing stores most naturally requires information to move backwards in time. This is because if we read several words from the same location, then the reads after the first are redundant. But if we store several words to the same location, all writes before the last write are redundant.

We can implement `store_word_c` the same way a hardware write-back cache does (second edition of [HePa90] page 379): cache lines are extended with a dirty flag; stores only go to memory when a cache line is discarded. The time problem above is solved by buffering the writes.

The load is unnecessary if subsequent stores eventually overwrite the entire word. Solving this problem requires extending the functionality of the cache to include not just dirty lines, but partially dirty lines. Thus the status of a line may be either clean or a mask indicating which bits are dirty and which are not present in the cache at all. When a line is flushed, if it is clean no action is required. If it is dirty and the mask is zero, then the word is simply stored. Otherwise a word is fetched from memory, bit-anded with the mask, bit-ored with the line contents, and written to memory.

5 Implementations

We currently have two implementations of bit-addressing: Nitrous and Simple, a first-order system. Both are available from `http://www.cs.cmu.edu/ spot`.

Nitrous [Draves96] is an automatic compiler generator for a higher-order, three-address-code intermediate language. It handles partially-static structures (product types), moves static contexts past dynamic conditionals (sum types), cyclic integers, sharing, and memoization. It uses the dynamic-conditional heuristic. Cache and signal libraries were implemented in a high-level language and compiled to the intermediate language[3].

[3] In fact, this compilation was performed with a generated compiler as well; the output of the output of cogen is fed into cogen.

A number of examples were specialized, compiled to C (including GCC's indirect-goto extension), and benchmarked. At the time of [Draves96], performance was about half that of hand-written, specialized C code; since then the performance has been significantly improved.

Unfortunately Nitrous fails to terminate when given more complicated input. The reason is unknown, but we suspect exponential static code is being generated as a result of the aggressive propagation of static data, particularly in the cache and inside nested loops.

In order to scale-up the examples, we built Simple, an on-line specializer that avoids using shift/reset or continuations by restricting dynamic control flow to loops (ie sum and arrow types are not fully handled). It is a straight-forward translation of the formal system presented in this paper. All procedure calls in the source programs are expanded, but the input language is extended with a while-loop construct that may be residualized:

```
Exp ::= ... | loop Var Exp Exp Exp Exp
```

which is equivalent to the following simple recursive procedure:

```
let fun G Var = if Exp then Exp else G Exp
in G Exp end
```

The loop construct is specialized as if it were a recursive procedure with the dynamic conditional heuristic and memoization: it is inlined until the predicate is dynamic, then the loop is entered and unrolled until the predicate is dynamic again. At this point, the static part must match the static part at the previous dynamic conditional.

Because Simple is based on symbolic expansion, code is duplicated in the output of the specializer. GCC's optimizer fixes most of these.

The specializer is written in SML/NJ without concern for speed but the examples here specialize in fractions of a second.

6 Example

The main example built with the simple system is an audio/vector library. It provides the `signal` type, constructors that create signals from scalars or sections of memory, combinators such as creating a signal that is the sum of two other signals, and destructors such as `copy` and `reduce`. The vector operations are suspended in constructed data until a destructor is called. Figure 17 contains a graphical representation of this kind of program.

Interleaved vectors are stored in the same range of memory; Figure 1(c) is an example of three interleaved vectors. With an ordinary vector package, if one were to pass interleaved vectors to a binary operation, then each input word would be read twice. A on-chip hardware cache makes this second read relatively inexpensive. But with the software cache the situation is detected *once* at code-generation time; specialization replaces a cache hit with a register reference.

Figure 18 gives the signature for part of the library. The semantics and implementation are mostly trivial; some of the code appears in Appendix B. One exception is that operations on multiple signals use a conjunction on the end test (Section 3.1). As a corollary, `endp` of an infinite signal such as a constant always returns true.

The delay operator returns a signal of the same length as its input, thus it loses the last sample of the input signal. The other possibility (that it returns a signal one longer) requires sum-types because there would be a dynamic conditional in the `next` method.

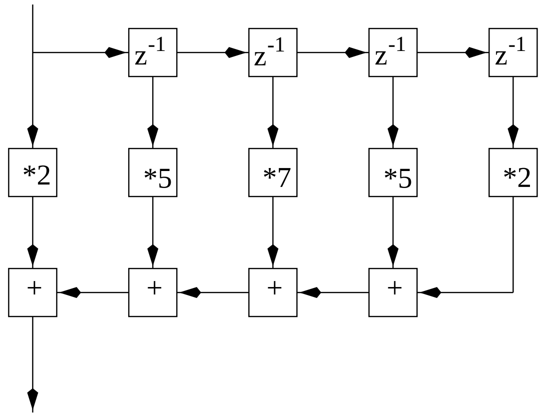

Figure 17: A graphical 'tinker-toy' DSP program. z^{-1} is a delay.

The filter combinator is built out of a series of delays, maps, and binops. Another combinator built from combinators is the FM oscillator.

Simple uses first-order analogues of the higher-order arguments. We can implement recursive filters (loops in the dataflow) with state, as `wavrec`, `scan`, and `delay1` do. A higher-order system would support a general purpose `rec` operator for creating any recursive program.

The benchmarks were performed by translating the specialized code to C and compiling with GCC v2.7.2 with the -O1 option. We also collected data with the -O2 option, but it was not significantly different so we do not present it. O3 is not available on our SGI. There are two groups of examples, the audio group (Figure 19) and the video group (Figure 20). The audio group uses 2000-byte buffers and 16-bit signals; the video group uses 4000-byte buffers and mostly 8-bit signals.

Each of the examples was run for 1000 iterations; real elapsed time was measured with the `gettimeofday` system call. The whole suite was run five times, and the best times were taken. The R4400 system is an SGI Indigo^2 with 150Mhz R4400 running IRIX 5.3. The P5 is an IBM Thinkpad 560 with 133Mhz Pentium running Linux 2.0.27.

The graphs show the ratio of the execution time of the code generated by Simple to manually written C code. In the audio group, this code was written using `short*` pointers and processing one sample per iteration. In the video group, the code was written using whole-word memory operations and immediate-mode shifts/masks. Some of the code appears in Appendix D.

Some of the static information used to create the specialized loops appears in Appendix C. These are generally arguments to the 'interpreter' `copy`, which is used for all the audio examples. The video examples also use `copy`, except iota, sum, and sum12.

The audio examples operate on sequential aligned 16-bit data unless noted otherwise:

inc add 10 to each sample.

add two signals to form a third.

filter2 filter with kernel width 2.

filter5 filter with kernel width 5. The manual code doesn't unroll the inner loop over the kernel.

fm 1 a one oscillator FM synthesizer.

```
sig
  type samp
  type signal
  type address
  type binop = samp * samp -> samp

  fun get: signal -> samp
  fun put: signal -> samp -> unit
  fun next: signal -> signal
  fun endp: signal -> bool

  fun memory: address * address
            * int * int -> signal
  fun constant: samp -> signal

  fun map: (samp -> samp) * signal -> signal
  fun map2: binop * signal * signal -> signal
  fun delay1: signal * samp -> signal
  fun scan: signal * samp * binop -> signal
  fun lut: address * signal -> signal
  fun sum_tile: samp * signal * int -> signal

  fun copy: signal * signal -> unit
  fun reduce: signal * samp * binop -> samp

  fun filter: signal * (samp * samp) list
               -> signal
  fun fm_osc: signal * int * address * int *
              signal * int -> signal
end
```

Figure 18: Signature for signal library.

fm 2 a one-in-one oscillator FM synthesizer.

lut a look-up table of size 256. The input signal is 8-bits per pixel.

sum all the samples in the input

wavrec an FM synthesizer with feedback.

The video examples operate on sequential aligned 8-bit data unless noted otherwise:

copy no operation.

gaps destination signal has stride 16 and size 8.

cs68 converts binary to ASCII by reading a six-bit signal and writing eight.

cs86 ASCII to binary by reading eight and writing six.

iota fills bytes with 0, 1, 2, ...

sum as in Figure 2, specialized as in Figure 3

sum12 a twelve-bit signal.

Figure 21 contains two more graphs. The graph on the left compares two ways of implementing the `sum` example. The baseline code reads whole words and uses explicit shifts and masks to access the bytes. This is compared to code that uses `char*` pointers, but is unrolled the same number of times (four and eight). Despite its higher instruction count, the word-based code runs faster (all the bars are higher than 1.0).

The graph on the right compares general code written using bit-addressing to specialized code. All the code is handwritten. As one expects, without specialization bit-addressing is very expensive. Higher levels of abstraction such as the signal library would incur even higher expense.

7 Conclusion

We have shown how to apply partial evaluation and specialization to problems in media-processing. The system has been implemented and the benchmarks show it has the potential to allow programmers to write and type-check very general programs, and then create specialized versions that are comparable to hand-crafted C code. Neither implementation is yet practical, but we belive both are fixable.

The basic idea is to introduce linear-algebraic properties of integers into partial evaluation instead of treating them as atoms. The programmer can write high-level specifications of loops, and generate efficient implementations with the confidence that the partial evaluator will preserve the semantics of their code. By making aliasing and alignment static, the operations normally performed by a hardware cache at runtime can be done at code generation time.

References

[ASeUl86] A V Aho, R Sethi, J D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley 1986.

[BoDu93] Anders Bondorf, Dirk Dussart. Handwriting Cogen for a CPS-Based Partial Evaluator. *Partial Evaluation and Semantics-Based Program Manipulation*, 1994.

[ChaUng91] Craig Chambers, David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. *Conference on Programming Language Design and Implementation*, 1990.

[CoHoNoNoVo96] Charles Consel, Luke Hornof, Francois Noël, Jacque Noyé, Nicolae Volanschi. A Uniform Approach for Compile-Time and Run-Time Specialization. *Dagstuhl Workshop on Partial Evaluation (LNCS1110)*, 1996.

[Consel93] Charles Consel. Polyvariant Binding-Time Analysis For Applicative Languages. *Partial Evaluation and Semantics-Based Program Manipulation*, 1993.

[DaFi92] Olivier Danvy, Andrzej Filinksi. Representing Control, a Study of the CPS Transformation. *Mathematical Structures in Computer Science*, 2(4):361-391.

[Danvy96] Olivier Danvy. Type-Directed Partial Evaluation. *Principles of Programming Languages*, 1996.

[Deutsch94] Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond *k*-limiting. *Conference on Programming Language Design and Implementation*, 1994.

[Draves96] Scott Draves. Compiler Generation for Interactive Graphics using Intermediate Code. *Dagstuhl Workshop on Partial Evaluation (LNCS1110)*, 1996.

[EnHsKa95] Dawson Engler, Wilson Hsieh, M Frans Kaashoek. ‘C: A Language for High-Level, Efficient, and Machine-independent Dynamic Code Generation. *Conference on Programming Language Design and Implementation*, 1995.

[Futamura71] Y Futamura. Partial evalutaion of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2:45-50.

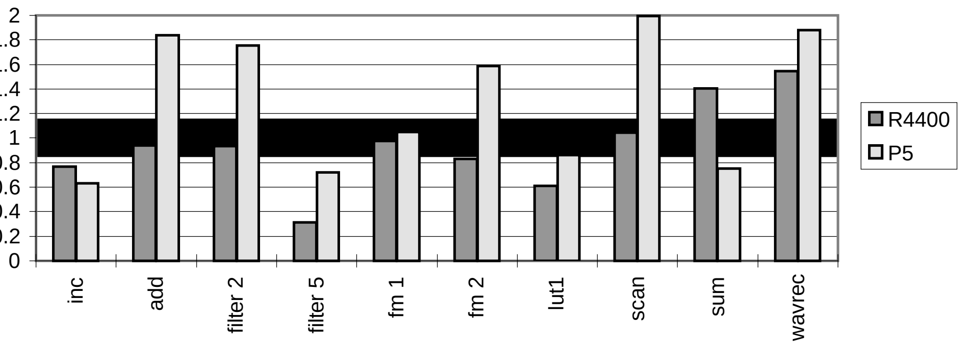

Figure 19: Audio group. Speed of automatically generated code normalized to speed of hand-specialized code.

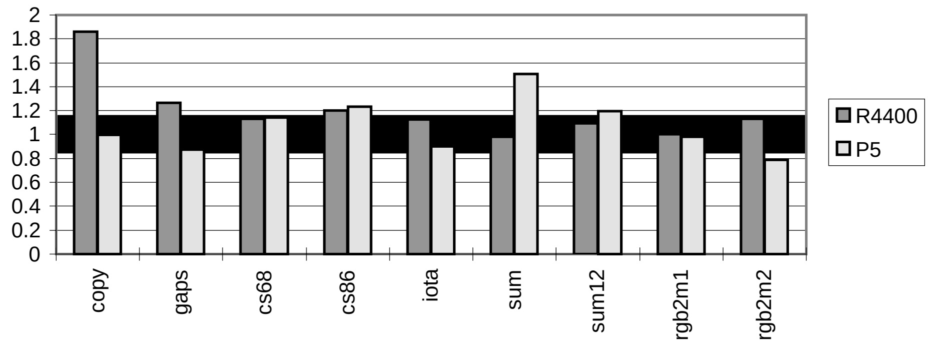

Figure 20: Video group. Speed of automatically generated code normalized to speed of hand-specialized code.

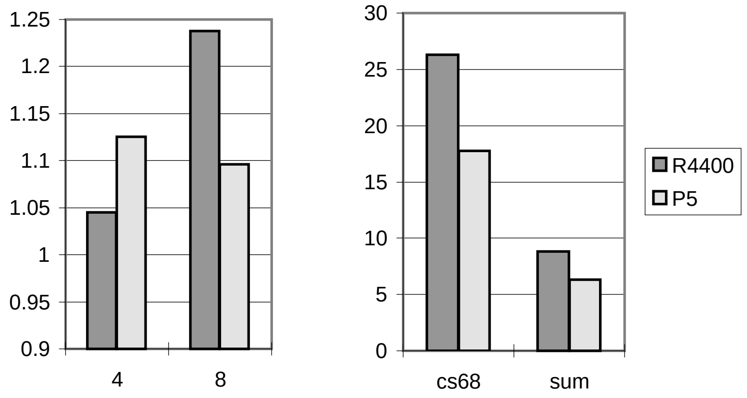

Figure 21: The graph on the left shows speed of bytes normalized to words unrolled four and eight times. The graph on the right shows the speed of general code normalized to specialized code.

[GoJo91] Carsten K Gomard, Neil Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1:21-69.

[GoJoSte96] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification*. Addison-Wesley 1996.

[Granger89] Philippe Granger. Static Analysis of Arithmetic Congruences. *International Journal of Computer Math*, ?:165-199.

[GuBoGaMa91] P Le Guernic, M Le Borgne, T Gauthier, C Le Maire. Programing real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1305-1320.

[Henglein91] Fritz Henglein. Efficient Type Inference for Higher-Order Binding-Time Analysis. *International Conference on Functional Programming Languages and Computer Architecture*, 1991.

[HePa90] John L Hennessy, David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann 1990.

[IEEE91] IEEE. *IEEE Standard 1076: VHDL Language Reference Manual*. IEEE 1991.

[JoGoSe93] Neil Jones, Carsten K Gomard, Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall 1993.

[JoSche86] Ulric Jørring, William Scherlis. Compilers and Staging Transformations. *Principles of Programming Languages*, 1986.

[JoSeSo85] Neil D Jones, P Sestoft, H Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. *Rewriting Techniques and Applications, Dijon, France*, 1985.

[LeLe96] Peter Lee, Mark Leone. Optimizing ML with Run-Time Code Generation. *Conference on Programming Language Design and Implementation*, 1996.

[Massalin92] Henry Massalin. *Efficient Implementation of Fundamental Operating System Services*. Columbia 1992.

[PiLoRei85] Rob Pike, Bart Locanthi, John Reiser. Hardware/Software Trade-offs for Bitmap Graphics on the Blit. *Software-Practice and Experience*, 15:131-151.

[Sheard96] Tim Sheard. A Type-directed, On-line, Partial Evaluator for a Polymorphic Language. OGI-TR-96-004.

[Steele90] Guy Steele. *Common Lisp the Language*. Digital Press 1990.

[Wadler92] Philip Wadler. The Essence of Functional Programming. *Principles of Programming Languages*, 1992.

[WeiCoRuSe91] Daniel Weise, Roland Conybeare, Erik Ruf, Scott Seligman. Automatic online program specialization. *International Conference on Functional Programming Languages and Computer Architecture*, 1991.

A Cache Source

A fragment of the LRU (least recently used) cache implementation.

```
val W = 32

fun mask b = (1 << b) - 1

fun load_sample (p, b) =
    let wa = p / W
    let ba = p % W
    let w0 = (load_word_cached wa)
    let s0 = (mask b) & (w0 >> ba)
    if ((ba + b) > W)
       (let ub = W - ba
        let w1 = load_word_cached
                 ((p+ub) / W)
        let s1 = (w1 & (mask (b - ub)))
        s0 | (s1 << ub))
     s0

fun flush_line line =
    let (addr, clean, mask, v) = line
    if clean line
    let v2 = (if (0 = mask) v
        (v | (mask & (load_word addr))))
    let line2 = (addr, true, 0, v2)
    store_word(addr, v2)

fun load_word_cached(addr) =
   let (effects,cache) = get_store
   if (is_pair(cache))
      (lw_loop(cache, (), addr))
      (load_word(addr))

fun lw_loop(cache, prev_cache, addr) =
 if (is_pair cache)
   (let (line, rest) = cache
    let (addr2, clean, mask, v) = line
     if (aliases(addr2,addr))
     (if (clean or (mask = 0))
         (cache_done(prev_cache, rest, addr,
                     true, 0, v))
         (error cannot_cross_streams2))
     (lw_loop (rest, (line, prev_cache), addr)))
    ((flush_line(left prev_cache));
     (let w = (load_word(addr))
     (cache_done ((right prev_cache), (), addr,
                  true, 0, w))))
```

B Signal Source

A fragment of the Simple implementation of the signal library:

```
fun memory_empty  (start, stop, size, stride) =
    (start = stop)
fun memory_next (start, stop, size, stride) =
    (v_memory, ((start+stride), stop, size, stride))
fun memory_get (start, stop, size, stride) =
    load_sample(start, size)
fun memory_put ((start, stop, size, stride), v) =
    store_sample(start, size, v)

fun constant_empty  k = true
fun constant_next k = (v_constant, k)
fun constant_get k = k
fun constant_put (k, v) = (error)
fun noise_empty  (state, ia, ic, im) = true
fun noise_next  (state, ia, ic, im) =
    (v_noise, (((lift (ia*state + ic)) % im),
               ia, ic, im))
fun noise_get   (state, ia, ic, im) = state
fun noise_put (state, ia, ic, im) = (error)

fun bin_empty (op, v, w) =
    ((vec_empty v) and (vec_empty w))
fun bin_next  (op, v, w) =
    (v_bin, (op, (vec_next v), (vec_next w)))
fun bin_get   (op, v, w) =
    (do_op (op, (vec_get v), (vec_get w)))
fun bin_put   ((op, v, w), q) = (error)

fun delay1_empty (h, v) = (vec_empty v)
fun delay1_next  (h, v) =
    (v_delay1, ((vec_get v), (vec_next v)))
fun delay1_get   (h, v) = h
fun delay1_put   ((h, v), q) = (error)

fun scan_empty (op, h, v) = (vec_empty v)
fun scan_next  (op, h, v) = (v_scan, (op,
       (do_op (h, (vec_get v))), (vec_next v)))
fun scan_get   (op, h, v) = h
fun scan_put   ((op, h, v), q) = (error)

fun lut_empty (m, v) = (vec_empty v)
fun lut_next (m, v) =
        (v_lut, (m, (vec_next v)))
fun lut_get (m, v) =
        (load_word (m + ((vec_get v))))
fun lut_put ((m, v), w) = (error)

fun sum_tile_empty (v, max, in) =
       (vec_empty in)
fun sum_tile_next (v, max, in) =
    let next = ((v + (vec_get in)) & (max-1))
    (v_sum_tile, (next, max, (vec_next in)))
fun sum_tile_get (v, max, in) = v
fun sum_tile_put (v, max, in) = (error)

fun reduce (op, init, vec) =
   loop (v, vec) ((lift init), vec)
        (vec_empty vec)
        ((do_op(op, v, (vec_get vec))),
         (vec_next vec))
        v

fun copy (a, b) =
    loop (a, b) (a, b)
         ((vec_empty a) and (vec_empty b))
         ((vec_put (b, (vec_get a)));
          ((vec_next a), (vec_next b)))
         ()

fun filter (i, k, pre) =
   if (is_pair k)
     (v_binop, (op_plus,
      (v_map, (op_times, (left k), i)),
      (filter ((v_delay1, ((left pre)), i),
               (right k), (right pre)))))
      i
```

```
fun fm_osc (mod_freq, c, wav, size,
            base_freq, init_phase) =
    let prec = 8
        (v_lut,
         (wav,
          (v_map,
           (op_shift_right, prec,
            (v_sum_tile,
             (init_phase,
              (size * (1<<prec)),
              (v_bin, (op_plus, base_freq,
               (v_map, (op_shift_right, prec,
                 (v_map,
                  (op_times, c,
                   mod_freq)))))))))))

fun rgb2m (r, g, b, m) =
    ((v_map, (op_div, 64,
        (v_bin, (op_plus, (v_bin, (op_plus,
              (v_map, (op_times, 30, r)),
              (v_map, (op_times, 25, g)))),
              (v_map, (op_times, 9, b)))))),
     m)
```

C Signal Examples

Programs implemented with the signal library.

```
val add = (op_plus, sig16, sig16_1, sig16_2)

val inc = (op_plus, sig16, (v_constant, 10),
           sig16_1)

val filter2 = ((v_bin,
                (op_plus,
                 (v_delay1,
                  ((’first), sig16)),
                 sig16)),
               sig16_2)

val kernel = (1, 2, 4, 2, 1, ())
val prefix = ((’a), (’b), (’c), (’d), (’e), ())
val filter5 = ((filter (sig16, kernel, prefix)),
               sig16_1)

val lut1 = ((v_lut, ((’buf), sig8)), sig16)

val wavtab1 =  ((v_lut_feedback,
                 ((’buf), 1024, 1, 32,
                  (’prev), sig16)),
                sig16_1)

val fm1 = ((fm_osc ((v_constant, 0), 0,
                    (’buf), 1024,
                    (v_constant, 256),
                    (’init_phase))),
           sig16)

val fm2 = ((fm_osc ((osc ((’buf), 1024,
                          (v_constant, 256),
                          (’phase0))), 1,
                    (’buf), 1024,
                    (v_constant, 256), (’phase1))),
           sig16)

val rgb2m_1 = rgb2m (rgba_r, rgba_g, rgba_b, mono8)
val rgb2m_2 = rgb2m (rgb_r, rgb_g, rgb_b, mono8)
val base64_encode = (aligned_6s, aligned_bytes)
val base64_decode = (aligned_bytes, aligned_6s)
```

D Manual Code

Baseline C code.

```
int
sum16(short *start, short *stop,
      int sum) {
  while (start != stop) {
    sum += *start++;
  }
  return sum;
}

void
filter2(short *start, short *stop,
        short *start1, short *stop1) {
  while (start != stop) {
    *start1 = start[0] + start[1];
    start++;
    start1++;
  }
}

void
filter5(short *start, short *stop,
        short *start1, short *stop1) {
  int i, t;
  while (start != stop) {
    t = 0;
    for (i = 0; i < 5; i++)
      t  += start[i];
    *start1 = t;
    start++;
    start1++;
  }
}

int
sum8(int *start, int *stop,
     int sum) {
   int v;
   while(start != stop) {
     v = *start;
     sum += (((v>>0)&255) +
             ((v>>8)&255) +
             ((v>>16)&255) +
             ((v>>24)&255));
     start += 1;
   }
   return sum;
}

void
iota(int *start, int *stop) {
  int i = 0;
  while(start != stop) {
    *start++ = i | ((i+1)<<8) |
      ((i+2)<<16) | ((i+3)<<24);
    i+=4;
  }
}

void
copy(int *start0, int* stop0,
     int *start1, int* stop1) {
  while (start0 != stop0)
    *start0++ = *start1++;
}

void
gaps(int *start0, int* stop0,
     int *start1, int* stop1) {
  while (start0 != stop0) {
    int v = *start0;
    int b0 = (v>>0)&255;
    int b1 = (v>>8)&255;
    int b2 = (v>>16)&255;
    int b3 = (v>>24)&255;
    int mask = 0xff00ff00;
    start1[0] = (start1[0] & mask)
      | b0 | (b1 << 16);
    start1[1] = (start1[1] & mask)
      | b2 | (b3 << 16);
    start0++;
    start1+=2;
  }
}

int
sum12(int *start, int *stop) {
  int sum = 0;
  while (start != stop) {
  int w0 = start[0];
  int w1 = start[1];
  int w2 = start[2];
  sum += ((w0 & 0xfff) +
          ((w0 >> 12) & 0xfff) +
          (((w0 >> 24) & 0xff)
           | ((w1 & 0xf) << 8)) +
          ((w1 >> 4) & 0xfff) +
          ((w1 >> 16) & 0xfff) +
          (((w1 >> 28) & 0xf)
           | ((w2 & 0xff) << 4)) +
          ((w2 >> 8) & 0xfff) +
          ((w2 >> 20) & 0xfff));
  start += 3;
  }
  return sum;
}

void
fm1(int *lut, int phase,
    short *start, short *stop) {
  while (start != stop) {
    *start++ = lut[phase>>8];
    phase += 256;
    phase = phase & ((1024*256)-1);
  }
}
```