Compiling Synchronous Kahn Networks to Efficient Reconfigurable Hardware

Jean Vuillemin[*], Jean-Baptiste Note
Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris France.

January 11, 2007

Abstract

We present research on *automatically compiling efficient reconfigurable hardware* from *high level software specification.*

Our abstract computational model is that of finite synchronous Kahn networks [10, 11]. In such networks, every variable carries a unique stream of integer values, as a response to some integer input stream.

Commercial systems exist to generate hardware (say VHDL) from high-level (say C) code [6]. Yet, none claims to generate *efficient hardware*, whose performance is no worse than twice that of *any hand-crafted* hardware design. Our system shows that it is possible to write the source code so that the compiler automatically generates such efficient hardware, at least within over a dozen leading edge video applications [13, 14, 19].

Within a software application (say *soft*), a specific co-routine (say *hard*) is *co-designed*, in both hardware and software. The compiler output for *hard* is an application specific hardware design for some reconfigurable system [19, 13, 16]. The compiler output for *soft* is a specific software running on the host, together with interfaces [14] to the *hard* co-routine running on the hardware co-processor.

The compiler proceeds in stages. Each stage exploits the previous *Automatic Annotations* **AA** in order to contribute useful **AA** of its own. The compiler trusts *source guidance*: each **AA** may be subsumed by an explicit *Manual Annotation* **MA** in the **source code** and each **MA** is checked at run-time.

By construction, the hardware co-processor performs the very same bit-wise computation as its matching software co-routine for *hard*. It speeds up the performance of the application, and it is *correct by construction*: no need to verify the compiled hardware as such, once the compiler has been certified. Conversely, some amount of verification & test can be done through the software-only version of *soft*; much more can be achieved through hardware acceleration, including *real-time* tests & verifications.

[*]This work is dedicated to Gilles Kahn, a long standing friend & Scientific Mentor.

1 Introduction

The goal is to automatically compile *efficient hardware* from the high-level *software* **source code** of some application. Our experimental setup comprises a standard host work-station which is tightly coupled through a high bandwidth bus to a re-configurable co-processor, either external [16] or internal [13].

1.1 Hardware/Software co-design

A software application is compiled from its *soft* **source code**. The binary code is first executed on the host, at standard level of performances. All subsequent variants (hardware & software) of this code are systematically checked for bit-wise compatibility with the above specification.

Some part (say *hard*) of *soft* is co-designed, i.e. compiled and executed on some available programmable hardware co-processor. Both together bit-wise perform the very same computation as *soft*, at hopefully much higher speed. The **source code** must be thoroughly verified and tested, first without and then with help from hardware acceleration.

The whole system is thus a fully automatic hardware accelerator for software applications. The required hardware configurations get **automatically** compiled and dynamically down-loaded into the re-configurable co-processor. They are automatically interfaced with software running on the host, and benchmarks on test data are automatically conducted & reported. The bulk of the work involved in bringing up such a hardware accelerator resides in implementing all the system routines (some in software, others in hardware) required for the application to operate at speed. This paper only surveys the *hardware compiler* aspect of such a system. See [14] for more on these and other issues in *automatic hardware acceleration*.

It must be noted that, while the overall Kahn network represented by *soft* is synchronous, the *hardware accelerated* implementation is not: the bus & driver layers asynchronously communicate data between the host and the re-configurable coprocessor, while each part realizes a synchronous Kahn network.

1.2 Compiling Hardware from Software

Our experimental hardware compiler proceeds in stages.

1. The source code is macro-generated to a *single static assignment* SSA form [5].

2. The range of variables is *automatically* analyzed. When range analysis fails to converge, some *manual annotation*[1] is required as source guidance in order to proceed.

[1]Typically, the range of some critical loop variable gets asserted in the **source code**.

3. Once range analysis succeeds, a bit-level binary representation is derived from the known finite ranges, for each variable and operator (arithmetic & memory) in the **source code**.

4. There results a *Register Transfer Level* RTL description for *hard*.

5. Through *source guidance*, the design is automatically re-mapped in time & space, so as to finely tune the amount of parallelism per input cycle.

6. The compiler then performs technology mapping and partial place & route to enforce layout regularity. It also automatically inserts *re-timing registers* [12] so as to optimize the final clock speed.

7. The result of this processing is fed into the vendor's *back-end* tools to generate the final FPGA *bit-stream* configuration for the *hard* co-routine.

Each compiler stage translates one version of the **source code** to the next, by adding *Automatic Annotations* **AA**. An **AA** does not change the (bit & cycle wise) semantics of the program: it merely records a specific type property of the annotated variable, for further use. All *annotations* are part of the syntax of the source language. An explicit manual annotation **MA** in the **source code** over-rules the **AA**: it is more stringent and it must be checked at run-time.

An experienced designer uses *source guidance* through **MA** to code each specific hardware detail, whenever the automatic compiler proves sub-optimal.

Our experience with over half a dozen leading edge real-time video applications is that the compiler [14] can synthesize high performance re-configurable designs, with little *source guidance*: a few **MA** within each **source code**.

1.3 Digital Dithering

Our hardware compiler is applied here to two algorithms for *digital dithering*.

The Art of dithering is known as *half-toning* through the history of printing. Its modern incarnation renders continuous-tone (256 levels of grey) digital images on half-tone (black or white) dot devices.

Dot printing devices perform these algorithms at very high speed, on account of the required resolution & pages per minute [13]. We borrow the vocabulary from black&white ink jet printers. Dithering is rather similar for laser and impact printers, color or not.

The ink-head moves in raster-scan order, at the rate of one column of ink-spots per cycle. The output $i \in \{0, 1\}$ of dithering digitally controls each ink-nozzle: a dot of ink (worth 256 units) is dropped ($i = 1$) or not ($i = 0$) on the current spot. For specificity, we fix the image line width to 630 pixels.

Each input pixel value p is added $e = p + d$ to the error d diffused by the previous pixels. Number $e/256$ represents the fractional ink quantity to ideally drop at the spot, for some hypothetical 256 tones printer. The error remaining $r = e - 256i$ is the difference between the actual and theoretical quantities of ink dropped on the spot.

Digital dithering is done in two stages within a feed-back loop:

- $dropInk$ controls the binary output i, and the final ink-nozzle.
- $diffuseError$ diffuses to nearby pixels the error r remaining at each spot.

1.3.1 Drop Ink

The binary decision $i = t \geq h$ drop/no drop results from comparing the total ink e to the threshold h. We illustrate two versions of *dropInk*:

- In this naive code[2], the threshold is constant and equal to the nominal ink per drop, $h = 256$.

$$\begin{array}{lrcll} \textit{fun} & i & = & \mathbf{dropInk_{RD}}(\mathbf{e}) & \{\textit{: ink drop if t[8]=1; } 0 \leq t < 512 \\ & i & = & e \gg 8; \} & \quad \textit{: ink drop if } e \geq 256;\ i \in \{0,1\} \end{array} \tag{1}$$

- In the advanced version, a static *Threshold Table th* (of size, say 16) is used to compute the pseudo-random value $h = th[t\&15] = th[t[0..3]]$; this effectively eliminates Moiré effects where rendering uniform tones.

$$\begin{array}{lrcl} \textit{var} & th & = & [10, 15, 11, 8, 12, 14, 12, 13, 15, 11, 14, 15, 9, 13]; \\ & & & \qquad\qquad \textit{: Threshold Table } th[0..15] \in [8, 15] \end{array}$$

$$\begin{array}{lrcll} \textit{fun} & i & = & \mathbf{dropInk_{FS}}(\mathbf{e}) & \{ \\ & h & = & th[e\&15]; & \qquad \textit{: threshold value; } 8 \leq h < 16 \\ & i & = & (e \gg 4) > h; \} & \quad \textit{: ink drop if } e > 16h;\ i \in \{0,1\} \end{array} \tag{2}$$

1.3.2 Error Diffusion

When the ink dropped e is not equal to the nominal drop value of 256, there remains an error $r = e - 256 \times i$. This error is split & diffused to nearby pixels, so as to best preserve the overall & local ink quantity.

General

$$\begin{tikzcd} & \boxed{1} \arrow[r] \arrow[dl] \arrow[d] \arrow[dr] & \boxed{p_2} \\ \boxed{p_3} & \boxed{p_4} & \boxed{p_5} \end{tikzcd}$$

Random

$$\begin{tikzcd} & \boxed{1} \arrow[r] \arrow[dl] \arrow[d] \arrow[dr] & \boxed{\frac{1}{2}} \\ \boxed{0} & \boxed{\frac{1}{2}} & \boxed{0} \end{tikzcd}$$

Floyd-Steinberg

$$\begin{tikzcd} & \boxed{1} \arrow[r] \arrow[dl] \arrow[d] \arrow[dr] & \boxed{\frac{7}{16}} \\ \boxed{\frac{3}{16}} & \boxed{\frac{5}{16}} & \boxed{\frac{1}{16}} \end{tikzcd}$$

In the above schemes, error r at pixel $\mathrm{P}[r, c]$ is diffused to four nearby pixels $\mathrm{P}[r, c+1], \mathrm{P}[r+1, c-1], \mathrm{P}[r+1, c], \mathrm{P}[r+1, c+1]$, by (rounded) ratio $p_k \times r$ for $k \in [2..5]$ so that $1 = p_2 + \cdots + p_5$.

1.4 Summary

In this paper, two specific source codes are compiled.

[2]Apart from their concrete syntax, the arithmetic operators here have the same meaning as in the language C; except for the comparison operator, whose result is an integer in $\mathbf{B} = \{0, 1\}$ rather than a *boolean* value. All variables shown in codes have the type *int*; more precisely, they denote indefinite streams of (arbitrary size) integer values.

Section 2 presents an overview of the FPGA compiler, as applied to the simple **source code** for *random diffusion*. *Random Diffusion* is an elementary code to compile, but it is poor at rendering real images.

Section 3 illustrates the compilation of the more advanced **source code** for Floyd-Steinberg diffusion, a state-of-the-art image rendering technique. We discuss the intrinsic difficulty of *range analysis* and existing approaches.

Section 4 concludes on an optimistic note regarding *compiling efficient reconfigurable designs from high level software.*

2 Compiling Random Diffusion

We illustrate the compiler stages on the simplest dithering algorithm (constant threshold and random diffusion). It is defined by the following **source code**:

$$\begin{array}{rrcll} \mathit{fun} & i & = & \mathbf{dither_{RD1}}(\mathbf{p}) & \{ \\ & e & = & p + \mathbf{z}(d); & \textit{: pixel error} \\ & i & = & e \gg 8; & \textit{: ink drop if i=1} \\ & a & = & (e \gg 1)\&127; & \textit{: half error} \\ & d & = & a + (e\&1) + \mathbf{z}^{629}(a); & \}\textit{: diffused error} \end{array} \tag{3}$$

In this code, operator $\mathbf{z}$ denotes the unit delay between consecutive pixels on the same line; $\mathbf{z}^{630}$ denotes the delay between pixels on the same column.[3]

Code (3) is programmed in Jazz [8], an experimental language for hardware synthesis. Jazz supports higher-types and strong type-inference (à la ML [9] or Haskell [3]), together with some *object programming* (à la Java) and *operator overloading* (à la C++). Translating code (3) between Jazz, C or Java is straightforward.

```
int DitherRd      (const unsigned char px) {
    static int    zd, za[L-1], idx = 0;
         int e  =  px + zd,
             i  =  e >> 8,
             a  =  (e >> 1) & 127,
             d  =  a + (e & 1) + za[idx];                (4)
       za[idx]  =  a;
            zd  =  d;
           idx  =  (idx + 1) % (L-1);
        return     i; }
```

All codes Jazz (3)/ C (4)/Java yield the same bit-stream output sequence i on the same integer-stream p of input pixels: $0 \leq p < 256$. Range analysis derives from this unique input assertion that all variables in (3) can be safely represented by 16 bits[4] signed integers.

[3]We assume that input pixels appear in *raster-scan order* and that line width is 630.
[4]9 bits suffice here.

A strength of Jazz is that the very same code (3) can be executed over many input types, from symbolic (for code generation and range analysis) to numeric (for simulation). All variables in the **source code** (3) have the same type T. Type T supports all memory ($\mathbf{z}$, $\mathbf{z}^{629}$) and all integer operations

$$(+, -, *, \div, <<, \gg, \neg, \&, |, \oplus, <, ==, >, \geq, \leq)$$

in language C over type **int**, including shifts and Boolean operations.[5]

In that respect, Jazz is comparable to Lava [4]. In Lava, type classes and *monads* [20] allow the programmer to define new *interpretations* for the same circuit code, by sharing or specializing already-implemented behaviors.

2.1 Single Static Assignment SSA form

Program (3) is translated to *Single Static Assignment form* [2]. Thus by construction, the **source code** (3) and its SSA form (5) have identical behavior. *Feedback loops* are marked in the SSA form and the resulting graph structures all subsequent stages in the compiler.

2.2 Range Analysis

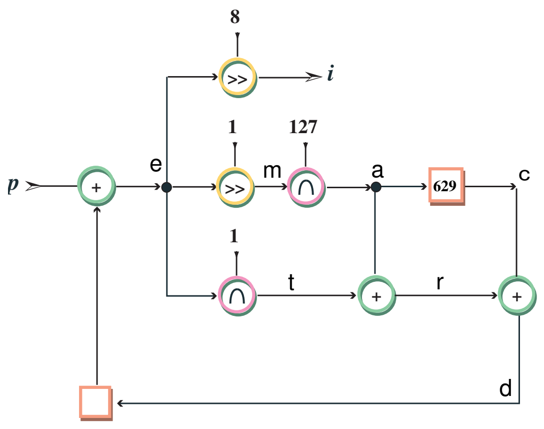
We now execute code (5) with an input $p : [0, 255]$ of symbolic type: *constant stream of interval*. The execution converges to a fixed-point in 3 iterations, as shown by the following simulation.

$$\begin{array}{lcllll}
\mathit{fun}\ i & = & \mathbf{dither_{RD3}(p : [0, 255])}\{ & : [0,255] & [0,255] & [0,255]\cdots \\
e & = & p + \mathbf{z}(d); & : [0,255] & [0,383] & [0,510]\cdots \\
i & = & e \gg 8; & : [0,0] & [0,1] & [0,1]\cdots \\
t & = & e\&1; & : [0,1] & [0,1] & [0,1]\cdots \\
m & = & e \gg 1; & : [0,127] & [0,191] & [0,255]\cdots \\
a & = & m\&127; & : [0,127] & [0,127] & [0,127]\cdots \\
r & = & t + a; & : [0,128] & [0,128] & [0,128]\cdots \\
c & = & \mathbf{z}^{629}(a); & : [0,0] & [0,127] & [0,127]\cdots \\
d & = & r + c;\} & : [0,128] & [0,255] & [0,255]\cdots
\end{array}$$

2.3 Bit-sizing

When the above analysis converges, the *fixed-point* interval ranges $[i, s]$ associated to each variable are represented in binary by arrays of $l = 1 + \lfloor \log_2(s - i) \rfloor$ bits; they are marqued as *unsigned* : $\mathbf{u}$ if $i \geq 0$, or *signed* : $\mathbf{s}$ if $i < 0$.

[5] One minor difference is that comparison operators $\{<, ==, >\}$ yield an integer in $\{0, 1\}$ rather than a Boolean, for the sake of type uniformity.



$$\begin{array}{rcll} \textit{fun} \quad i & = & \mathbf{dither_{RD2}(p)} \quad \{ & \\ f & = & \mathbf{z}(d); & :\ \textit{feedback error} \\ e & = & p+f; & :\ \textit{pixel error} \\ i & = & e \gg 8; & :\ \textit{ink drop} \\ m & = & e \gg 1; & :\ \textit{remaining error} \\ t & = & e\&1; & :\ \textit{parity} \\ a & = & m\&127; & :\ \textit{half error} \\ r & = & a+t;; & :\ \textit{row error} \\ c & = & \mathbf{z}^{629}(a); & :\ \textit{column error} \\ d & = & r+c; \quad \} & :\ \textit{diffused error} \end{array} \tag{5}$$

Figure 1: Random Diffusion in SSA form

The following bit-sized version is automatically annotated from code (5).

$$\begin{array}{rcll} \textit{fun} \quad i & = & \mathbf{dither_{RD4}(p : u_8)} & \{ \\ e & = & p+\mathbf{z}(d)\ :\mathbf{u}_9; & i = e \gg 8\ :\mathbf{u}_1; \\ m & = & e \gg 1\ :\mathbf{u}_8; & t = e\&1\ :\mathbf{u}_1; \\ a & = & a\&127\ :\mathbf{u}_7; & \\ r & = & a+t\ :\mathbf{u}_8; & c = \mathbf{z}^{629}(a)\ :\mathbf{u}_7; \\ d & = & r+c\ :\mathbf{u}_8; & \} \end{array} \tag{6}$$

2.4 Register Transfer Level RTL

Code (6) is then translated to bit-level operations on the finite bit vectors which now represent each variable. The result is an RTL description, which is automatically simplified to:

$$\begin{array}{rcll} \textit{fun} \quad i & = & \mathbf{dither_{RD5}(p : u_8)} & \{ \\ e[0..8] & = & p[0..7]+\mathbf{z}(d[0..7]); & \\ c[0..6] & = & \mathbf{z}^{629}(e[1..7]); & \\ d[0..7] & = & e[0]+e[1..7]+c[0..6]; & \} \end{array} \tag{7}$$

2.5 Technology mapping

The RTL description is then decomposed into *atomic building blocks* (memory, logic, arithmetic & communication) and each block is directly mapped the target FPGA technology. It is crucial here that each group of operators be represented by its most efficient structure, and that the *back-end* compiler for the specific FPGA technology does use that specific best structure.

In current practice, a large number of annotations must be generated by our *front-end compiler* in order to *force* the vendor's *back-end compiler* [22] to best use it's own internal structures. Among others, we found that it is mandatory to *guide* the vendor's back-end by:

- relative gate placement within arithmetic operators, based on bit order;
- relative gate placement between operators, based on topological order;
- systematically *re-time* between feed-forward atomic operators.[6]

We refer to [14] for more details on this specific *FPGA technology* part of the compiler. The final hardware layout is found in Figure 2.

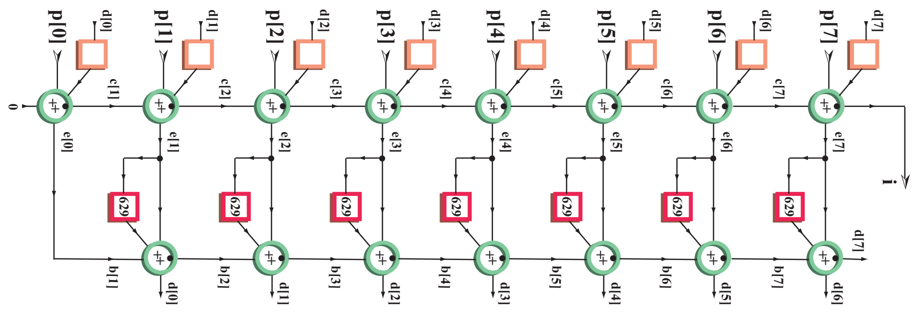


Figure 2: Hardware layout compiled from (3).

The synthesized *Digital Synchronous Circuit* comprises 15 full-adders fa (including one half-adder), 8 unit delays $\mathbf{z}$ and 7 delays $\mathbf{z}^{629}$ by 629 clock cycles. The design operates at 100 Mhz on a Virtex-II [21] FPGA.

2.6 Automatic Space Time Tradeoffs

So far, we have assumed that the input pixels are presented by 8 bits in parallel, one pixel at each processing cycle. We can measure (or simply estimate) the actual processing bandwidth P of our automatic implementation on some device, and compare it to the nominal requirement R.

$R < P < 2R$ We are doing just fine.

[6]The theory of re-timing is from [12]. Modern FPGA have *many* internal registers. Unlike ASIC technology, the problem here is not to minimize their number, but to achieve the best possible bandwidth while keeping a low overall *latency*.

$R > P$ We are too slow, say by a factor 2: $P < R < 2P$. In this case, the design must be unfolded twice in space:

1. The effective bandwidth doubles $P' = 2P$, as we process 2 pixels per clock cycle, and the clock cycle is kept the same.[7]
2. The amount of logic & control doubles: from 30 odd gates to 60 in our case.
3. The RAM to store the diffusion buffer must now have twice the IO bandwidth. Yet its content remains invariant at 629×7 bits (one block of local sRAM in current FPGA technology [21]).
4. More unfolding is needed when $2P < R$.

$2R < P$ We are too fast, by at least a factor 2! So, we can fold part of the computation from space to time and trade processing bandwidth for area. Figure 3 illustrates this case.

Each input pixel p appears in binary at the rate of one bit per clock cycle, least significant bit first and for 8 cycles. Bits of output i are produced on cycles 8,16,24,$\cdots$ which are multiples of 8.

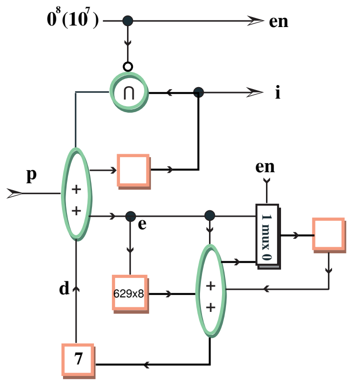


Figure 3: Bit-serial circuit compiled from (3) under the explicit manual annotation **MA** on input $p : bitSerial(8)$.

3 Floyd-Steinberg Dithering

FloydSteinberg dithering [7] is used by millions of current ink&laser jet printers.

3.1 Range Analysis

The **source code** (Figure 4) is unfolded to 17 *Static Single Assignement* SSA definitions (9) through standard techniques [2].

[7]This can effectively be achieved within the *feed-forward* parts of the design, by simply inserting re-timing register. Unfolding *feedback loops* requires more advanced techniques, and it is not always possible.

```
                                                                   : Threshold table
                var ths  =  [10,15,11,8,10,15,13,12,14,8,14,12,15,9,13,9];
fun FloydSteinberg(px)   =  di {
                     te  =  px+z(ce+e0+7*e1);                        : total error
                     e0  =  te & 15;                                  : error zero
                     eq  =  te ≫ 4;                               : error quotient
                     di  =  ths[e0]<eq;                                 : drop ink
                     e1  =  di ? eq-16 : eq;                           : error one
                     ce  =  z^628(3 * e1 + z(5 * e1 + z(e1))); }  : current error
```

Figure 4: Source code for Floyd-Steinberg diffusion.

Naive Interval Analysis succeeds on code (3) in section 2.2, but it fails on code (9): ever larger intervals are computed at each iteration, for most variables. Yet, it is sufficient to add a single manual annotation **MA** to the **source code** (9) in order for NIA to properly and automatically converge, namely:

$$e1 : [-7{,}15]. \tag{8}$$

With this single **MA** (in addition to the input assertion $px : [0, 255]$), the naive interval analysis of code (3&8) converges in 3 cycles[8] to the proper fixed-point ranges, for all 16 remaining variables in (9).

	var		definition	range	type	long name
1	de	=	**z**(ke)	[-112,255]	s9	*: diffused error*
2	te	=	px+de	[-112,510]	s10	*: total error*
3	e0	=	te&15	[0,15]	u4	*: error zero*
4	eq	=	te ≫ 4	[-7,31]	s6	*: error left*
5	th	=	ths[e0]	[8,15]	u4	*: threshold*
6	di	=	th<eq	[0,1]	u1	*: drop ink*
7	e1	=	di ? eq-16 : eq	[-7,15]	s5	*: error one*
8	e3	=	3×e1	[-21,45]	s7	*: error three*
9	e5	=	5×e1	[-35,75]	s8	*: error five*
10	e7	=	7×e1	[-49,105]	s8	*: error seven*
11	s1	=	**z**(e1)	[-7,15]	s5	*: sum one*
12	r5	=	e5+s1	[-42,90]	s8	*: r five*
13	s5	=	**z**(r5)	[-42,90]	s8	*: sum three*
14	r3	=	e3+s5	[-63,135]	s9	*: r three*
15	ce	=	$\mathbf{z}^{628}$(r3)	[-63,135]	s9	*: column error*
16	le	=	ce+e7	[-112,240]	s9	*: line error*
17	ke	=	e0+le	[-112,255]	s9	*: current error*

(9)

One understands that invariant (8) comes from the values stored in the lookup table, which are in the $[8, 16]$ range. This propagates range conditions on the

[8] All registers except for the one on line 1 are recognized as *feed-forward*, and thus effectively eliminated from the range computation.

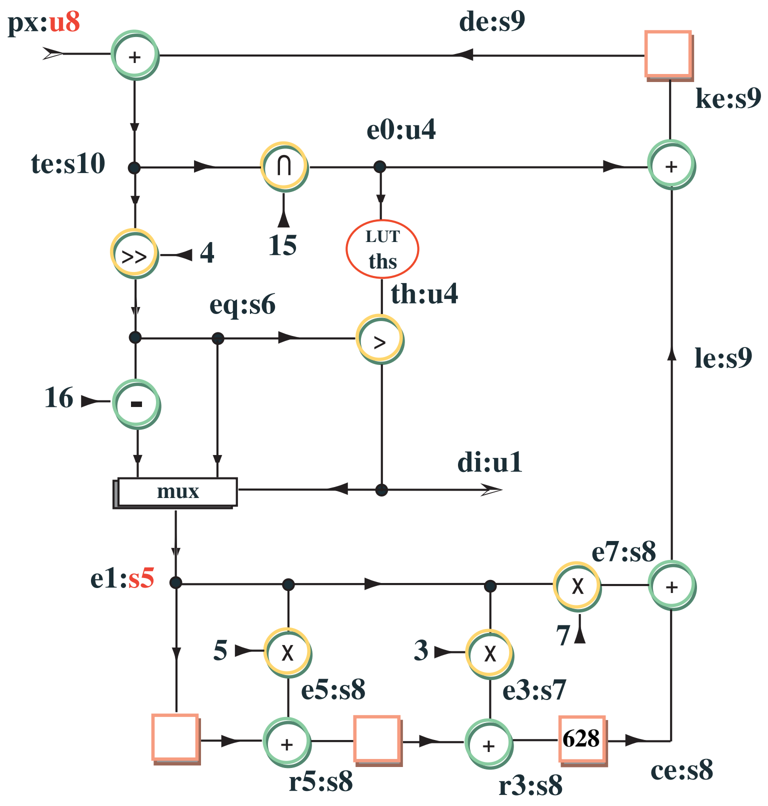


Figure 5: Floyd-Steinberg Diffusion with bit sizes

control of a multiplexer, and yields the expected range for the $e1$ variable. The fact that each error is split four ways, with coefficients summing up to one and regularly spread through time in the diffusion buffer, is then automatically captured by naive interval analysis.

One should note that the *Finite State Machine* represented by code (9) has over 2^{4400} states. All methods based on systematic state explorations are thus doomed. The same happens with almost all current video processing algorithms: due to large internal data buffers, the number of states is just too big to handle, and more abstract memory models are mandatory.

Yet, the general question of range analysis is undecidable. So, all attempts to solve it must contain a *time-out*, which admits *failure of convergence*.

The ASTREE [1] system[9] automatically and correctly analyzes (a C version of) code (9), for all 17 variables, and without any manual annotation. So, this weakness in our experimental system (relying on ASTREE to provide *critical range assertions*) can be removed, through more work/cooperation.

[9]we thank our colleagues at ENS for helping us experiment with ASTREE

Nevertheless, source guidance through **MA** asserting the range of a few peculiar variables remains a mandatory feature in all systems, such as ours.

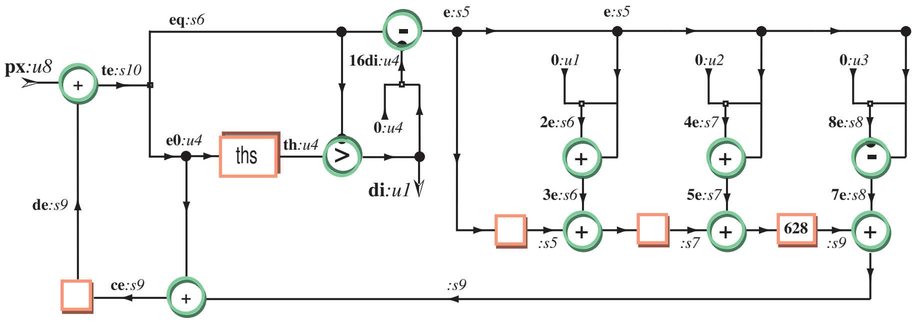


Figure 6: Floyd-Steinberg Diffusion Circuit Schemas

3.2 Hardware Synthesis

The bit level code (9) is further simplified, constant multipliers are eliminated, and each resulting block is mapped to the available FPGA technology.

software	hardware	operator
te = px+Z(ke)	te[0..9] = px[0..7]+Z(ce[0..8])	u8+Z(s9) : s10
th = ths[te&15]	th[0..3] = ths[te[0..3]]	*rom*(u4) : u4
di = th<(te ≫ 4)	d[0..5] = th[0..3]−te[4..9] di=d[5]	u4−s6 : u1
e1 = (te≫4)-16*di	e1[0..3] = te[4..7]	s5
	e1[4] = te[8]⊕di[0] ⊕ 1	u1⊕u1 : u1
e3 = (e1≪1)+e1	e3[0] = e1[0]	s7
	e3[1..6] = e1[0..4]+e1[1..4]	s5+s3 : s6
e5 = (e1≪2)+e1	e5[0..1] = e1[0..1]	s8
	e5[2..7] = e1[0..4]+e1[2..4]	s5+s2 : s6
e7 = (e1≪3)-e1	e7[0..7] = e8[0..7]-e1[0..4]	s8-s5 : s8
	e8[0..3] = 0 e8[4..7] = e1[0..3]	s8
s1 = Z(e1)	s1[0..4] = Z(e1[0..4])	Z(s5) : s5
s5 = Z(e5+s1)	s5[0..7] = Z(e5[0..7]+s1[0..4])	Z(s8+s5) : s8
ce = Z^l(e3+s5)	ce[0..8] = Z^l(e3[0..6]+s5[0..7])	Z^l(s7+s8) : s9
ke = e7+ce+(te&15)	ke[0..8] =e7[0..7]+ec[0..7]+te[0..3]	s8+s8+s4 : s9

The synthesis of the Floyd-Steinberg algorithm on a Virtex-II (xc2v2000) [21] yields a design running at 68.9MHz. It is unfolded 4 times in space to allow for parallel processing of all 4 color channels of a CYMK image.

The design, compiled for processing 720x576 (standard PAL resolution) images, consumes 306 Virtex-II slices and four RAM blocks, used to implement the diffusion buffer. The compiled design processes 68.9Mdots/second. Altogether, this amounts to printing well over two full A4 sheets per second at 1200 dpi resolution.

4 Conclusion

Beyond the two dithering examples presented here, over a dozen other leading-edge video applications from [13, 15, 19] have been so implemented [14].

These specific applications were chosen because of their computational needs, and structure as finite synchronous Kahn networks. Our compiler does not attempt to compile into hardware source code which is not of this constrained form.

For this selected set of data-flow applications, the methodology has proved effective in that the circuits compiled from the high-level software specification are all *very efficient*: their performance is no worse than twice that of the best-known independent implementation.

The only way to achieve efficiency is through good **source code**. No compiler can ever repair a poorly conceived design! The hardware compiler is a tool to help the experienced designer specify concisely and safely the required implementation choices/constraints.

The process is fully automatic from the **source code**. All generated designs may not be efficient in the above strong sense. Yet, they are all correct by construction.

Exact Range Analysis is a key component of the compiler, and so is technology mapping. If either of these components fails to be *very efficient*, the compiler as a whole will not generate efficient designs.

Neither of these two key components is Turing computable. Even in selected decidable sub-cases, many hard combinatorial problems can be naturally encoded as range-analysis or technology-mapping problems.

Hence, source guidance in the form of **MA** or other is a mandatory feature in all such general purpose hardware compiler.

At the same time, advanced techniques exist for range analysis [1] and technology mapping [14] which can speed-up the design process in very substantial ways.

Finally, we are keen to observe here that the *very same* **source code** leads to both the best compiled software version and the best hardware version of the same application. This has been observed many times before [17, 19, 18, 13, 15]. Some amount of *source guidance*, in part specific to software and in part to hardware, is usually necessary to achieve success on both fronts. Yet altogether, this entails just a few **MA**. Most improvements to the algorithm are beneficial to both hardware and software, and they are naturally reflected in a unique way within the **source code**.

Acknowledgements This work has benefited from many contributions by Marc Pouzet.

References

[1] P. Cousot & al. The ASTRIE static analyzer. *Ecole Normale Supérieure*, 2005 http://www.astree.ens.fr/.

[2] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java.* Cambridge University Press, 2002.

[3] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava: hardware design in haskell. *Proceedings of the third ACM SIGPLAN*, 1998.

[4] Koen Claessen. The lava home page.

[5] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadek. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, A 13(4):451–490, March 1991.

[6] B. Draper, W. Najjar, W. Bvhm, J. Hammes, R. Rinker, C. Ross, M. Chawathe, and Josi Bins. Compiling and optimizing image processing algorithms for fpgas. *Department of Computer Science, Colorado State University*, 80523, June 2000
http://www.cs.ucr.edu/˜najjar/papers/camp00.pdf.

[7] R. W. Floyd and L. Steinberg. An adaptive algorithm for spatial gray scale. *SID 75, Int. Symp. Dig. Tech. Papers*, 36, 1975.

[8] A. Frey, G. Berry, P. Bertin, F. Bourdoncle, and Jean Vuillemin. The jazz home page. 1998
http://www.exalead.com/jazz/index.html.

[9] INRIA. The caml home page. 1985
http://caml.inria.fr/index.fr.html.

[10] Gilles Kahn. The semantics of a simple language for parallel programming. *Information Processing 74: Proceedings of the IFIP Congress 74*, North-Holland:471–475, 1974.

[11] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. *Information Processing 77: Proceedings of the IFIP Congress 77*, North-Holland:993–998, 1977.

[12] CE Leiserson and JB Saxe. Retiming synchronous circuitry -. *Algorithmica*, 1991.

[13] A. Marshall, J. Vuillemin, T. Stansfield, I. Kostarnov, and B. L. Hutchings. A re-configurable arithmetic array for multimedia applications. In *Proceedings of the 1999 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 135–143, 1999.

[14] J.B. Note. *Compilation automatique de logiciel en circuit reconfigurable efficace*. These de Doctorat, Ecole Normale Supérieure, 2007.

[15] J.B. Note, M. Shand, and J. Vuillemin. Realtime video pixel matching. In *International Conference on Field Programmable Logic and Applications*, pages 507–512, 2006.

[16] M. Shand. Programmation de la carte pci pamette. *Ecole Normale Suphrieure*, 2005
http://www.di.ens.fr/AlgorithmiqueMaterielle.html.

[17] M. Shand and J. Vuillemin. Fast implementation of RSA cryptography. In *11-th IEEE Symposium on Computer Arithmetic*, 1993.

[18] J. Vuillemin. Re-configurable systems: Past and next 10 years. In *Vector and Parallel Processing - VECPAR'98*, volume 1573 of *L.N.C.S.*, pages 334–354. Springer-Verlag, 1998. (invited talk).

[19] J. Vuillemin, P. Bertin, D. Roncin, M. Shand, H. Touati, and P. Boucard. Programmable active memories: the coming of age. *IEEE Trans. on VLSI*, 4, NO.1:56–69, March 1996.

[20] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text*, pages 24–52, London, UK, 1995. Springer-Verlag.

[21] Xilinx. Virtex-2 Platform FPGA User Guide (UG002 version 2.0), 2005
http://www.xilinx.com/bvdocs/userguides/ug002.pdf.

[22] Xilinx. Xilinx ISE 8 software manuals, 2006
http://toolbox.xilinx.com/docsan/xilinx82/books/manuals.pdf.