IRIS: a firmware design methodology for SIMD architectures

Jan Jacobs
Océ Technologies BV,
PO Box 101,
5900MA Venlo, The Netherlands
jan.wm.jacobs@oce.com

Leroy van Engelen, Jan Kuper, Gerard J.M. Smit
University of Twente
dept EEMCS, PO Box 217,
500AE Enschede, The Netherlands

Rui Dai
National University of Singapore
Design Technology Institute Faculty of Engineering,
10 Kent Ridge Crescent, Singapore 119260

Abstract

Developing code for SIMD type hardware architectures is a tedious job. This is caused by the absence of both a coherent methodological framework and a hardware independent tooling.

Moreover, the inherently difficult nature of programming dedicated massively parallel embedded processors, complicates the matter. This paper describes a single framework, called IRIS*, to generate code for SIMD architectures. This framework is illustrated with a concrete case "Stochastic Image Quantisation".* IRIS *is based on an incremental construction of executable representations, which converge to the final target implementation in a semi-automated way.*

1 Introduction

Nowadays embedded systems manufacturers are facing tough problems in developing high performance applications. The ever growing functionality of applications combined with new programmable many-core processors increase the development complexity. Therefore Patterson [2] states: "Although compatibility with old binaries and C programs are valuable to industry ... we welcome new programming models and new architectures if they simplify efficient programming of such highly parallel systems". In addition to this we believe that parallelism not always can be derived automatically from sequential code with enough quality: we need the option to code the parallelism explicitly by the application programmer. In this paper, we focus on a methodology that improves the programmer's efficiency for *Single Instruction Multiple Data* (SIMD) architectures. The development of applications for SIMD architectures needs special attention because: (1) massive parallelism cannot be expressed adequately by current languages, (2) variables can have all bit dimensions (e.g. 10 bit integers), and (3) data dimensions of the problem and the limited memory resources on a processing element do not match in general (requiring tiling). The de facto way applications are programmed on such dedicated systems is by manually adapting sequential code, which is mostly written in C. This adaptation involves the replacement of the time critical sequential parts by parallel code. Most tooling is supplied by the manufacturer of the processor hardware and is, to no suprise and without exception, a C-compiler supporting *intrinsic instructions* (hardware dependent predefined functions). This means that the design can only be validated at the end of the development cycle, when finally the code becomes available.

Striking examples which demonstrate the weaknesses of the current approach are analysis and design faults that are discovered in late phases of the development. Thus we propose a *methodological framework for SIMD firmware development* that should at least:
a. be an ***i****ntegral* design method that supports firmware development for the whole trajectory (from problem-scouting till maintenance),
b. be ***i****nteractive* and be executable during the whole development process,
c. be ***i****ncremental*, enabling elaboration on the current state of the design,
d. supporting ***r****euse* to improve quality and efficiency,
e. be domain ***i****ndependant*, i.e. be applicable to multiple application domains.
In this paper we propose a methodological framework, called IRIS, that satisfies all these requirements. As a crucial feature of IRIS we assume that the same language is

available during the complete design process, which supports executability at all stages. We call such a language an *architectural language*. We propose an architectural language that is close to mathematics and understandable for the developer, leading to ***r**eadable* and compact code without any reference to implementation in early phases and with concise description of details in later phases. A ***s**ingle* language supporting these multiple roles is a necessity in IRIS.

First we give an overview of related work (Section 2), next we describe the IRIS methodology (Section 3). We briefly introduce the case (Section 4), followed by an elaboration of the case using IRIS (Section 5). Finally we present and discuss the results (Section 6) and the conclusions (Section 7).

2 Related Work

An influential development approach for hardware-software co-design, the Y-chart [10], is based on concurrent elaboration on multiple domains (coupled to stakeholders) at different abstraction levels. These domains, which in fact are different views for specifying a hardware system, are: behavioural (functional), structural (hierarchy of interconnected components, computer architecture) and the physical/geometrical domain (physical placement in space and physical characteristics). It is a very generic methodology, mostly used for hardware development but not well suited for developing code for existing many-core programmable processing systems.

The "Iterative Design Methodology" [8], puts emphasis on the iterative aspect in hardware software co-design. The extra-functional design properties as performance, power and resource consumption, are analysed by using post-mapping analysis tools. However, for interactive code development we need instant mapping analysis.

In addition to the common direction for functional development, in [14] an orthogonal direction, namely that of *design space exploration* is introduced. Design Space Exploration is a structured way of identification and evaluation of design alternatives, and the development of criteria. The ultimate choice, which is part of *decision recording*, starts off a next development cycle.

Agile methods such as Extreme Programming (XP) [4] try to reduce development time, typically from months to weeks, by reintroducing interactivity to the design process. These methods, however, mostly use an implementation language for the development roles. This leads to less readable and maintainable code in particular for the early phases. Recently [11] more emphasis is put on raising the level of abstraction by using new parallel languages instead of extending the traditionally used sequential languages (mostly C-based). However, these languages lack possibilities for the detailed control at elementary processor level.

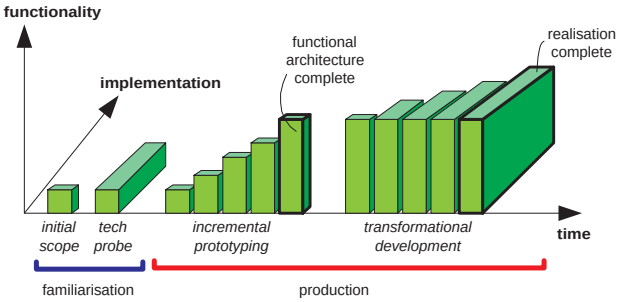

Figure 1. Design dimensions in IRIS

Platform based design [13] recognises the importance of both top-down and bottom-up development dimensions. For Image Processing applications, Bagdanov [3] advocates the separation of development and implementation in large Object Oriented frameworks (Horus). He selected a functional language for application development and C++ for implementation.

From the software economics side it is known since long that two relevant issues influence the choice of a development methodology and in particular of the architectural language. First, the cost of reworking the software is much smaller (by factors up to 200) in earlier phases than later phases [7]. Second, the length of description is the dominant factor in software development costs [6]. The shorter the description the better, giving credit to declarative languages (e.g. functional languages).

None of the above approaches fulfills all requirements mentioned before.

3 The IRIS Methodology

The IRIS design methodology, for deriving firmware for SIMD architectures should support different application domains and should be strongly phased, to allow for the different development roles, see Figure 1. In Section 5 the methodology will be illustrated with a case study. In our methodology we recognise three main phases: I) Familiarisation, II) Incremental prototyping, and III) Transformational development.

I. Familiarisation. The goal of this phase is to come up with a provisionary demarcation of the system boundary and some confidence on the feasibility with respect to the intended hardware. This actually corresponds to the design activities normally deployed between the behavioural and the structural domains in the Y-chart methodology [10]. The physical domain is absent in our approach since we assume

that the (many-core) hardware technology is already available.

We start with the scouting of both the problem (initial scope) and the intended hardware architecture (tech probing). In order to maximise the degree of freedom for system development an abstract 'mathematical' description is made of the formulated problem. At the same time models are made of the target hardware – partly based on sample programs provided by the hardware supplier – to better understand its behaviour. Both activities use the architectural language. At the end of this phase, when sufficient confidence has been built up in both application and hardware architecture, the choice of the hardware is fixed. However, some parameters such as number of processors, clock frequency, size of memories, may change at a later stage. Actual code production consists of the following two phases: *incremental prototyping* and *transformational development*.

II. Incremental prototyping. The goal of this phase is to establish the specification of the system. This phase leads via a number of intermediate steps to a complete specification, the *Functional Architecture*, and to a validation testset, a baseline set used in next phases. This specification is executable –as all the intermediate steps–, is independent of the target hardware, and serves as a live description of the system. The functional architecture marks an important milestone in the customer-architect co-operation. At this point we know the desired functionality of the system and we can turn to the transformational development, which is hardware architecture dependent.

III. Transformational development. The goal of this phase is a satisfactory realisation of the desired functionality on the selected hardware architecture. This phase consists of behaviour preserving transformations (except for the trade-off subphase), which progressively involves making design choices determined by the hardware architecture used, see Figure 1 (right part). The validation testset is progressively extended at the same pace as the functional decomposition. This allows intermediate checking against the current complete validation testset. The Transformational Development phase exhibits the following subphases: *Trade-off*, *Reorganisation*, *Template*, and *Translation*.

Trade-off. The goal of this subphase is to deliver a *golden reference*, which can be used for validation purposes for downstream transformations. Because of hardware limitations often concessions have to be made to the accuracy of computations, bit-width of variables, or even computation speed. Because of possible (mostly tiny) concessions made to the functionality, this subphase involves, besides architect and implementor, also the customer.

Reorganisation. The goal of this subphase is to rephrase the executable model in a top-down manner such that it is more geared towards the chosen hardware architecture. This and following subphases involve only behaviour preserving transformations.

Template. The goal of this subphase is to identify reusable components which can reduce current and future work. These components may consist of common code fragments or even complete modules. The development direction is bottom-up, showing the abstraction of a code fragment (as a template instance) to the template. Both reorganisation and template subphases address the platform based issues [13].

Translation. The goal of this subphase is to realise a smooth transition to the target hardware. This involves a fully automatic translation from the model of the design coded in the architectural language, following the template and all earlier subphases, into the native target language (mostly C+intrinsics) of the chosen hardware.

The unique contribution of IRIS to the field of developing firmware for a SIMD architecture is – to the best of our knowledge – that the framework is: integral, interactive, incremental, domain independent, and utilises a single architectural language that is close to mathematics and understandable by a developer.

The IRIS framework depends heavily on the right choice of this architectural language. The language should be: (a) *flexible*, in the sense that it supports modelling of high level descriptions (close to mathematics) as well as implementation issues as data parallelism or even low level bit field assignments, (b) *compact*, since compactness of description is a virtue in reducing costs (c) *executable*, to offer verifiability of work in all phases, (d) *interpretative*, in order to realise the needed interactivity, and (e) *general purpose*, to allow for creating auxiliary tooling, such as memory utilisation or performance monitoring.

In IRIS we use a functional language (like Haskell [5] of J [16]) as the architectural language because it fulfills the requirements mentioned above. We believe in a single architectural language for all phases that supports multiple roles because of *ease of use*: (1) one single framework is better facilitated by a single language provided the different roles involved can be served adequately, (2) a language close to mathematics facilitates precise specifications, (3) the language should facilitate concise description of implementation details, (4) code refactoring (for example in the Template subphase) is hindered when interfaced over cross-language domains, and (5) one language to investigate *and* document suitable alternatives is more beneficial than using different languages.

It turned out that a functional language best satisfies the above mentioned properties and the requested support for multiple roles.

4 Case study

We illustrate the IRIS methodology with stochastic image quantisation on an SIMD architecture [12] in Section 4.1. Other applications which were developed using IRIS are colour image processing for a printer, mining and visualising document spaces and raster detection, but are not discussed here. In Section 4.2 we present the target SIMD hardware architecture, the Linedancer of Aspex, followed by a functional view on the hardware (Section 4.3).

4.1 Stochastic Image Quantisation

Business graphics are characterised by its use of relatively few colours, and relatively large areas having the same colour. The result of scanning business graphics often shows undesired variations in colour in such single coloured areas. To improve the quality of the scan we use a technique called *stochastic image quantisation* and which is used in combination with *simulated annealing* (see [9]).

The scan process samples an original and returns a matrix of colours of pixels. In the context of this paper we assume, without loss of generality, that all colours are grey-values. The matrix typically has a size of 5000×7000 pixels, whereas grey-values typically fall in the range 0..255. An example of a histogram for grey-values is shown in Figure 2. The objective of image quantisation is to assign all

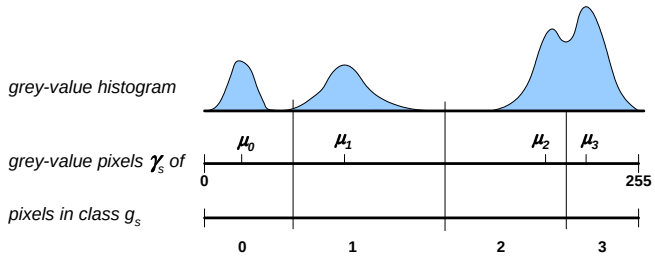

Figure 2. Histogram of grey-values

pixels to a limited number of classes. Let L be the number of classes, in Figure 2 $L = 4$. Let s be a pixel, then γ_s denotes the grey-value of s, whereas g_s denotes the class to which s is assigned. Let $\mathcal{S}$ be the set of all pixels, then the mean of a class c is μ_c:

$$\mu_c = mean\{\gamma_s \,|\, s \in \mathcal{S}, g_s = c\} \tag{1}$$

Stochastic image quantisation [15] now takes (the nearest integer to) μ_c as the best grey-value for class c. However, the question which pixel should be assigned to which class is not so easy to answer. The final answer to that question is determined in an iterative way and depends on a certain quality measure. The method of simulated annealing repeatedly assigns a new class c' to each pixel s in a random way, compares the result with the previous assignment, and chooses the best. A quality function is used to chose between the new class $g_s = c'$ or the old one $g_s = c$. After several iterations this process leads to an optimal quality.

One specific quality criterion per pixel, given the class assignment function g, is the so-called *fidelity*:

$$fid_g(s) = (\gamma_s - \mu_{g_s})^2 \tag{2}$$

Thus, the fidelity of a pixel s is the square of the difference between the actual grey-value γ_s of s and the mean grey-value μ_{g_s} of the class to which s is assigned. The lower $fid_g(s)$ is the better the class mean fits the scanned image pixel.

A second quality criterion is *regularity* which expresses the property of business graphics that relatively large areas have the same colour. That is, regularity indicates how well the grey-value of a pixel fits in its immediate surroundings. Let $s = (i, j)$. Then we define $\mathcal{N}_s = \{(k,l) \,|\, \sqrt{(k-i)^2 + (l-j)^2} \leq R, (k,l) \neq (i,j)\}$ as the *neighbourhood* $\mathcal{N}_s$ of pixel s. Thus, $\mathcal{N}_s$ contains all pixels within distance R from s, except s itself. Let g_r be the class of a pixel in the neighbourhood of s. Then the regularity is defined by:

$$reg_g(s) = |\,\{r \in \mathcal{N}_s \,|\, g_s \neq g_r\}\,| - |\,\{r \in \mathcal{N}_s \,|\, g_s = g_r\}\,| \tag{3}$$

The lower $reg_g(s)$ is, the more uniform the neighbourhood is.

Thus, the quality criterion per pixel, *energy*, combining fidelity $fid_g(s)$ and regularity $reg_g(s)$, is defined by $e_g(s)$:

$$e_g(s) = fid_g(s) + \beta \cdot reg_g(s), \tag{4}$$

where the weight $\beta > 0$ allows for a better, image dependent, quantisation. The value for β is determined experimentally and is in most cases an integer in the range $[1, 100]$ [15]. The quality criterion for the complete image is defined by the matrix $E_g(\mathcal{S})$:

$$E_g(\mathcal{S}) = [\![e_g(s)]\!]_{s \in \mathcal{S}}, \tag{5}$$

This matrix $E_g(\mathcal{S})$ is used by the simulated annealing procedure to produce the final quantisation matrix $g(\mathcal{S})$. Since the value $e_g(s)$ of each pixel must be minimised, *quality* is defined as the negated sum of all energies per pixel:

$$Q_g(\mathcal{S}) = -\sum_{s \in \mathcal{S}} e_g(s) = -\sum_{s \in \mathcal{S}} fid_g(s) + \beta \cdot reg_g(s) \tag{6}$$

This quality is used during development to make motivated choices for various design parameters.

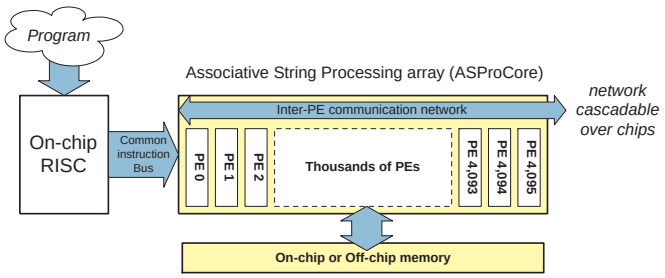

Figure 3. The scalable architecture of the Linedancer

4.2 Linedancer

Aspex's Linedancer [1] is an implementation of a parallel associative processor. The processor contains 4096 simple processing elements in a SIMD arrangement (ASProCore), see Figure 3. The control is centralised in a RISC processor (SPARC). Each of these processing elements (PEs) on the Linedancer device has about 200 bits of memory (of which 64 bits are fully associative) and a single bit ALU, which can perform a 1 bit operation in a single clock cycle. Operations on larger bit-fields, specified by a start location and a field length, take multiple clock cycles. Multiple Linedancer devices can be connected together to create an even wider SIMD array, allowing a scalable solution.

A Linedancer is programmed in an extended version of C, with additional functions for controlling the ASProCore.

The Linedancer processor is chosen because it fits a pixel parallel model well (scalable in number of pixels) and the associative functionality facilitates the necessary table lookups for the quantisation class means.

4.3 Linedancer in a functional perspective

A functional language fits well in describing operations in a SIMD architecture. For example, in the expression `map f matrix` the function `f` is applied to all elements in the given `matrix` in parallel. As a second example, the expression `fold f v list` iteratively applies `f` to the start value `v` and the next element of the list. When all elements of the list are dealt with, the end value is delivered.

The functions `map` and `fold` can be combined in a straightforward manner such that one can easily specify the parallel application of iterative processes.

5 Case-based illustration of the Methodology

In this section we follow the IRIS methodology as outlined in Section 3.

5.1 Incremental Prototyping Phase

After the familiarisation phase (Figure 1), we turn to the stepwise creation of a complete functional model based on the mathematical model of the system as given by the equations (1) – (6). This model can be immediately transcribed in a functional language such as Haskell (see [5]) by defining the corresponding functions:

```
mu c     = mean [ gamma s | s<-S; mem c (g s)]          (1)
fid g s = (gamma s - mu (g s))^2                        (2)
N i j    = [ (k,l) | (i,j) <- S
                   ; sqrt((k-i)^2 + (l-j)^2) <= R
                   ; (k,l) <> (i,j)
                   ]
reg g s = (length [r | r<-N s; g s <> g r]) -
           (length [r | r<-N s; g s == g r])           (3)
e g s    = fid g s + beta * reg g s                     (4)
E g S    = map (e g) S                                  (5)
Q g S    = - sum (map (e g) S)                          (6)
```

Note that, e.g., the grey-value γ_s is transcribed as `gamma s`, where `gamma` is a function, and `s` its argument. Thus, `gamma s` denotes the grey-value of pixel `s` and is generated by the scanning process.

Some further explanation of the notation: `[ e | ... ]` is notation for lists, close to mathematical notation for sets; `mem c x` is a standard function which checks whether `c` is a member of the list `x`; the functions `mean lst` and `sum lst` calculate the mean and the sum (respectively) of the list `lst`. The environment `N i j` of pixel `s=(i,j)` is parameterised by radius `R`.

We remark that this formulation of the model is just a first specification, but already at this stage it is executable for simulation purposes. Thus, instant feedback is facilitated and consequences of this specification can be explored.

5.2 Transformational Development Phase

In this section we illustrate the transformational development phase for the case of stochastic image quantisation.

Trade-off Subphase. In this phase we collect all concessions to the functional architecture to guarantee bit true behaviour for later subphases. This golden reference will guide the validation in the remaining subphases of the development trajectory. To map the algorithm on a Linedancer several implementation concerns have to be considered. Two of them, tiling and accuracy, are described in detail below.

Tiling. Choosing a pixel-per-PE scheme means that a single Linedancer can host a 64×64 tile of pixels. To process larger images we use *tiling*, i.e. we divide the image in small tiles (of 64×64 pixels) that fit on the Linedancer. Tiles need to be fetched with sufficient overlap to enable neighbouring tiles to pass information to each other. A 64×64 tile of

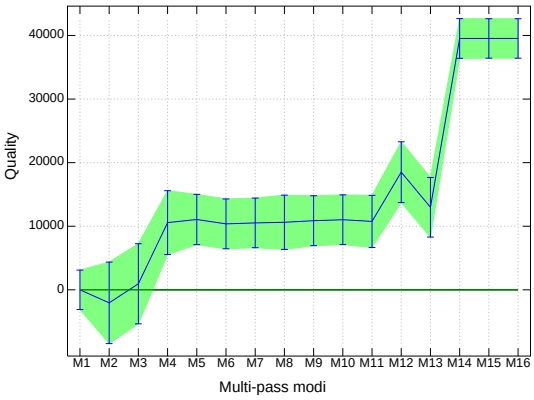

Figure 4. Multi-pass modi for 128 iterations

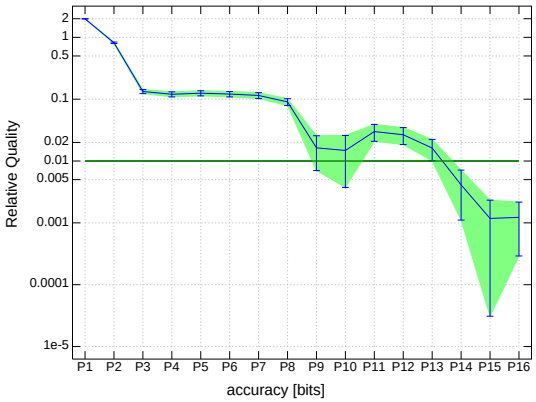

Figure 5. Quality versus accuracy for *fidelity*

pixels is read from memory, next a number of iterations are performed and after that the result is sent back to the memory and the next tile can be fetched. A *pass* is defined as such a single traversal of all tiles through the image, effectively passing on information between neigbouring tiles.

Since the Simulated Annealing procedure requires ± 100 iterations (100 is determined experimentally) a lot of multi-pass modi exist, for example 4 passes of each 25 iterations or 50 passes of 2 iterations. The effect of various multi-pass modi on quality can be quickly determined since the models are executable and the interactive approach allows fast updating. In this way the various design alternatives can be evaluated quickly in an early stage of the design (see Figure 4).

Accuracy. The Linedancer does not support floating point arithmetic. For the various variables an accuracy analysis is made to determine the necessary bit-width in an integer arithmetic scheme. This is important because certain operations like addition, are linearly dependent on the bit-width of variables. For the computation of the optimisation criterion, the fidelity term (2) takes a large bit budget because of the square operation of a subtraction of two 8-bit values. The width of the fidelity bit-field is initially dimensioned to 20 bits. However, an accuracy analysis, directly performed in the model, shows that 14 bits are sufficient for storing the addition result, see Figure 5.

At the end of this subphase the golden reference constitutes the *implemented specification*: all remaining subphases are kept bit true with respect to this reference.

Reorganisation subphase. In this subphase the model is expanded in a top-down manner, gradually adding more details dictated by the hardware architecture. We mention a few issues in this phase.

Precomputation of Constants. Some constants can be computed during the initialisation and e.g. prestored in a particular memory field administered per PE, or in a Look Up Table (LUT) on the Lindancer's control processor. For example, in the definition of fidelity $fid_g(s)$ (see definition (2)), the mean μ_c of a class c is a constant, and thus it is efficient to calculate it only once. We will assume that for every class c the mean μ_c will be stored in a lookup table on the Linedancer control processor (SPARC).

Transformational laws. For every pixel the *fidelity* has to be computed according to the definition

```
fid g s = (gamma s - mu c)^2   where c = g s
```

The simplest way for a programmer of a more general parallel processor (e.g. MIMD) would be to let each PE do all the processing of a single pixel. The procedure that every PE then has to execute is simple as well: just walk through the lookup table until you find your own class means, and then execute the above definition. In terms of the architectural language this means that a `fold` function (applied to the LUT) that is `map`-ped over all PEs representing the pixels in the image. Thus, this simplistic approach would lead to a program that essentially looks like (see Section 4.3):

```
map (fold f v0 lut) image
```

where the `fold` function iterates the function `f` over the lookup table `lut`, and then `f` makes sure that the initial value `v0` is updated with the correct value from `lut`.

However, given the limitations in local computational possibilities and memory size of a SIMD architecture like the Linedancer, individual PEs can not execute such an iterative process. The consequence is that the iteration has to be executed by the control processor, and the relevant data of each class have to be broadcasted to all PEs. Each PE only executes the above definition when its own class `g s` matches with the broadcasted current class index `c` of the LUT. Thus, the control processor performs a `fold` over the LUT, and `map`s the LUT data at each step to all associating PEs. In terms of the architectural language this pattern looks like (apart from some minor formal details):

```
fold (map f PEs) image lut
```

Again, the `fold` -function iterates over the lookup table `lut` , but now it is a "broadcast" function (`map f PEs`) that is iterated, i.e., the function `f` (which took care that a pixel is updated with the correct value) is broadcasted to all pixels. This broadcasting is done for each entry in `lut` and at each step the variable `image` is updated for the relevant part.

Without going into details we remark that there is a precise *law* that transforms the first specification into second one. That is to say, this law transforms a straightforward specification that is very simple to design, into a more complex executable program. Such laws are important to guarantee correctness and therefore play an important role in IRIS. It is one of the advantages of a functional language as architectural language that such laws can be formulated precisely and proven formally.

A second application of the same law is discussed below.

Expression optimisation. In order to take less execution time, each definition has to be checked for possibilities to optimise the computation. For example, definition (3) is straightforward and easy to specify, but the list of neighbours of each pixel has to be travelled through *twice* in order to calculate the respective lengths. The following equivalent definition subtracts or adds 1 when the `g r` is equal or unequal (respectively) to `g s` and travels the list of neighbours only once:

```
reg g s = sum [ if (g r == g s) (-1) (+1) |  r <- N s ]
```

According to definition (4) the outcome of this expression has to be multiplied by the parameter `beta` .

One of the advantages of choosing a functional language as architectural language is that also at early stages in the design process the definitions are executable, thus experiments with real data are possible. A simple experiment showed that the above definition of `reg` can be slightly optimised further (168 versus 180 cycles per pixel, given a 12 pixel neighbourhood $\mathcal{N}_s$ and $\beta \in \{1 \cdots 255\}$) by adding or subtracting this parameter `beta` straightaway. Thus, definition (4) can be replaced by the definition:

```
e g s = fid g s +
          sum [ if (g r == g s)(-beta)(+beta) | r <- N s ]
```

We remark that the equivalence of these definitions can be easily shown.

Transformational laws (2). Again, this definition of `e` has to be broadcasted to all PEs. Note that determining the sum of a list requires an iteration over the list, i.e., we have the same pattern as before: a `fold` inside a `map` . Then clearly the same problem arises, such a specification is not executable on the Linedancer. However, we can apply the same law as before, leading to a `map` inside a `fold` , which *is* executable on the Linedancer.

Template subphase. During this subphase bottom up developments facilitate the discovery of common patterns. This not only includes reusable macros for code fragments or even complete modules but also includes support for instruction coding and translation. As time progresses, *experience* translates into more powerful components (bottom-up). Templates are intelligent pieces of interactive functionality that serves several roles. First of all the functional behaviour of the involved Linedancer instruction(s), *functional emulation*, should be properly modelled. Second, obeying the *calling conventions* for all relevant type of instructions should be enforced. Related to this is the support for *allocating variables* to the scarce memory resources. Programming the Linedancer often involves a lot of shuffling w.r.t. the bit-field specifications (often supported by a spreadsheet). Both the calling convention and the allocation support use a special calling convention for variables to express the allocation of Linedancer memory to the variables. We express this in the variable name as `<name>_<start_position>_<length>` such that memory can be allocated based on these names (call by name). Further details fall outside the scope of this paper. In this way the template is able to serve the three different roles mentioned above, directly from the model code. Finally, the syntax of the template-call should be rich enough to enable automatic generation of the target code for this call (facilitating the translation subphase).

Translation subphase. At the end of the design process the functional models have to be translated into *imperative* code for the Linedancer, written in the architecture specific language Linedancer-C. The specific details of that language fall outside the scope of this paper, hence we restrict ourselves to pseudo code.

In many cases, the functional specifications can be translated straightforwardly into pseudo code. For example, the expression (`arr` stands for an array, `v0` for an initial value)

```
fold f v0 arr
```

translates into

```
a = v0;
forallSeq x in arr do
  a = f(a,x);
```

where the additional variable `a` plays the role of an accumulation variable which contains the required value after termination of the `for` -loop.

Clearly, in this case the `for` -loop goes through the list in a sequential way, as suggested by its name `forallSeq` . The parallel variant is expressed by

```
map f arr
```

and translated into

```
forallPar i in arr_indexes do
  arr[i] = f (arr[i]);
```

where `forallPar` suggests a parallel `for`.

Applying this to the definition of `e` as derived in Section 5.2 yields:

```
forallPar s in S do
  sum_g[s] = 0;
  forallSeq r in neighbours(s) do
    sum_g[s] = sum_g[s] +
                 if g[s] == g[r] then (-beta) else (+beta);
  end for;
  e_g[s] = fid_g[s] + sum_g[s];
end for;
```

In addition to this pseudo code, we again need the special naming convention for variables (as mentioned in the template subphase) to specify the bit-fields in Linedancer-C.

6 Results and Discussion

In this section we will first present and discuss the results of this particular case followed by the results w.r.t. the methodology IRIS for all cases.

6.1 Stochastic image quantisation

A dual Linedancer system running at 300 MHz is 128× faster than a 2 GHz Pentium based implementation. A full A4 page would run in 15 sec on the current Linedancer-P1 system, and 4 sec based on the next generation, currently available, Linedancer-HD processor. The system is scalable, i.e. doubling the number of processors also doubles the performance. A productivity decrease due to larger image sizes – whether caused by increased paper size, resolution, or number of colours – can be repaired by scaling the number of processors.

The development effort is reduced significantly since image quantisation is modelled as an optimisation problem, with a specific perceptual optimisation criterion, that uses a generic optimisation procedure (simulated annealing). Massively parallel embedded processing turn these inherently simple but compute intensive schemes into feasible solutions.

6.2 The IRIS methodology

IRIS has been tested using three very different cases. This section summarises the integral results w.r.t. the methodology.

Both single framework and single language for the whole trajectory provides for a smooth transition through the various phases. The incremental development enables a better traceability of design decisions over time, since the design space explorations were performed when the functional decomposition triggered them. Moreover, interactivity and executability offer an almost instant verification of modelling steps for maintaining quality. In embedded systems making compromises is inevitable; the *trade-off* subphase accomodates this, effectively establishing a *golden reference* model at the end of this subphase. The *template* subphase supports the search for reusable components, that speeds up the development process and adds quality to the design. Finally, design space exploration takes less time because the evaluation of design alternatives can be done in situ. Since the exploration models themselves are available in executable form, a partial redesign of the system will take less time.

From the three cases the following results are obtained w.r.t. the architectural language. A functional language, such as Haskell [5] or J [16], is flexible enough to describe high level as well as low level concepts. More precisely, for the early phases it is close to the mathematical descriptions and is concise to maintain software quality. For the implementation phases it is able to express (massively) parallelism, has array capabilities, and features for modelling hardware concepts. Furthermore, it has graphics capabilities for monitoring the extra-functional properties, in particular in the trade-off subphase.

7 Conclusions

IRIS can be characterised as a confidence-by-construction framework: it offers for the application developer an incremental way of system design, which converges to a target language implementation. Interactivity and executability provide early feedback, in particular on wrong problem interpretation or design faults at early design phases. In case of design changes, models of previous phases can serve as a solid base. Decoupling the development language from the target hardware architecture language offers freedom of choice for migration to different target hardware architectures. Design space exploration and the decision recording during development increases quality and takes less time because the evaluation of design alternatives can be done in situ. All this is realised by using a single language based development framework for the whole development trajectory, and in this way lay the foundation for our integral IRIS framework.

References

[1] Aspex semiconductor: Technology. Website, 2008. http://www.aspex-semi.com/q/technology.shtml .

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[3] A. D. Bagdanov. *Style Characterisation of Machine Printed Texts*. PhD thesis, University of Amsterdam, May 2004.

[4] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2 edition, 2004.

[5] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall Press, 2 edition, 1998.

[6] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, Inc., 1981.

[7] B. W. Boehm. Understanding and controlling software costs (invited paper). In *IFIP Congress*, pages 703–714, 1986.

[8] T. A. C. M. Claasen. System on a chip: Changing ic design today and in the future. *IEEE Micro*, 23(3):20–26, 2003.

[9] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. John Wiley & Sons, Inc., second edition, 2001.

[10] D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[11] H. Goldstein. Winner: Cure for the multicore blues. *IEEE Spectrum*, 44(1), 2007.

[12] J. W. M. Jacobs, L. van Engelen, J. Kuper, and G. J. M. Smit. Image quantisation on a massively parallel embedded processor. In *SAMOS*, pages 139–148, 2007.

[13] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform-based design. *IEEE trans on Computer-Aided Design*, 19(12), December 2000.

[14] P. Lieverse, P. van der Wolf, ed Deprettere, and K. Vissers. A methodology for architecture exploration of heterogeneous signal processing systems. In *Proceedings 1999 Workshop on Signal Processing Systems (SiPS'99)*, pages 181–190, Taipei, Taiwan, Oct. 20–22 1999.

[15] T. Sziranyi, J. Zerubia, L. Czuni, D. Geldreich, and Z. Kato. Image segmentation using Markov random field model in fully parallel cellular network architectures. *Real-Time Imaging*, 6:195–221, 2000.

[16] D. Thomson. *J: The Natural Language for Analytic Computing*. Research Studies Pre, 2001.