Modeling and Instrumentation of Memory Hierarchies using Functional Programming

Tomasz Toczek, Dominique Houzet, Stéphane Mancini

Modeling and Instrumentation of Memory Hierarchies using Functional Programming

Tomasz Toczek Dominique Houzet Stéphane Mancini

***Abstract*— We propose a methodology for the modeling, instrumentation and design space exploration with prototyping of complex designs exhibiting difficult to anticipate performance caveats. It relies on monadic functional programming as a design specification means. Its advantages lie in its model-of-computation (MoC) mixing capabilities, its suitability for both human-assisted and automatic design space exploration, as well as its relative ease of use compared to other functional approaches. We show those aspects on a case study involving memory fetches from a set of tomographic backprojection coprocessors working in parallel.**

I. Introduction

Memory hierarchies have become an essential part of high-performance hardware, yet rely on spatial and temporal locality as well as other often difficult to control aspects of applications' memory reference patterns. It is therefore interesting to wisely choose the structure of a memory hierarchy, and to tune the parameters of the components involved to maximize the performances of the targeted applications. Such parameters include on-chip memory sizes, cache associativities, data replacement policies, possible prefetching mechanisms, and so on. Oftentimes, it is difficult to predict ahead of time which strategy will bring the best results.

A common way to tackle this problem is to specify the design using generic components, and use a set of scripts meant to test which combinations are the most efficient. While often being acceptable, this solution presents several drawbacks. The additional effort required to write the scripts themselves is non-negligible as soon as the design grows too complex for all the parameter combinations to be tested exhaustively. Also, third party components often ship with their own instantiation scripts (this is especially true of caches and networks-on-chip, for instance). As the number of such components involved in a project grows, it becomes increasingly difficult to use them together. Often, the optimal performances are obtained with a subtle combination of parameters of all the components involved that the individual instantiation scripts have trouble to find.

We propose a novel approach based on monadic programming. Alternative implementations and possible parameterizations of the components described by the designer are specified from the ground up. While still allowing heuristic policies for individual component instantiation, the different combinations of alternatives are also meant to be explored design-wide, either be providing adequate feedback to the designer, or automatically through the use of stochastic optimization algorithms. We illustrate the approach through the extraction of relevant design information on a set of tomographic backprojection units, each being connected to a cache. The caches fetch their data from the external memory via an arbiter, which merges their requests if possible. Each of the described components is able to "tune" itself as necessary thanks to the facilities our approach allows.

II. State of the Art

The general reasons for employing functional programming for hardware description [1], [2] include its adequacy to handle the data representations likely emerging during such a task. Pattern matching can significantly ease the expression of graph transformations, and lambda calculus permits to write efficient and flexible combinator-based component libraries. What is more, it is suited for techniques such as staged programming (i.e. execution of code generated at runtime), embedded languages and quasi-quotation (i.e. seamless integration of a user-defined syntax within the language), which find their use quite naturally in the HDL domain or closely related domains such as hardware simulation.

Bluespec System Verilog (BSV) [3] is one of the most well known commercial tools employing functional programming as a foundation. It works on the RTL level, and brings most of the advantages of equational reasoning to hardware design. It is possible to describe very generic and flexible libraries in BSV, thereby increasing code reuse. Resulting designs are concise and little prone to errors, often showing very little amounts of glue logic.

Lava [4] presents itself as a Haskell library, and was an inspiration for our approach on many levels. It makes use of polymorphism and monadic programming as well. A given polymorphic construct can be used for tasks such as simulation, synthesis or verification depending on the actual type it is bound to. Lava designs are implemented on the RTL abstraction level.

ForSyDe [5] allows to specify designs using several models of computation. It typically takes high level specifications as an input and makes use of transformational design refinement to process them. It is discouraged to mix different MoCs within a design using this approach, however.

III. Principles

A. Overview

Our approach is implemented as a Haskell library strongly relying on the language's class system. It has two rather distinct parts:

- A set of classes representing different MoCs (see section III-B), as well as a set of polymorphic functions and types on top of them
- A set of *engines*, that is a set of types implementing the MoC classes, typically making use of the explorable state monad (see section III-C)

The part of the program corresponding to the design description is meant to be expressed polymorphically, using one or more MoCs. This allows its different parts to be interpreted in several ways – as hardware, as software running on a dedicated processor, as a thread on a processor, etc. – and also makes it easier to embed one MoC into another, as we will later see.

A design is bound to an engine depending on what is meant to be done with it. The engines implementing a given MoC differ among themselves by their internal data representation, and hence criteria such as level of abstraction, simulation and performance estimation capabilities and accuracies, code generation capabilities...

B. Models of computation

A model of computation is simply a class providing the designer with the primitives for specifying a design. Those include basic element instantiation, communication means, as well as possible means of integration with other MoCs. For our case study, we chose to use two models of computation: single clock domain register transfer level (RTL) and imperative. It is possible to integrate imperative computations within a RTL design through processes, which behave much like their VHDL counterparts. Common aspects of models of computations, such as the possibility to use combinatorial nodes (ie. signals not needing associated registers), can be factorized by using a common "ancestor" class:

```
class Explorable em => WithAffectation em where
    (#=) :: Transfer ty -> Transfer ty -> em ()

class WithAffectation em => WithNodes em where
    -- ... (combinatorial node handling primitives)

class WithNodes em => Imperative em where
    -- ... (imperative control structures)

class WithNodes em => RTL em where
    process::(forall em'. Imperative em'=>em'())->em()
    -- ... (register instantiation, and so on)
```

Transfer is a type representing either an object, such as a combinatorial node, a register or a variable, or a computation on a set of such objects. A set of arithmetical and logical operators is defined on it. The *ty* argument is a representation of the "type" of the transfer (signed, unsigned, floating point, and so on). Type safety is therefore to some degree enforced by Haskell's typing system, but not entirely. For instance, the burden of checking the operand size consistency during arithmetic operations is on the engine. This is not a severe drawback, since such errors can easily be caught at the elaboration. (Dependently typed languages, such as Ωmega [6], could handle all the checks at compile time.)

Essentially, the models of computation contain the classical control structures one would expect of them. We define "#=", an affectation operator similar in behaviour to the ":=" or "<=" operators in VHDL. Its semantic varies depending on what it is used on. Applied on combinatorial nodes, it specifies the value taken on the next delta cycle. On registers, the value taken on the next clock cycle. On variables, the value change is immediate so it impacts the subsequent statements. Please also take note of the type of *process*, which embeds imperative processes in RTL designs. It underlines the fact that the user is expected to specify their design in a fully polymorphic way.

One key difference between such an implementation and other approaches such as Lava, beside the polymorphism of the user design specification, is the constraint of explorability on all the above classes. Explorable designs are instances of the *Explorable* class, described in the next section. Since even *Transfer* is an instance of *Explorable*, the designer is free to provide alternatives wherever she sees fit. Also, the nesting of MoCs in equivalent to the transformation of an explorable design into another explorable design, bringing a degree of flexibility hard to reach with ordinary instantiation or exploration scripts. What is more, exploration strategies are clearly separated from the design itself, just as they should.

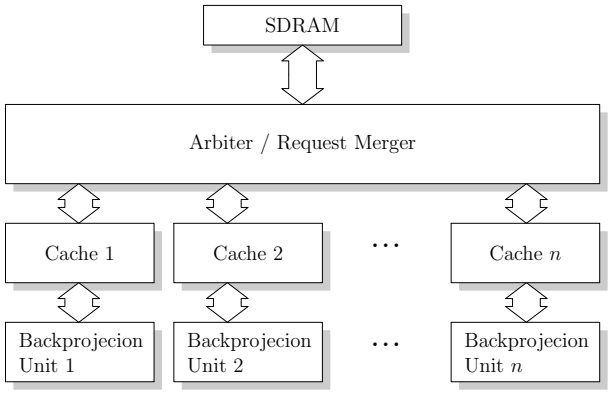
C. Explorable state monad

The explorable state monad is one of the key elements of the proposed approach. From the outside, an explorable state simply offers the possibility to express alternative "implementations" of some monadic computations, and their relative probabilities:

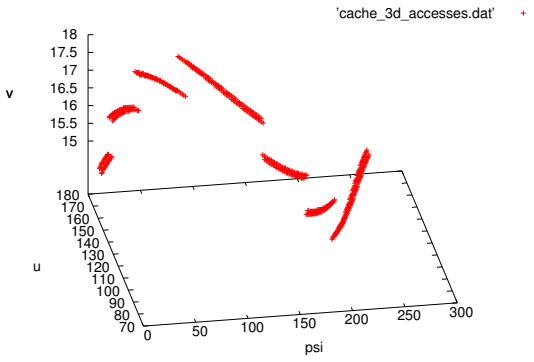
```
class Monad m => Explorable m where
    equiprobable :: [m a] -> m a
    nest::(State s1 a -> m a) -> ExpState s1 a -> m a
```

We also define `ExpState`, a variant of the state monad [7] instancing this class. The state monad used in conjunction with the do-syntax can be seen as a simple way to express sequences of statements in a functional language, defining a transformation of a "state" type. A statement may put a value in the monad (similar to returning a value in imperative languages), and also have some side effects on the state. This is often used to artificially recreate paradigms such as imperative or declarative programming in a functional context. When it comes to HDL design, the possible "state" types would express the design, in part or in whole, at a given abstraction level. Hence, using the explorable state monad instead of the ordinary one makes the design explorable. Our engines (that is, the instances of our MoC classes) are implemented through this explorable state monad:

```
class ExpInfo info where
    eiCall :: (s->(a,s)) -> info -> s -> (info,a,s)
    eiEquiprobable ::
        [ExpState s a] -> info -> s -> (info,a,s)
    eiNested ::
        (State s' a -> ExpState s a)
        -> ExpState s' a -> info -> s -> (info,a,s)
data ExpState s a = ExpState {
    runExpState::ExpInfo info => info->s->(info,a,s)}
instance Explorable (ExpState s) where
    equiprobable alt = ExpState (eiEquiprobable alt)
    nest tr s = ExpState (eiNested tr s)
```



. Fig. 1: Overall design schematic



. Fig. 2: Typical 3D reference pattern for a backprojection unit

Essentially, it is a variant of the state monad whose encapsulated transition function acts not only on the state s, but also on an instance of the *ExpInfo* class.

We also provide a set of exploration functions, by carefully defining some instances of the *ExpInfo* class. Envisaged exploration algorithms range from simple random walk or stochastic hill climbing to simulated annealing or genetic programming. The use of non-stochastic algorithms (such as simple hill climbing or exhaustive exploration) is also possible. Finally, user assisted exploration methods are also implementable. We limited ourselves to exhaustive exploration in order to obtain the results presented in this article.

The automatic optimisation algorithms typically require the user to provide an energy function from s to `Float` to minimize or maximize – possibly a closure. Diverse engines provide diverse kind of such functions, with different accuracy/speed trade-offs.

On important aspect of the proposed methodology is the possibility to mix several MoCs. This is done through nesting (see the $nest$ and $eiNest$ functions in the above listings). As can be seen, the way nesting is implemented makes it in practice virtually impossible to use any other instance of *Explorable* than *ExpState*.

IV. Case study

A. Described Design

We chose positron emission tomographic reconstruction (that is, volume reconstruction from raw MRI scanner data, called a sinogram) as a case study. The overall design schematic used can be found on the figure 1. It is composed of a set of tomographic backprojection coprocessors, their caches and a central arbitration mechanism. Each of the tomographic units processes several voxels, in order to maximize cached data reuse: the computation of each reconstructed voxel value involves 256 memory references to a 3D sinogram [8] stored in the SDRAM, and nearby voxels generate similar reference patterns during reconstruction. Tomographic units process non-recovering yet adjacent sets of voxels, and therefore produce memory references to some degree similar. To make use of this locality, the arbiter not only coordinates each cache's requests to the SDRAM, but also merges the ones that overlap.

The caches themselves are fully associative, but since they are rather small and contain only a few slots (see IV-B for the sizes used), this is not an unrealistic design choice. The sinogram itself is stored as a regular 3D array. The impact of the cache associativity and of the addressing scheme has already been studied [9], so we did not focus on those aspects in this paper.

All of the components were described using the RTL MoC with embedded imperative processes as necessary. In each of the blocks we have selected several performance-related parameters, and set several realistic possible values for each of them through the use of alternatives.

B. Results

As it stands, our approach permits to use the design in several ways:

1) It can be simulated directly using a simulator specified in Haskell (or another language by providing bindings)
2) It can be converted into RTL-level VHDL code, which in turn can be simulated using a third party tool or prototyped on a FPGA. In order for this method to properly work for exploration purposes, this simulation or synthesis needs to be called directly from the exploration function coded in Haskell, and performance data extracted and fed back to the tool. Hopefully, this is easily feasible from an implementation standpoint.

We chose to do the former, but since our simulator is cycle and bit accurate, both approaches would have yield the same results. For larger designs, FPGA prototyping would be a wiser choice.

Several significant design parameters were selected and assessed for their influence on the overall performances. The resulting performance figures are shown on tables I and II. The ranges given are obtained by automatically varying the parameters which are not fixed for a given table cell.

The table I shows that short cache lines perform the best, despite the fact that burst accesses to the SDRAM are *not*

cache line sizes	cache sizes		
	128	256	512
64	0.33 ... 0.60	0.33 ... 0.60	0.33 ... 0.60
128	0.19 ... 0.44	0.25 ... 0.44	0.25 ... 0.44

TABLE I

EFFICIENCY (AVERAGE NUMBER OF FETCHES PER CYCLE) DEPENDING ON THE CACHE PARAMETERS

block sizes	merging policy	
	only grouping	actual merging
$2 \times 2 \times 1$	0.19 ... 0.46	0.16 ... 0.54
$2 \times 2 \times 2$	0.16 ... 0.51	0.16 ... 0.51
$2 \times 4 \times 1$	0.16 ... 0.46	0.22 ... 0.6
$2 \times 4 \times 2$	0.12 ... 0.42	0.12 ... 0.42
$4 \times 2 \times 1$	0.19 ... 0.46	0.28 ... 0.57
$4 \times 2 \times 2$	0.12 ... 0.33	0.12 ... 0.33
$4 \times 4 \times 1$	0.19 ... 0.49	0.19 ... 0.49
$4 \times 4 \times 2$	0.12 ... 0.38	0.12 ... 0.38

TABLE II

EFFICIENCY (AVERAGE NUMBER OF FETCHES PER CYCLE) DEPENDING ON THE ARBITER BEHAVIOUR AND TOMOGRAPHIC UNIT BLOCK SIZE

pipelined in our setup. It can also be seen that 512 byte caches perform no better than their 256 byte counterparts. This is due to the fact that the back projection units iterate on *psi* (as seen on figure 2), and never "go back". Reuse mostly happens when a given sinogram element is required for several voxels in the block a backprojection unit reconstructs. If the cache is too small and overwrites those elements before they can be taken in account for all the voxels it impacts, trashing happens (one 128-byte slot cache case on the table I).

The table II is much more interesting because less predictable. First of all, we put some additional logic int the arbiter allowing it to detect overlapping requests, and fetch the data corresponding to those requests in a single burst. As can be seen, this bit of logic tends to help; given that the SDRAM bursts are not pipelined as is, this was to be expected. When this logic is removed, only *exactly identical* requests are detected, and grouped into a single burst. The block size parameter exhibits a somewhat stranger behavior. To keep things fair, when varying the block size, we also vary the number of tomographic units, in such a way that the whole system always computes the same overall block of $4 \times 4 \times 2$ voxels. Increasing the block size of each backprojection unit increases to locality, while adding additional units increases parallelism. It is not immediately obvious which strategy is the most efficient. The table II shows the best trade offs are 4 units with $2 \times 2 \times 2$ blocks with "simplified" request merging, and $2 \times 4 \times 1$ blocks with "advanced" request merging. Situations of this kind arise frequently, and that very fact is one of the reasons for using approaches such as the one described.

V. PERSPECTIVES

So far, we have merely evaluated a few parameter combinations and introduced a limited number of variable design parameters. However, on a setup such as the one used for our tests, a higher number of alternatives could have been specified:

- possible set associative caching (as done in [9])
- alternative sinogram addressing schemes (as done in [9])
- the choice of the permutation of coordinates for sinogram indexation should be left the tool
- the arbiter could be further simplified by possibly removing the request grouping logic
- several content replacement policies could have been implemented in the cache model
- several prediction mechanisms could be implemented in the cache model
- the whole cache hierarchy could have been possibly made 3D aware

While keep a small number of variable parameters can be justified in order to ease the interpretation of the results, it would be equally interesting to implement a great number of those parameters as alternatives, and try and stress stochastic exploration algorithms using prototyping on a very large design space. This is what we aim for in the short term.

VI. CONCLUSION

We have presented a methodology allowing to describe, parametrize and instrument IPs in a design. Its strong points reside in the easy specification of alternative implementations of parts of the design, hereby efficiently dealing with difficult optimization cases arising in complex systems. The alternative implementations are effectively "hidden" in the hardware blocks they concern, making it possible to design efficient black boxes capable of adapting themselves to the context they will be used in. The approach does not impose any particular level of abstraction, and several models of computation can be seamlessly mixed within a design. Future prospects include the evaluation of stochastic optimization methods on very large design spaces expressed in such an approach.

REFERENCES

[1] T. Sheard, "Types and hardware description languages," in *Hardware Design and Functional Languages, A satellite event of ETAPS*, A. Martin, C. Seger, and M. Sheeran, Eds., vol. 5161, 2007, available from: `http://www.cs.pdx.edu/~sheard/` .

[2] M. Sheeran, "Hardware design and functional programming: a perfect match," vol. 11, no. 7, pp. 1135–1158, 2005.

[3] Bluespec, http://www.bluespec.com/.

[4] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in haskell," in *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM Press, 1998, pp. 174–184.

[5] I. Sander and A. Jantsch, "System modeling and transformational design refinement in ForSyDe," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 1, pp. 17–32, January 2004.

[6] T. Sheard and N. Linger, "Programming in omega," in *CEFP*, ser. Lecture Notes in Computer Science, Z. Horváth, R. Plasmeijer, A. Soós, and V. Zsók, Eds., vol. 5161. Springer, 2007, pp. 158–227. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88059-2_5

[7] M. P. Jones, "Functional programming with overloading and higher-order polymorphism," 1995.

[8] N. Gac, S.Mancini, M.Desvignes, and D.Houzet, "High speed 3d tomography on cpu, gpu and fpga," *EURASIP Journal on Embedded systems*, vol. Special issue : Design and Architectures for Signal Image Processing (to be published), 2008.

[9] T. Toczek, D. Houzet, and S. Mancini, "Another take on functional system-level design and modeling," in *Forum on specification and Design Languages*, 2009.