Aspect Driven Compilation for Dataflow Designs

Paul Grigoraş[*], *Xinyu Niu*[*], *Jose G. F. Coutinho*[*], *Wayne Luk*[*], *Jacob Bower*[†] and *Oliver Pell*[†]

[*] Department of Computing, Imperial College London, 180 Queen's Gate, London SW7 2AZ, UK

[†]Maxeler Technologies Ltd, 1 Down Place, London W6 9JH, UK

Email: {paul.grigoras09, niu.xinyu10, gabriel.figueiredo, w.luk}@imperial.ac.uk, {jacob, oliver}@maxeler.com

***Abstract*—This paper proposes a novel hardware compilation approach targeting dataflow designs. This approach is based on aspect-oriented programming to decouple design development from design optimisation, thus improving portability and developer productivity while enabling automated exploration of design trade-offs to enhance performance. We introduce FAST, a language for specifying dataflow designs that supports our approach. Optimisation strategies for the generated designs are specified in FAST, making use of facilities in the domain-specific aspect-oriented language, LARA. Our approach is demonstrated by implementing various seismic imaging designs for Reverse-Time Migration (RTM), which have performance comparable to state-of-the-art FPGA implementations while being produced with improved developer productivity.**

I. Introduction

Existing work shows that dataflow machines emulated on FPGAs achieve performance gains of up to several orders of magnitude compared to traditional control based architectures [1], [2]. By eliminating the fetch-decode-execute cycle of von Neumann architectures [3], dataflow designs require less area for caching and control which increases performance and decreases power consumption. Due to its regularity, a dataflow design can be statically scheduled into a deep hazard-free pipeline through which data are streamed, achieving ideal throughput rates of one result per clock cycle.

The stream based execution model is well suited for implementing high throughput applications that operate on large amounts of uniform data streams. However, dataflow languages are not commonly used and imperative languages such as Java, C and C++ are popular [4]. The following challenges should be overcome to facilitate the adoption of dataflow designs:

1) Specifying dataflow designs, in an intuitive, well understood language that is concise, facilitates the translation of existing designs, and is sufficiently expressive to support the requirements of modern high-performance applications.
2) Specifying optimisation strategies, decoupled from the application code in a manner that makes the specification easy to reuse and to customise, and is comprehensive enough to allow capturing of optimisations at various levels: *algorithmic transformations* of the original application that expose parallelism or improve communication between CPU and accelerator, *design-level transformations* that enable exploration of platform specific optimisations and *productivity related transformations* that improve developer productivity.
3) Systematic design space exploration of dataflow designs driven by these parameterizable optimisation strategies, that increase developer productivity and allow exploration of design level trade-offs.
4) Applying these design techniques to create and optimise high-performance applications.

We propose a methodology for addressing these challenges based on the following contributions:

1) We introduce FAST, a dataflow language based on C99 syntax that can be used to create high-performance designs. We implement a compiler that translates designs in FAST to MaxCompiler [5] designs which are then compiled and executed on a Maxeler MaxWorkstation containing a MAX3 DFE with a Virtex 6 FPGA chip and 24GB of DRAM.
2) We present novel aspects for specifying optimisation strategies at the system level, the implementation level, the exploration level and development level. We implement these aspects using LARA [6], an aspect oriented language for embedded reconfigurable systems.
3) We propose an automated method for design space exploration of FAST dataflow designs, driven by the aspect definitions.
4) We evaluate our approach by implementing a high-performance design for an application based on the Reverse Time Migration technique for seismic imaging.

II. Design Flow

We propose a novel design flow for aspect-driven compilation of dataflow designs to meet the following requirements and design goals, addressing key areas in developing hardware acceleration solutions:

1) *Performance:* specify high-performance dataflow designs, that achieve significant speedup over sequentially executed implementations; exploit the reconfiguration capability of FPGA devices to improve performance and efficiency;

2) *Portability:* improve portability of dataflow designs, to allow reuse of optimisation strategies on various platforms;
3) *Integration:* simplify translation of existing applications to high-performance dataflow designs to facilitate the integration of the proposed design flow with existing (predominantly imperative) application code;
4) *Productivity:* improve developer productivity by providing high-level means of specifying dataflow designs and controlling compilation strategies that reduce compilation time and generating boilerplate code automatically.

To meet these requirements we propose the following approach. Firstly, we introduce FAST (described in Section III), a novel language for specifying dataflow designs. We specify the accelerated portion of the original applications using FAST dataflow kernels. By maintaining compatibility with C99 syntax we improve developer productivity by providing a familiar language and introduce the possibility of combining hardware and software specifications. Secondly, by using an aspect driven compilation flow we decouple optimisation from design development, improving design portability, and we automate the generation of code and design space exploration improving productivity. Finally, systematic design space exploration is used to identify maximum performance configurations, subject to platform specific constraints.

The proposed design flow is illustrated in Fig. 1 and follows the steps:

1) a C application containing an embedded high-level dataflow design is developed from the original source application. The design is implemented using FAST as described in Section III;
2) the dataflow design is transformed by the aspects in the repository to generate new designs (e.g. with multiple word-length configurations). The classes of aspects used with our approach are introduced in Section IV;
3) the generated configurations are compiled using a back-end compilation toolchain (currently MaxCompiler) to dataflow designs implemented on FPGAs;
4) the feedback from the compilation process is used to drive the design space exploration, repeating the weaving and compilation process until user specified requirements are met.

Compared to existing work described in [7] and [8] our approach emphasises and provides more freedom in the exploration of design level optimisation (such as word length optimisations and mapping of arithmetic blocks to DSPs) by using a combination of implementation aspects (shown in Fig. 1) and FAST optimisation options.

Additionally, our approach targets a dataflow architecture as opposed to the von Neumann architecture proposed in related work, which typically includes a General-Purpose Processor (GPP) and a custom accelerator. We consider additional optimisations to achieve performance improvements as a result of a systematic design space exploration process.

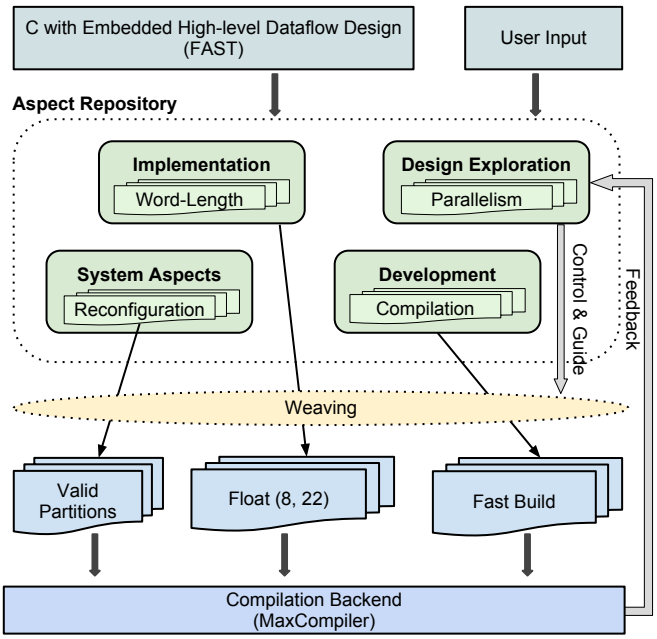

Fig. 1. Proposed approach for aspect-driven compilation of dataflow designs.

III. The FAST Language

FAST (Facile Aspect-driven Source Transformation) is a novel language for specifying dataflow designs that are used as a starting point for the design flow proposed in Section II. In particular, we use C syntax to capture dataflow computations, and, instead of heavily relying on API libraries to specify the design (as in MaxCompiler [5] or Streams-C [9]), we use aspects to implement the transformations required for the actual implementation.

FAST provides the following features that are required by the proposed flow:

- *Imperative specification of dataflow designs*. C99 syntax is enforced by the FAST compiler which is based on the ROSE Framework [10]. The familiar syntax makes the language easy to adopt thus facilitating translation of existing implementations to dataflow designs.
- *Good integration with existing source level translation and weaving tools*. Simple syntax allows the language to interact well with existing compilers or source to source translation frameworks, allowing source level optimisations to be applied through different tools.
- *Combined hardware/software design*. Specifications of dataflow kernels and CPU run-time software can be mixed. The example shown in Fig. 2 can be compiled with the GCC toolchain, but when using the FAST compiler, the pragma indicates the link between the software and hardware, which results in an accelerated hardware/software solution.
- *Support for data path and control path generation*. FAST allows specifying both data and control operations that are automatically mapped to stream multiplexers.

FAST is used to express the simplest form of a dataflow design while optimisations and other transformations are encapsulated in aspects which are developed separately and applied through aspect weaving. This results in a flexible approach for generating and exploring the space of efficient dataflow designs.

Designs in FAST are compiled to MaxCompiler designs composed of inter-connected functional kernels. Communication between kernels is asynchronous, so they can operate independently, and compute only when all active inputs have available data.

Table I summarises the features of FAST and Fig. 2 shows an example dataflow kernel used to value European options. Kernels are defined as regular C functions with inputs clearly defined as arguments in the function signature. Streams are represented as regular C99-style pointers. Normal array notation can be used to generate either previous or future values or dereference the stream to obtain the current stream value. Negative indices are allowed for accessing previous stream values and supported offset expressions are linear expressions comprised of constants or variables (either loop induction variables, or normal variables but for which a compile time range of values is specified, as a requirement for generating efficient hardware). Constructs such as loops are supported as long as their bounds are known at compilation time and are used to parametrise dataflow designs.

TABLE I
SUMMARY OF THE MAIN FEATURES OF THE FAST LANGUAGE.

Feature	Description	Method (see Fig. 2)
Input/Output	Declared in function header	C99 (line 1)
	`in()`,`out()`	FAST API (lines 2,11)
Control	Ternary op., `if` statement	C99 (line 11)
	Stream mux (`mux()`)	FAST API
Computation	+, *, /, -	C99 (line 8)
	log, exp, sqrt, sin etc.	#include <math.h>
Streams	Declared as pointers	C99 (line 1)
	Accessed with array index	C99 (line 8)
Optimisation	C pragmas	C99 (line 7)
Parameterization	Constants, variables	C99
Hardware Mapping	C pragmas	C99 (line 17)

The FAST API provides higher-level constructs such as I/O functions (`in()`, `out()`), counters (Fig. 2, Lines 4–5) and functions to multiplex streams (`mux()`). C function calls are mapped to dataflow kernels via pragmas (Line 17) which provides the flexibility of selecting a particular dataflow configuration based on run-time conditions. Support for run-time reconfiguration is included but will be described in a future publication due to lack of space.

```
1   void Price_FPGA(float* p, float c_0_0_0,
        float c_p_0_0, float c_n_0_0, int n1,
        int ORDER) {
2     in(p);
3
4     float* i4 = count(1000, 1);
5     float* i1 = countChain(n1, 1, i4);
6
7     #pragma FAST DSPBalance:full
8     int result = p[0]  * c_0_0_0 + p[1]  *
          c_p_0_0 + p[-1] * c_n_0_0;
9
10    int up =(i1 >= ORDER) && (i1 < n1 - ORDER);
11    out(up ? result : p);
12  }
13
14  void Price_CPU(...) {...}
15
16  int main() {
17    #pragma FAST hw:Price_FPGA
18    Price_CPU(...);
19  }
```

Fig. 2. FAST dataflow kernel for European Options pricing.

IV. ASPECTS

Aspects are standalone modules that capture functional cross-cutting concerns that are decoupled from the primary function of a program. AspectJ [11], a widely used extension for Aspect-Oriented Programming (AOP) in Java, captures program execution points (such as method calls) at run-time to allow new code to be executed before, after or in place of these execution points through a process called *weaving*. The main motivation behind AspectJ in particular, and AOP in general, is to solve the modularisation problem when dealing with multiple cross-cutting functional concerns.

The LARA aspect-oriented design-flow [6], on the other hand, performs the weaving process at compile-time to satisfy non-functional concerns, such as to improve performance on a particular hardware platform. For this purpose, the LARA weaving process manipulates and transforms the application sources. These new generated sources (woven code) incorporate functional elements of the original sources, and non-functional concerns captured by LARA aspects.

In this paper, we combine the LARA aspect design-flow with FAST dataflow designs. In particular, FAST uses standard C99 syntax to capture dataflow computations while aspects specify decoupled optimisation and transformation strategies that operate on FAST descriptions. This approach makes the functionality of the application easier to understand, more maintainable and portable since it is no longer obscured by various structural or algorithmic transformations, as well as platform specific optimisations. In addition, strategies coded in LARA can be re-applied automatically in different applications, thus improving developer productivity.

We report four types of aspects (Table II) used with designs in FAST:

System Aspects. System aspects capture transformation or optimisation strategies that affect the whole application such as

those concerning hardware/software partitioning, monitorisation and run-time reconfiguration capabilities. The goal of hardware/software partitioning is to improve the overall execution time by identifying parts of the code to be offloaded to hardware (Section IV-A). Monitorisation aspects instrument the application code to extract run-time behaviour, and uncover opportunities for optimisation (Section IV-B). Run-time reconfiguration can be used to remove idle functions from the accelerator at specific points in time, so that additional resources can be dedicated to functions that are active [12].

Implementation Aspects. Implementation aspects focus on low level design optimisations that can be applied to designs in FAST to improve timing or resource usage. For instance, operator optimisation aspects (Section IV-C) can be used to map operators in the program to dedicated hardware resources. Word-length aspects specify the numerical representation of variables and expressions in the design.

Exploration Aspects. Exploration aspects deal with strategies that generate multiple designs to find an optimal implementation based on user requirements. Exploration aspects can act on any level of the design flow (C code, C and FAST, or FAST functions). They enable systematic exploration of trade-offs and optimisation opportunities. Examples of exploration aspects include iterative aspects (Section IV-D) which generate a sequence of solutions until a termination criterion is satisfied, and metaheuristic-based aspects to find optimal solutions in a very large design space.

Development Aspects. Development aspects capture transformations that have an impact on the development process such as debugging (Section IV-E), and, potentially, simulating kernels or improving compilation speed. Separating these concerns makes the original code easier to maintain and enables the automatic application of these transformations to a wide range of designs, improving developer productivity. Simulation aspects could be applied to dataflow designs to generate equivalent state-based C code thus enabling pure software simulation. Compilation aspects, on the other hand, could be applied during the development process to create versions of the dataflow design that compile faster by reducing the operating frequency, removing debug blocks or applying design-level optimisations that can resolve timing constraints. Naturally, reducing the compilation time would increase developer productivity.

A. Hardware/software partitioning

FAST functions describing dataflow computations can be embedded within the C application but cannot be invoked directly by software C functions. Instead, a FAST pragma must be used on top of software function definitions or C calls to indicate an alternate hardware implementation. For instance, the code shown in Fig. 3 indicates that the software implementation of `f()` can be mapped to the dataflow implementation described in `fast_f()`. This way, our design-flow can automatically switch from a pure software application to a software/hardware design.

TABLE II
TYPES OF ASPECTS USED IN FAST

Aspect Type	Aspect Name	Description
system	• hw/sw partitioning • monitorisation • reconfiguration	capture mapping between application modules and GPP/FPGA accelerators
implementation	• operator optimisation • word-length spec	capture low-level hardware optimisations
exploration	• iterative • metaheuristic	generate multiple implementations based on design space exploration strategies
development	• simulation • debugging • compilation	improve developer productivity

```
1  void fast_f() {/* dataflow implementation */}
2
3  void      f() {/* software implementation */}
4
5   #pragma FAST hw:fast_f
6  f();
```

Fig. 3. Mapping of C function calls to dataflow kernels using FAST pragmas.

Hence, a hardware/software partitioning strategy can be performed in five steps:

1) detecting hotspots in the program;
2) detecting code patterns from hotspots that are suited for dataflow computation and acceleration;
3) performing the outlining transformation so that each candidate for acceleration is enclosed in a function `f`;
4) deriving a dataflow version `fast_f` from state-based `f`;
5) placing a FAST pragma on top of each function call to `f` and associate it to the corresponding `fast_f` function.

Each of these steps can be described as a separate LARA aspect and combined to form a hardware/software partitioning strategy.

B. Monitorisation Aspect

To find potential hotspots in the application, for instance to perform hardware/software partitioning, we can use the aspect in Fig. 4. With this aspect, the weaver can automatically instrument any C application to self-monitor its innermost loops at run-time, as they are natural candidates for dataflow-based acceleration. In particular, this monitorization aspect can compute the following information for every innermost loop: (a) the average number of times it has been executed, (b) the average number of iterations, (c) the loop average time, and (d) the loop iteration average time. For this purpose, we use a monitoring API composed of 4 functions to mark the beginning and end of the loop (`monitor_instanceI` and `monitor_instanceE` respectively), and to mark the beginning and end of an iteration (`monitor_iterI` and

monitor_iterE respectively). These monitoring functions keep an account of the frequency of execution and the time to complete the whole loop and a single iteration.

```
1  aspectdef LoopMonitor
2  select function.loop{is_innermost} end
3  apply
4      $loop.insert before
5          %{monitor_instanceI("[[$loop.key]]");}%;
6      $loop.insert after
7          %{monitor_instanceE("[[$loop.key]]");}%;
8  end
9
10 select function.loop{is_innermost}.entry end
11 apply $begin.insert after
12         %{monitor_iterI("[[$loop.key]]");}%;
13 end
14 select function.loop{is_innermost}.exit end
15 apply $begin.insert before
16         %{monitor_iterE("[[$loop.key]]");}%;
17 end
18 end
```

Fig. 4. Aspect that instruments the application to monitor loop activity. The information generated can be used to identify hotspots.

The aspect code is as follows: line 2 selects all loops in the application that are innermost (loops with no other loops enclosed); lines 3–8 place an instance monitor call before and after each selected loop; lines 10–13 select all entry points inside the loop and insert a monitoring call to mark the beginning of each iteration; lines 14–17 place an instance monitor call to mark the end of each iteration. The following table shows an example of applying the aspect from Fig. 4 on a C-style function containing a loop:

original code	**woven code**

`void f() {`
 	
`  while (i < N) {`
 	
`  i++;`
 	
`  }`
 	
`}`
	`void f() {`
	`  monitor_instanceI("f:1");`
	`  while (i < N) {`
	`  monitor_iterI("f:1");`
	`  i++;`
	`  monitor_iterE("f:1");`
	`  }`
	`  monitor_instanceE("f:1");`
	`}`

Each monitoring call in the woven code receives as a parameter the loop key, which uniquely identifies the loop within the application. The loop key is generated by concatenating the function name with the hierarchical position of the loop within the abstract syntax tree. For instance, *f:2:1* corresponds to the 1st loop inside the 2nd outermost loop of function `f`. The hotspots can be identified by an aspect (not shown) that takes the profiling information generated by the monitorization API calls, and that uses an heuristic to compute the most profitable computations to be offloaded to hardware.

C. Operator Optimisation Aspect

To provide architectural details to FAST designs, such as mapping operators to DSP blocks, we can use the FAST pragma shown in Fig. 5 at the top of a statement (including code

```
1  #pragma FAST balanceDSP:balanced
2  {
3    x = x * y;
4    x++;
5  }
```

Fig. 5. The FAST balancing pragma provides fine grained control over the mapping of computation to either DSPs or LUT/FF pairs.

blocks). The balancing parameter corresponds to the degree of utilisation of DSP blocks in a statement.

The aspect shown in Fig. 6 is the strategy for balancing DSP blocks in every statement of an application. Instead of adding the above pragma manually, we provide a set of rules (lines 3–4) that define where to place the `balanceDSP` pragma. In this example, we established the rule that full DSP block utilisation is applied to any statement that has 5 or more multipliers and adders, balanced if 3 or more multipliers, and no DSP utilisation otherwise.

```
1  aspectdef DspBalancing
2  var op_granularity =
3   [{DspBalance:'full',MultiplyOp: 5,AddOp: 5 },
4    {DspBalance:'balanced',MultiplyOp:3}];
5
6  select function.statement end
7  apply
8     for (var i in op_granularity) {
9        var gprofile = op_granularity[i];
10       var match = true;
11       for (var k in gprofile) {
12          if (k != 'DspBalance') {
13             match &= ($statement.num_construct(k)
14                      >= gprofile[k]);}}
15       if (match) {
16          var pragma = '#pragma FAST balanceDSP:'
17                      + gprofile.DspFactor;
18          $statement.insert before "[[pragma]]";
19          break;}}
20    end
21 end
```

Fig. 6. Aspect for exploring mapping of computation to DSP blocks.

D. Iterative Aspect

Using LARA we can implement and combine these aspects to enable systematic design space exploration of all the optimisation options exposed by the FAST backend resulting in the generation of a large number of designs. The feedback-directed compilation process of LARA can be used to capture and extract feedback from the backend reports pertaining to resource usage or timing information and automatically adjust the compilation process.

An example of a LARA aspect for design space exploration is shown in Fig. 7. It highlights the feedback capabilities of the design flow: the aspect will generate and build the FAST designs until the resource usage passes a specified LUT threshold, and at each step increasing a particular design attribute, such as exponent, mantissa or the parallelism of the design (by replicating the computational pipeline).

```
1   aspectdef DesignExploration
2   input
3      attribute,
4      start, step,
5      lut_threshold,
6      config
7   end
8   config[attribute] = start;
9   var i = 0;
10  do {
11    var designName = genName(config);
12    call genFAST(designName, config);
13    buildFAST(designName);
14    config[attribute] += step; i++;
15  } while (@hw[designName].lut < lut_threshold
16          && i < LIMIT);
17  end
```

Fig. 7. Exploration aspect that generates multiple FAST designs by varying a design attribute (e.g. number of kernels or mantissa) until a LUT threshold is reached.

E. Debugging Aspect

Because the current execution model does not provide run-time debugging of hardware designs, the easiest solution to debug designs is to log the values of various streams during execution. The insertion of debug statements can be encapsulated in aspects. It is particularly important to separate debug aspects from the original application code since debug blocks can influence the compilation time and timing constraints as well as the behaviour of the design. As an example, the aspect in Fig. 8 instruments the code to log every change in the value of a variable.

```
1  aspectdef WatchVar
2  select function.vref end
3  apply
4     $vref.parent_stmt.insert before
5      %{ log("[[$vref.name]]", [[$vref.name]]); }%
6     $vref.parent_stmt.insert after
7      %{ log("[[$vref.name]]", [[$vref.name]]); }%
8  end
9  condition $vref.is_out end
10 end
```

Fig. 8. Aspect for automatically instrumenting the code to watch any change in the value of a program variable.

V. Evaluation

A. RTM Implementation

We evaluate the proposed approach by implementing a high-performance application based on the Reverse Time Migration method for seismic imaging which is used to detect geological structures, based on the Earth's response to injected acoustic waves. The technique models the propagation of injected waves using the isotropic acoustic wave equation [13]:

$$\frac{d^2p(r,t)}{dt^2} + dvv(r)^2 \bigtriangledown^2 p(r,t) = f(r,t) \qquad (1)$$

We approximate the differential equation using stencil computation to perform a fifth-order Taylor expansion in space and first-order Taylor expansion in time.

We use FAST to implement the dataflow kernels and aspects to generate multiple configurations for the design by creating two kernels that are used to control the memory command read and write streams (CmdRead, CmdWrite) and the computation kernel (RTM).

To illustrate the potential benefits of our approach we analyse the results of using the debugging aspect of Section IV-E. Table III compares the number of lines of code required for the FAST with aspect design with the equivalent MaxCompiler implementation showing a reduction in code size of up to 42% for the run-time reconfigurable design and a reduction in the number of API calls (including debug calls) of up to 67% which translate to increased productivity.

TABLE III
Code measures for the RTM kernels comparing FAST and MaxCompiler.

Kernel	Aspect	FAST		MaxCompiler	
	LOC	LOC	# API calls	LOC	#API Calls
CmdRead	12	26	6	59	39
CmdWrite	12	28	39	79	56
RTM Static	12	246	43	403	175
RTM RTR	12	377	91	669	275

B. Results

Results of the design space exploration using the aspect in Fig. 7 with variable mantissa illustrate the trade-offs between accuracy and resource usage (Fig. 9). We observe irregular, large variations when decreasing the mantissa from 18 to 16 and 24 to 22 which is the effect of the backend tools mapping arithmetic to a combination of both DSPs and LUT/FF elements. The mantissa boundaries at which this optimisation occurs are platform specific, depending on the architecture of the DSPs. Hence, automating this optimisation via aspects and decoupling it from the original source code makes the application more portable and facilitates discovery of interesting trade-off opportunities using design space exploration.

The DSP balancing aspect shown in Fig. 6 allows to explore the resource trade-offs of implementing arithmetic operations in either DSPs or LUTs and FFs (Fig. 10) and helps to avoid over mapping on DSPs for arithmetic intensive applications.

Design space exploration using the aspect in Fig. 7 with increasing parallelism level can be used to investigate design scalability. For example, for the described RTM implementation, Fig. 11 shows that performance scales linearly with the number of parallel pipelines and that significant speedups can be obtained by the FAST dataflow design compared to the CPU only implementation. Depending on the problem size, our approach can be used to achieve a significant speedup over software only versions which is comparable with the best published FPGA results for static designs [12], [13].

Fig. 11 also shows a model of the performance benefits of using a run-time reconfigurable implementation generated using

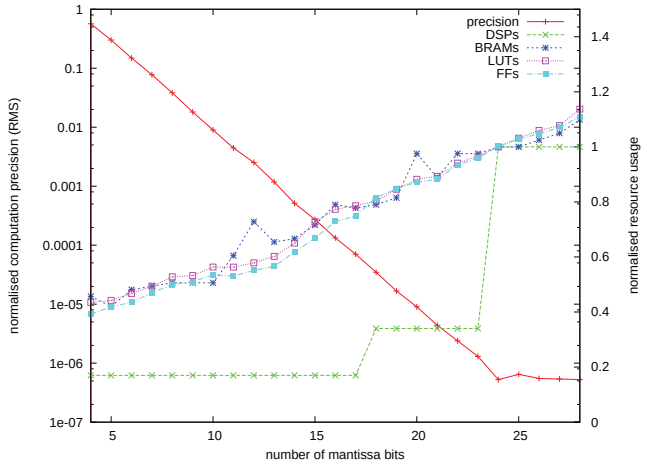

Fig. 9. Exploration of accuracy vs resource usage trade-offs using the aspect shown in Fig. 7 with variable mantissa.

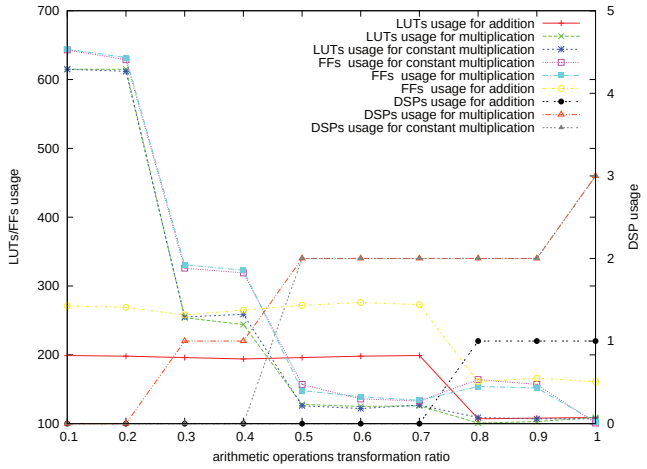

Fig. 10. Exploration of DSP and LUT/FF balancing for functional units implementing a single arithmetic operation using the aspect shown in Fig. 6.

the proposed aspect-oriented approach. Two configurations were created for the RTM FAST kernel. Since, in our model, during the first half of the execution time, the backward propagation and imaging functions are idle, the first configuration requires only half the resources. Hence, the number of parallel pipelines can be doubled, halving the execution time of the first configuration. The speedup obtained is comparable to [12], but the partitioning and optimisation exploration process is automated via aspects, which increases developer productivity. The automated process improves portability of the design, allowing optimisations based on design space exploration to be carried out on various platforms (hence subject to varying resource constraints) without manual intervention.

VI. Related Work

A number of dataflow languages have been developed targeting FPGAs but also multi-core platforms. Table IV summarises some of the important features of these languages

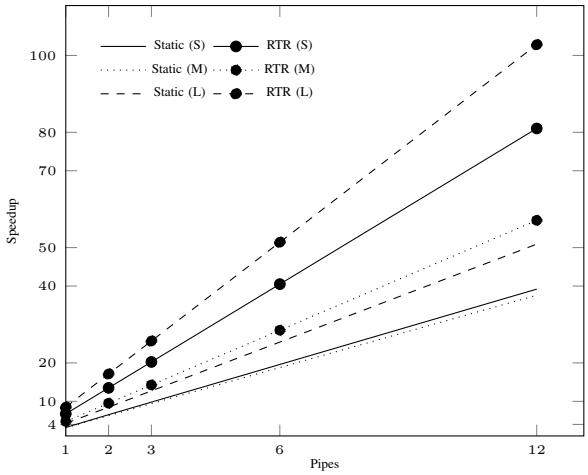

Fig. 11. Scalability of the RTM dataflow design explored using the aspect shown in Fig. 7.

compared to FAST.

Lucid [14], SISAL [15], [16] and Lustre [17], are examples of functional dataflow languages. The latter is based on a synchronous programming model, facilitating safety verification for critical software [18] rather than performance. The functional programming style complicates the translation of existing imperative applications and none have existing implementations for FPGAs, so a performance comparison is not possible.

Streams-C [9] and ImpulseC [19] adopt imperative ANSI C syntax and an execution model based on Communicating Sequential Processes and introduce non-standard syntax and constructs for specifying designs such as special comment blocks which are used to annotate the C application code. The specialised syntax makes the languages harder to integrate with existing source-to-source translation or aspect weaving frameworks.

Hybrid approaches such as MaxCompiler [5] separate the CPU run-time component from the accelerated one, providing a C run-time environment and a Java API for building dataflow designs via meta-programming. The separation complicates the development process, hindering sharing of design parameters and, consequently, the design space exploration process. The use of meta-programming simplifies design parametrisation, but can make resulting programs harder to understand. In contrast, the proposed approach allows the computation description, which includes CPU and dataflow components, to be specified using a single language and to be decoupled from design parametrisation and other optimisation strategies which are captured as LARA aspects. This separation of concerns results in more intuitive and maintainable descriptions.

The use of LARA aspects in guiding the compilation process of C applications is described in [7] and [8] but the backend compilation targets a von Neumann architecture (with a GPP and custom accelerator) unlike the dataflow architecture proposed in this paper. The approach described in [7] and [8] relies more on high-level source transformation

TABLE IV
FEATURE COMPARISON OF THE FAST/LARA APPROACH AND EXISTING DATAFLOW IMPLEMENTATIONS.

Language	Syntax	Paradigm	Support	Implementation	Design Parametrisation	Optimisation Strategies
Lucid	Lucid	Functional	Software	Multiprocessor	Manual Source Transformation	Manual Code Revision
SISAL	SISAL	Functional				
Lustre	Lustre	Synchronous				
MaxCompiler	C99(SW) Java(HW)	Imperative(SW) Dataflow(HW)	Combined	CPU + FPGA	Meta-programming	
Streams-C ImpulseC	C99	Imperative(SW) CSP(HW)			Compiler Directives	
FAST/LARA	C99(SW/HW) LARA(Aspects)	Imperative(SW) Dataflow(HW) AOP(Aspects)			Compiler Directives + Automated Aspect-Directed Source Transformation	

whereas our approach is based on a systematic design space exploration process, which enables the analysis of more low-level optimisations. Finally, [7] and [8] do not consider development aspects which can be used to improve developer productivity.

VII. CONCLUSION

This paper presents a novel development approach for dataflow designs based on the FAST language, which supports aspect-driven compilation. We show that this approach meets the requirements in performance, portability, integration and productivity. The approach has been used in producing many optimised designs, including run-time reconfigurable designs for seismic imaging based on the RTM algorithm which are over 100 times faster than the corresponding software versions.

Current and future work includes aspect descriptions and the related facilities that cover systems with different types of accelerators, and their optimisation. Our approach can also be extended to cover systematic development of aspect descriptions for optimising a variety of application domains, from Monte-Carlo simulations in finance [20] to genetic sequence matching [21].

ACKNOWLEDGMENTS

This work is supported in part by the China Scholarship Council, by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by UK EPSRC, by Maxeler University Programme, and by Xilinx.

REFERENCES

[1] M. Flynn, O. Pell, and O. Mencer, "Dataflow Supercomputing," in *Proc. FPL*, 2012.

[2] O. Mencer, "Maximum Performance Computing for Exascale Applications," in *SAMOS*, 2012.

[3] J. von Neumann, "First draft of a report on the EDVAC," *Annals of the History of Computing*, pp. 27 –75, 1993.

[4] Tiobe Software, "Tiobe Programming Index," http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html, 2012.

[5] O. Lindtjorn, R. G. Clapp, O. Pell, O. Mencer, M. J. Flynn, and H. Fu, "Beyond Traditional Microprocessors for Geoscience High-Performance Computing Applications," *Micro, IEEE*, vol. 31, no. 2, pp. 41–49, 2011.

[6] J. M. Cardoso, T. Carvalho, J. G. Coutinho, W. Luk, R. Nobre, P. Diniz, and Z. Petrov, "LARA: an Aspect-Oriented Programming Language for Embedded Systems," in *Proc. AOSD*, 2012.

[7] J. Cardoso, J. Teixeira, J. Alves, R. Nobre, P. Diniz, J. Coutinho, and W. Luk, "Specifying Compiler Strategies for FPGA-based Systems," in *Proc. FCCM*, 2012.

[8] J. M. Cardoso, R. Nane, P. C. Diniz, Z. Petrov, K. Krátkỳ, K. Bertels, M. Hübner, F. Gonçalves, J. G. d. F. Coutinho, G. Constantinides *et al.*, "A New Approach to Control and Guide the Mapping of Computations to FPGAs," in *ERSA*, 2011.

[9] M. Gokhale, J. Stone, J. Arnold, and M. Kalinowski, "Stream-oriented FPGA Computing in the Streams-C High Level Language," in *Proc. FCCM*, 2000.

[10] D. Quinlan, "ROSE: Compiler Support For Object-Oriented Frameworks," *Parallel Processing Letters*, vol. 10, pp. 215–226, 2000.

[11] G. Kiczales, "Aspect-oriented Programming," in *Proc. ICSE*, 2005.

[12] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell, "Exploiting Run-Time Reconfiguration in Stencil Computation," in *Proc. FPL*, 2012, pp. 173–180.

[13] M. Araya-Polo *et al*, "Assessing Accelerator-based HPC Reverse Time Migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 1, pp. 147–162, 2011.

[14] E. A. Ashcroft and W. W. Wadge, "Lucid, a Nonprocedural Language with Iteration," *Communications of the ACM*, vol. 20, no. 7, pp. 519–526, 1977.

[15] J. Gurd and W. Bohm, "Implicit Parallel Processing: SISAL on the Manchester Dataflow Computer," *Proceedings of the IBM-Europe Institute on Parallel Professing*, 1987.

[16] J. McGraw *et al.*, "SISAL: Streams and Iteration in a Single-assignment Language," Lawrence Livermore National Lab, Tech. Rep., 1983.

[17] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The Synchronous Data Flow Programming Language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991.

[18] N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and Verifying Real-time Systems by Means of the Synchronous Data-flow Language LUSTRE," *IEEE Transactions on Software Engineering*, vol. 18, no. 9, pp. 785–793, 1992.

[19] "Impulse C." [Online]. Available: http://www.impulseaccelerated.com/products_universal.htm

[20] Q. Jin, D. Dong, A. Tse, G. Chow, D. Thomas, W. Luk, and S. Weston, "Multi-level Customisation Framework for Curve Based Monte Carlo Financial Simulations," in *ARC*, 2012.

[21] J. Arram, K. Tsoi, W. Luk, and P. Jiang, "Hardware Acceleration of Genetic Sequence Alignment," in *Proc. ARC*, 2013.