A Simple Programming Model for New-Generation Hardware

Brad Alexander Andrew Wendelborn *

January 3, 2006

Abstract

A large abstraction gap is emerging between new highly parallel hardware architectures and the von-Neumann model that underpins most software. Better compiler technologies or new programming models are needed to bridge this gap. In this paper, we propose a software architecture for an implementation of a compiler mapping a simple functional programming language to highly parallel FPGA hardware.

1 Introduction

At their core, most programming languages present programmers with the abstract model of a von-Neumann machine. This model is simple for the programmer and, historically at least, has mapped well to commodity hardware. However, in recent times, in order to achieve rapid growth in processing speed, hardware has evolved away from the von-Neumann model. Today, the simple model that programmers see is a fiction maintained by a combination of sophisticated compilers and extraordinarily complex and highly-tuned hardware. Unfortunately, this fiction is becoming increasingly difficult to maintain on the hardware side. It is now clear that current techniques, such as deep pipelining, dynamic out-of-order issue, and aggressive caching, will not be enough to sustain rapid growth in processor speed in years to come[1, 11].

In future, rapid increases in processing speed will have to come from different approaches to hardware design. A number of new, highly parallel, hardware designs have been posited[4, 13, 6]. These general-purpose designs can deliver substantially better performance than conventional architectures.

Another interesting approach to increasing processing speed is to specialise the hardware architecture to the application at hand. Inexpensive Field-Programmable-Gate-Array's (FPGA's) combined with sophisticated tools and libraries to abstract from details of specifying, and optimising, low-level functionality and layout[5, 8, 12, 3] make such specialisation increasingly accessible. It is this approach, applied to the targetting of high-performance applications to hardware, that we canvass in this article.

*Department of Computer Science, University of Adelaide, Adelaide 5005, Australia. Phone: +61 8 303-5681, E-mail: brad@cs.adelaide.edu.au, andrew@cs.adelaide.edu.au

1.1 Our approach

This article presents a software architecture for automatically translating programs, written in a simple functional programming language, to FPGA hardware. The source language, Adl, abstracts over almost all aspects of parallelism and communications while still being general enough to easily express a broad range of applications. The novelty of our approach lies in our use of point-free code, a highly transformable notation with explicit communication, to bridge the wide gap between the source language and the concrete notation used to target the FPGA.

It should be noted that the core, front-end, components of this architecture are taken from an existing implementation targetting distributed parallel machines described in[2]. The architecture proposed in this article refines the optimisation stage of this implementation and outlines the structure of a new back-end.

We start by giving a short overview of our architecture and follow with a description of each of its parts.

2 Architectural Overview

Figure 1 illustrates the software architecture for the compiler described in this work. The boxes represent stages in the compilation process. The bold boxes denote stages that have an existing implementation[1]. The dashed lines in the diagram represent programming interfaces. The heavy dashed line represents the, already implemented, application programming interface. The labels on each arc in the diagram represent programming notations. The internal interfaces of the system are dominated by point-free code. The choice of point-free form is, in part, due to its innate transformability and, in part, due to its support for parallelism and explicit communication.

In the following, we describe each stage of the architecture in turn. Starting with the source language, Adl.

3 Adl

Adl is a small strict functional language designed, primarily, for writing parallel numerical applications. Parallelism in Adl is expressed through operations on vectors including, `map`, `reduce` and `scan`. The functional model was chosen to avoid artificial temporal dependencies between operations, which can obscure the meaning of the program and limit scope for transformation.

To illustrate what Adl code looks like, figure 2 shows a short program that adds two to every element of a nested input vector by the use of nested `map` functions. The `map` operation applies the function in its first argument, in parallel, to every element in the vector in its second argument. As can be seen from the example, vectors can be nested. Adl permits arbitrary nesting of vectors, and tuples, its other aggregate structure.

As well as the features above, Adl also supports vector indexing, which supports arbitrary communication, and dynamically bounded iteration. The current implementation of Adl does not

[1]The top two bold boxes have been implemented by us and the bottom box, hardware mapping, is vendor software from Xilinx corp. `http://www.xilinx.com/`

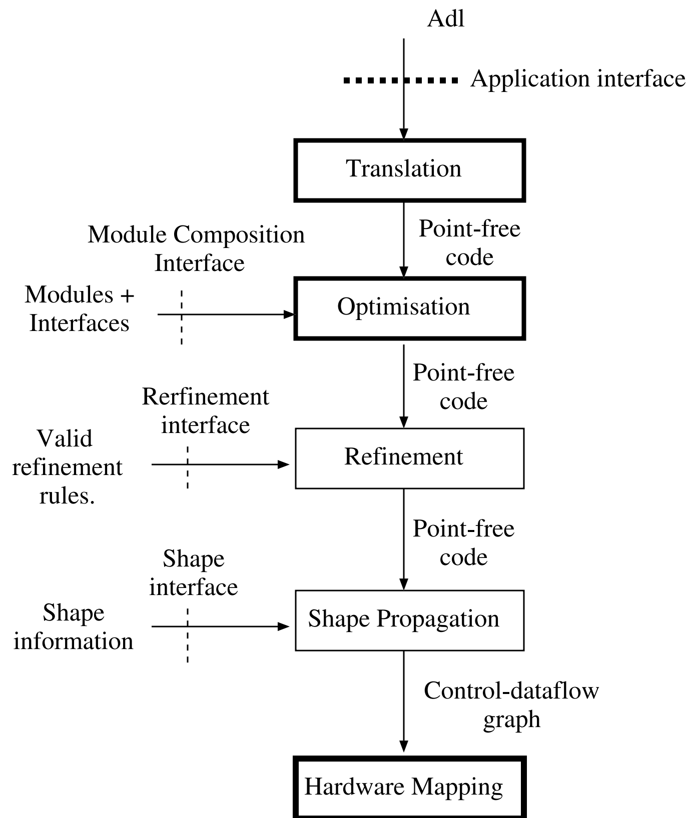

Figure 1: Overview of a software architecture for mapping Adl into hardware. Programming interfaces are denoted with dashed lines. The heavy dashed line denotes the existing applications programming interface.

```
main a:vof vof int :=
    let
       f x :=
          let
             g y := y + 2
          in
             map (g,x)
          endlet
    in
       map (f,a)
    endlet
```

Figure 2: An Adl program that adds two to every element of a nested input vector.

support recursion[2].

Adl's functional model and the nature of its parallel operations combine to avoid issues of interference, deadlock, and starvation that face programmer in settings where the expression of parallelism is more explicit. The reader is referred to [2] for a further introduction to Adl language. Next, we describe the components of our software architecture, starting with the translator.

4 The Translator

The translator converts Adl code to point-free code. Point-free code cannot contain variables as a means to store temporary values. Without the support of an implicit store, data must be explicitly routed between operations. A primary task of the translator is to produce code that ensures that data is routed to where it needs to be. In performing this task, the translator is very conservative. It routes all data that was originally in the scope of each operation in the Adl source to the doorstep of the corresponding operation in the point-free code. In most cases, not all of this data is required. It is the task of the next stage, the optimiser, to reduce this flow of redundant data.

5 The Optimiser

The optimiser systematically applies rewrite rules to the point-free code produced by the translator to produce code which has been shown, in our previous experiments, to be of comparable efficiency to hand-written point-free code. The rewrite rules are semantics-preserving which means that at every stage of the optimisation process we have semantically valid program.

Point-free code does not lend itself to the complex global analyses often found in traditional approaches to program optimisation. Instead, rewrite rules typically have only a very localised impact. Global changes are wrought by applying rules in a manner that propagates changes, in waves, through the whole program. In our optimiser, code which expresses the precise data needs of the latter part of the program is propagated back toward the start of the program, leaving more efficient code in its wake.

The biggest problem to overcome in defining the optimisation process is the great diversity of code to be handled. It is impossible to collate enough rewrite rules to apply, directly, to the variations of code that occur. The approach we use is to have a very small number of high-impact rules which can significantly increase the efficiency of the program where they are applied. These high-impact rules are only rarely applicable to unprocessed code so we apply sets of simple normalisation rules to shape the code to normal forms where high-impact rules can be applied. In other words, we change the code to fit our rules.

The benefit of our approach is that we achieve effective optimisation without the use complex rules or difficult analyses. The trade-off is the extra compute-time required to perform normalisation. We are of the opinion that, for most numerically intensive applications, this trade-off is well worthwhile.

[2]Though we leave the way open for the introduction of recursion in a future version of the language we have not found this restriction an impediment in the range of applications we have written so far.

5.1 Module composition

The optimiser is made up of modules in the form of sets of rules. Most modules are dedicated to some form of normalisation[3]. These modules are reused quite heavily in different parts of the optimiser.

Though this heavy reuse is a very positive outcome, it is tempered by the occasional error arising from incompatibilities in the grammars produced by the normalisation modules and the grammars expected by subsequently applied transformation modules. Without automated checking for compatibility between modules, such errors are difficult to track down and remedy. One part of our proposed architecture is the formalisation of interfaces to our modules and the introduction of static checking for compatibility[4] of grammars. We envisage that the development of interfaces will lead to a system that is much easier, and safer, to reconfigure by changing the way modules are composed[5]. An ability to reconfigure the system is valuable because, in our experience, optimisation is an open-ended process, where new applications can expose new opportunities for improvement. Ultimately, we plan to develop a formal interface for such reconfiguration such as that shown on the top-left of figure 1.

Our optimiser produces efficient point-free code with explicit communication between all operations. Some of this communication is expressed using general-purpose operations which are quite demanding of hardware. A stage of refinement to specialise these operations to a more efficient form is outlined next.

6 Refinement

The optimisation process produces efficient point-free code given the information that is available in the source program. One thing that the optimiser does to achieve this outcome is collapse multiple invocations of vector indexing functions into a single bulk-communications function called select.

Parallel versions of general communications operations, analogous to select, have worked efficiently on large-scale distributed architectures[9]. Unfortunately, the scalable, dynamic parallel access required by select is more difficult to implement within the constrained environment of an FPGA. The proposed refinement stage of the compiler provides a partial remedy to this problem by substituting more specialised communications operations for select. This substitution process is driven by refinement rules indicating where such substitution is allowable, given the user user-supplied assumptions about input data. The refinement stage of the compiler will produce point free code with specialised communication constructs inserted where possible. This code contains no information about the size of any vectors that it operates on. Such information needs to be embedded in the code before it can be translated to hardware. We describe this stage of compilation, called Shape Propagation, next.

---

[3]Examples of this include associating all function compositions to the left or right, minimising the length of composition sequences and removing redundant identity functions.

[4]Static checking of grammar interfaces in compilation systems has been used before, in a different setting and on a smaller scale in [7].

[5]Also note, that because each module consists of semantics-preserving rewrite rules, such reconfiguration cannot cause the optimiser to start producing incorrect programs.

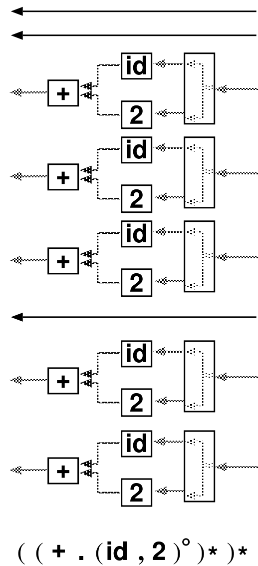

Figure 3: The code from figure 2 specialised with shape information of $(2, [3, 2])$ for the input vector.

7 Shape Propagation

Shape propagation takes information about the scale of vector data entering the point-free program and transmits this information through the point-free program. The shape information consists of the length of the input vector, or, in the case of nested vectors, a pair with the first element being the number of sub-vectors and the second element being a vector of shapes of sub-vectors. The result of shape propagation is a graph of individual operations and the communications links between them.

Figure 3 shows such a graph for the Adl program from figure 2, after propagation of the shape of a very small nested input vector. It is worth noting that Shape propagation a specialisation of the more general concept of Shape inference[10].

The graph produced by shape propagation still needs substantial translation to be mapped into hardware. We describe this stage next.

8 Mapping to Parallel Hardware

We have chosen to target our language to the moderately high-level notation of Control/Data Flow Graphs (CDFG's)[6]. The CDFG's we generate will be fed into tools, developed by Xilinx, for mapping to the FPGA. These tools will deal with most low-level details of hardware including,

[6]Another viable option is the Caltrop Actor Language (CAL)[8]. CAL allows operations to be expressed as actors. Actors can encapsulate internal state and can communicate with each other via explicit input and output ports. This framework is sufficient to capture the operations expressed in our point free code but, because our operations are purely functional, we do not have any a-priori need for internal state. If the maintenance of internal state exacts a penalty in terms of runtime performance then it is desirable to opt for a less powerful notation such as CDFG's.

the types of components to use and layout, leaving us free to concentrate on encoding operations and communications.

The target hardware platform is the Xilinx XUP Virtex II Pro Development System[7]. We expect to implement the constructs in our point-free code incrementally, starting with simple elements, of the type shown in figure 3, and working our way toward more complex constructs. We expect to find the implementation of constructs supporting random access to vector elements especially interesting.

9 Summary

A large gap is emerging between new highly parallel hardware architectures and the von Neumann model currently used to program them. Better compiler technologies or new programming models are needed to bridge this gap. We described a software architecture for an implementation of a programming model which provides generality and ease of use to the programmer whilst still allowing for efficient and effective targetting to an FPGA platform. The key to this process is the transformability of the point-free intermediate notation used in this implementation and the freedom it provides in the design of architecture components.

Acknowledgments We would like to thank Rob Esser and Xilinx for their invaluable guidance on the proposed research.

References

[1] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *ISCA*, pages 248–259, 2000.

[2] B. Alexander and A. Wendelborn. Automated transformation of BMF programs. In *The First International Workshop on Object Systems and Software Architectures.*, pages 133–141, 2004.

[3] W. Böhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar. Mapping a single assignment programming language to reconfigurable systems. *J. Supercomputing.*, 21(2):117–130, 2002.

[4] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[5] Katherine Compton and Scott Hauck. Reconfigurable computing: a survey of systems and software. *ACM Comput. Surv.*, 34(2):171–210, 2002.

[7] See: `http://www.xilinx.com/univ/xupv2p.html`.

[6] W. Dally and P. Hanrahan. Merrimac: Supercomputing with streams. In *Supercomputing '03*, 2003.

[7] Merijn de Jonge and Joost Visser. Grammars as contracts. *Lecture Notes in Computer Science*, 2177, 2001.

[8] Johan Eker and Jrn W. Janneck. An introduction to the caltrop actor language (whitepaper). URL: http://embedded.eecs.berkeley.edu/caltrop/docs/CaltropWhitePaper.pdf.

[9] Jonathan M.D. Hill and David B. Skillicorn. Lessons learned from implementing BSP. *Future Gener. Comput. Syst.*, 13(4-5):327–335, 1998.

[10] C. Barry Jay. *Research Directions in Parallel Functional Programming, K. Hammond and G.J. Michaelson (eds)*, chapter 9: Shaping Distributions, pages 219–232. Springer-Verlag, 1999.

[11] Kunle Olukotun and Lance Hammond. The future of microprocessors. *Queue*, 3(7):26–29, 2005.

[12] Roger M.A. Peel and Wong Han Feng. Using csp to verify aspects of an occam-to-fpga compiler. In Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green, editors, *Communicating Process Architectures 2004*. IOS Press, 2004.

[13] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Dataflow: The road less complex. In *In Workshop on Complexity-effective Design (WCED) in conjunction with the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.