Indian Institute of Technology, Bombay
Department of Computer Science and Engineering

SYNOPSIS
of the Ph. D. thesis entitled

A competitive pathway from high-level programs to hardware specifications

Proposed to be submitted in partial fulfilment of the degree of
Doctor Of Philosophy
of the
Indian Institute of Technology, Bombay
by
Sameer D. Sahasrabuddhe

Advisors
Prof. Kavi Arya and Prof. Madhav P. Desai

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

Introduction

The design of large hardware systems is an expensive venture due to two reasons: the need for highly trained manpower, and verification of the system at every step of the process. These costs can be reduced through high-level synthesis, the process of generating hardware from high-level programs.

High-level synthesis starts with an executable specification, i.e., a representation of the desired functionality in a machine-readable and simulatable form[2]. Such an executable specification can also be a program written in a high-level programming language. The executable specification is translated into a circuit implementation by a compiler flow that lowers the level of abstraction in stages.

The use of high-level programs as the starting point makes hardware design accessible to a very large set of users. If the compiler is guaranteed to produce a correct implementation of the input specification, the resulting hardware does not have to be verified. Only the behaviour of the input specification has to be verified, which can be done using existing software verification practices.

High-level synthesis is an active research area due to these potential savings in hardware design. But most efforts have at least one of two important shortcomings: restrictions are imposed on the input language, or the abstraction is not sufficient enough to hide hardware details. As a result, programmers are unable to use standard practices in writing programs and the compiler is restricted in the scope of optimisations used on the generated hardware.

The main aim of our work is to explore the design of a high-level synthesis process that preserves common practices in software programming while providing a verifiable and optimisable path to a hardware implementation. This requires a compiler flow that can generate efficient circuits from complex high-level programs. Such a compiler flow must have the following features:

1. The flow should be independent of the programming language used.
2. The flow should guarantee a correct implementation of the specified behaviour.
3. The flow should support optimisations that can scale to very large systems.

We present a compiler flow that achieves this goal by introducing an an intermediate step between software and hardware compilation in the form of an intermediate representation called AHIR[1]. This decouples the high-level issues encountered when writing programs, from low-level issues in the hardware implementation. We describe a translation method that that always produces a correct AHIR specification from a high-level program, thus eliminating the need for verification.

An AHIR specification is factorised into three components: control-flow, data-flow and storage. The three components can be analysed and transformed separately without affecting

each other, enabling optimisations that can scale with the size of the circuit. We demonstrate this with an optimisation that improves resource utilisation in the data-path using a static analysis of the control-path. The specification is independent of implementation delays and can be routinely mapped to hardware. Timing correctness can be guaranteed by satisfying a set of single-sided delay constraints.

Related work

A number of attempts have been made to create a path from high-level programming languages to hardware descriptions. These can be loosely categorised as follows:

Improvements over RTL

Some efforts attempt to raise the abstraction in an RTL description in order to support higher-level constructs. For example, Bluespec[3] describes the behaviour of a system in terms of explicit state entities and *guarded atomic actions*. The control logic for the interaction of these actions is synthesised automatically. The resulting hardware has been shown to be competitive with hand-coded RTL.

But this is a tool meant for a hardware designer, not a programmer. An effort that proposes a new language for hardware design introduces new requirements in the skills needed to use that language. The user has to be aware of the components involved, and the low-level state that is being manipulated by operations in the system. The language can be quite powerful in expressing the architecture of the hardware, but the target user is a hardware designer who can effectively utilise this expressive power.

Modified high-level languages

Some efforts reinterpret existing programming languages as hardware descriptions, and also extend them with special features. These systems essentially propose new programming languages that may be superficially similar to the parent languages. But it becomes difficult to retain standard programming practices in this setup — some of the common knowledge gained by programmers has to be revaluated in this new use of the language.

The language SA-C[4] for example is a purely functional subset of the C programming language. SA-C declares that variables represent values and not storage. Thus, pointers are not allowed in SA-C, and information is passed by value. The language introduces new syntax to work with arrays which is especially powerful in expressing DSP algorithms.

But SA-C makes an *a priori* choice with respect to the programming paradigm used. Although the syntax superficially represents an imperative language, the program must be designed in a functional style. SA-C effectively creates a new language, and the programmer is forced to revaluate existing programming practices within the scope of this modified language.

On the other hand, Handel-C[5] is a language that guarantees complete ISO-C compatibility and further extends it to include features such as arbitrary-width data types, primitives for parallel or sequential execution, thread synchronisation mechanisms, etc. These features can prove very effective in describing the intended hardware.

An important property of Handel-C is that the compiler guarantees a cycle-accurate implementation of every program, which facilitates verification. But this cycle-accurate nature

restricts the scope of optimisation since the compiler cannot reorder instructions. As a result, the programmer must manually optimise sequences of instructions. This can be an error-prone process and opportunities could be easily missed simply because they were not obvious to the programmer.

In both these examples, the focus is on the language used by the designer. The designer is encouraged to use specific language features to describe the intended hardware, instead of the compiler inferring hardware from the behaviour of the program. Thus, these languages represent completely different design styles hidden behind a syntax that resembles C.

High-level programs as hardware specifications

Some efforts simply use standard programming languages as a starting point that provides a hardware specification rather than a description. This approach makes it easy for programmers to write a hardware specification like any other program, while the compiler is free to automatically infer a suitable implementation. Such an approach usually employs an intermediate representation as a transition step from software to hardware.

For example, the Phoenix project uses an intermediate representation called Pegasus[6] for a compiler flow from C to hardware[7]. Pegasus has a data-flow architecture where operations use asynchronous handshakes to exchange data along data-flow edges. Control-flow is represented as predicates associated with the operations. Special token flow edges are used to represent dependences between memory operations. A description in Pegasus can be implemented directly by translating each operation to a micropipeline stage. The compact nature of Pegasus allows the compiler to implement a number of high-level transformations in a native manner.

The SPARK[8] project uses an internal representation based on hierarchical task graphs (HTG). Statements that have no control-flow between them are aggregated together into basic blocks. These basic-blocks are used as HTG nodes to form hierarchical structures such as branching and loops. This hierarchy captures all the information that is useful for high-level transformations such as code motion and speculation[9]. The compiler uses a heuristic that combines these transformations with scheduling and resource binding in order to improve the schedule length as well as resource utilisation in a single implementation phase.

Our work involving AHIR is similar to these efforts since the goal is to transparently compile software programs into hardware. AHIR differs from both Pegasus and SPARK in the fact that an AHIR specification is factorised into three separate components: control flow, data flow and memory. This factorisation is the key to a compiler flow that can scale to very large systems. The components can be optimised and implemented separately as long as specified constraints are satisfied.

An AHIR specification always represents a circuit that can be directly translated to a hardware implementation. The decoupled control and data paths respectively represent op-

eration sequences and operators bound to them. The compiler can use this information to perform delay-insensitive low-level optimisations natively in AHIR itself.

AHIR: A Hardware Intermediate Representation

AHIR is a compact graph-based hardware representation, that provides a delay-independent specification of the circuit. A specification in AHIR can be directly implemented by a straightforward translation to corresponding hardware components. But the implementor is free to produce other implementations as well, based on additional information available about the target platform. Features such as parallelism and resource sharing may be available in the input language, or introduced by the compiler. In either case, the intermediate representation is rich enough to describe them.

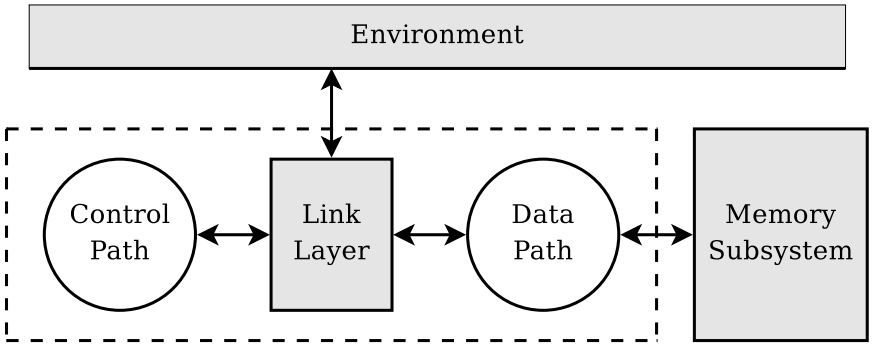

Figure 1: An AHIR module.

A module in AHIR consists of two closely interacting components: a control-path and a data-path. The data-path in a module is a pool of operators connected by wires. The control-path is a petri-net that specifies the ordering of events in the module.

The control-path is required to be in a special class of petri-nets that we call "Type-2". This is defined in terms of constraints specified on the structure of the petri-net. The constraints make it easy to design analyses that are scalable with the size of the petri-net. At the same time, the class is powerful enough to express useful sequencing concepts such as parallelism, pipelining, etc. Every member of this class is live and safe. We provide an algorithm that determines membership of this class in linear time.

The control-path and data-path interact through a link layer by exchanging request-acknowledge handshakes that encapsulate delays occurring in the implementation. AHIR specifies a set of constraints on the delays involved — an implementation that satisfies these constraints is guaranteed to be correct.

An AHIR specification does not include any details about the memory architecture; only a simple linear address space is assumed. Each data-path can specify an arbitrary number of memory access ports. The only requirement from the memory subsystem is that a request

made on an access port must be served eventually.

High-level Synthesis using AHIR

AHIR provides an effective path from complex high-level programs to hardware implementations. We have built a compiler flow using AHIR, that consists of two phases: a software phase that optimises and translates a high-level program to AHIR, and a hardware phase that optimises and translates an AHIR specification to a hardware implementation. The result is an end-to-end compiler flow that translates C programs to synthesisable VHDL descriptions.

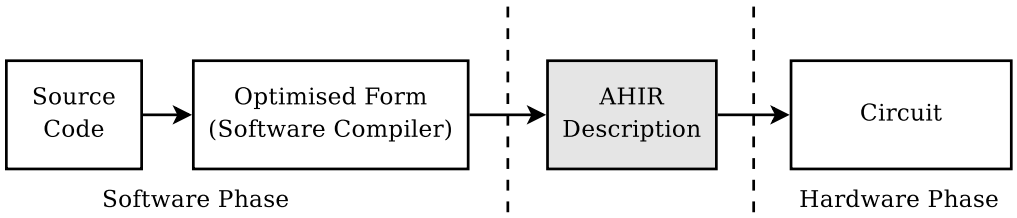

Figure 2: High-level Synthesis using AHIR.

The software phase consists of existing software compilation techniques along with a back-end that generates AHIR specifications. The translation to AHIR uses a well-known intermediate form called a CDFG (Control Data Flow Graph). We derive a CDFG from the optimised program and then produce an AHIR specification using a simple piece-wise construction method. We prove that the method used guarantees correctness *by construction* — the AHIR specification correctly implements the behaviour specified by the CDFG.

The control, data and storage components of the AHIR specification can now be implemented independently. The factorisation allows us to efficiently analyse and transform these components in order to optimise the circuit. We demonstrate this with an optimisation that improves resource utilisation by sharing resources in the data-path.

The resource sharing is based on a static analysis of the control-path that identifies pairs of operations that cannot be active at the same time. The algorithms and supplementary data structures used by the analysis are very simple and close to linear in complexity. Experimental results show that even this simple approach is quite effective in improving the resource utilisation of the circuit.

The optimised AHIR specification is translated to a circuit implementation by the hardware phase. The compiler is free to explore various architectures in this phase, as long as the timing constraints in AHIR are satisfied. The constraints are easy to satisfy in practice by sufficiently padding the relevant circuit delays. We translate the AHIR specification to a synchronous circuit implementation by directly replacing each component with an equivalent VHDL implementation.

Experiments

The current implementation has been tested on a number of examples chosen from diverse application domains: LINPACK, Red-Black Trees, FFT, AES block cipher and A5/1 stream cipher. We compare the performance of the generated circuits with two extremes: equivalent programs running on a microprocessor (Intel Pentium IV), and hand-crafted circuits that implement the same behaviour (results obtained from third-party literature). The metric used for comparing performance is the ratio of the throughput delivered by the circuit to the area occupied by it.

The synthesis results show that it is possible to translate complex programs to hardware circuits using AHIR. The generated circuits are competitive with equivalent programs running on a general-purpose microprocessor. But the performance is less than that of hand-crafted circuits by two orders of magnitude. This gap can be bridged by further work in three key areas: expressing parallelism in the high-level specification, introducing optimisations on the AHIR specification, and using customised memory subsystems that support parallel accesses.

Summary

We have established a pathway from high-level programs to hardware, with the following features:

1. An intermediate representation that is easy to analyse and transform using methods that can scale with the size of the circuit. The scalability is made possible by an explicit factorisation of the intermediate representation into three components: control, data and storage.

2. A verifiable path from high-level programs to the abstract representation that is built on existing software compilation techniques.

3. A set of constraints to ensure correctness when translating the intermediate representation to a hardware implementation. These constraints are easy to satisfy in practice and allow considerable freedom to the hardware implementor.

These features ensure that the entire path is verifiable and scalable, thus providing a competitive option for high-level synthesis.

Future work

Future research work in needs to target two goals in order to provide a viable alternative for hardware design. One is to extend the reach of the compiler in terms of programming languages and design paradigms, and the other is to improve the quality of hardware generated.

A universal design platform

AHIR can potentially be used as a universal design platform that unifies *software compilation* (translating programs to executables) with *hardware compilation* (synthesising circuits from programs). The expressive power of AHIR is evident at two levels: as a framework for exploring hardware architectures, and as a target for the design and implementation of high-level languages. A combination of these two aspects will result in a compiler flow that can map various classes of high-level languages to different low-level architectures.

Our experiments in translating C programs to hardware demonstrate that AHIR can support imperative languages, with suitable implementations for specific features. Functional programming languages can also be implemented with AHIR using techniques similar to those used in microprocessors. The Type-2 petri-net in AHIR is also sufficient for supporting synchronous languages like Esterel. Further research should be aimed at exploring the practical aspects of supporting these languages and at providing the theoretical background for verifying their implementation.

Hardware optimisations

The synthesis of large systems starting from complex high-level programs requires further work towards improving performance at the system level. The compiler must be able to generate implementations that deliver good throughput while occupying acceptable amounts of resources. These two parameters are usually traded-off with each other since faster computations come at the cost of more hardware, and *vice versa*. AHIR provides many opportunities for creating transformations that improve the final circuit, such as pipelining for improved throughput and sharing resources across modules for reduction in area.

Memory subsystem

In addition to optimised hardware, the performance delivered by the memory subsystem is critical to the overall performance of the system. High-level synthesis has to be coupled with an automated memory design flow to generate application-specific memory subsystems that can keep up with the hardware system.

Some work has been done by related projects towards developing simple performance models[10] for memory subsystems. Such models will support an integrated memory subsystem design procedure that can explore large areas of the design space in a feasible manner. For a given application, the design process may be parameterised by the properties of the expected memory access trace[11].

References

[1] S. D. Sahasrabuddhe, H. Raja, K. Arya, and M. P. Desai, "AHIR: A Hardware Intermediate Representation for Hardware Generation from High-level Programs," in *20th International Conference on VLSI Design*, pp. 245–250, January 2007.

[2] D. D. Gajski, F. Vahid, and S. Narayan, "A System-Design Methodology: Executable-Specification Refinement," in *European Design and Test Conference (ED&TC) 94*, pp. 458–463, February 1994.

[3] Arvind, R. Nikhil, D. Rosenband, and N. Dave, "High-level synthesis: An Essential Ingredient for Designing Complex ASICs," in *International Conference on Computer Aided Design (ICCAD 2004)*, November 2004.

[4] "Cameron Project and Single Assignment C (SA-C)." http://www.cs.colostate.edu/cameron/.

[5] "Celoxica: Software-Compiled Systems Design." http://www.celoxica.com/.

[6] M. Budiu and S. C. Goldstein, "Pegasus: An Efficient Intermediate Representation," tech. rep., School of Computer Science, Carnegie Mellon University, April 2002.

[7] G. Venkataramani, M. Budiu, T. Chelcea, and S. Goldstein, "C to Asynchronous Dataflow Circuits: An End-to-End Toolflow," in *International Workshop on Logic & Synthesis*, (Temecula, CA), pp. 501–508, June 2004.

[8] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "SPARK : A High-Level Synthesis Framework For Applying Parallelizing Compiler Transformations," in *International Conference on VLSI Design*, January 2003.

[9] S. Gupta, N. Savoiu, N. Dutt, R. Gupta, and A. Nicolau, "Using Global Code Motions to Improve the Quality of Results for High-Level Synthesis," in *IEEE Transactions On Computer-Aided Design Of Integrated Circuits And Systems*, vol. 23, February 2004.

[10] G. Hazari, *Bottleneck Analysis and Performance Modeling of VLSI Memory Sub-systems*. PhD thesis, Department of Electrical Engineering, IIT Bombay, 2009. Proposed for submission in January 2009.

[11] A. Singla, "Memory Access Pattern Analysis," Master's thesis, Department of Electrical Engineering, IIT Bombay, June 2008.