AHIR: A competitive pathway from high-level programs to hardware

Sameer D. Sahasrabuddhe, Kunal P. Ghosh, Sreenivas Subramanian, Kavi Arya and Madhav P. Desai
Indian Institute of Technology – Bombay, Powai, Mumbai – 400076, INDIA
Email: {sameerds@cse, ssreenivas@ee, kunalghosh@ee, kavi@cse, madhav@ee}.iitb.ac.in

***Abstract*—With rapid increase in the size and performance of VLSI systems, it is now possible to map increasingly complex applications to hardware. However, the design effort involved in mapping an algorithm to hardware is substantial. Low-power embedded microprocessors have gained favour as reliable platforms for rapid development and deployment of algorithms in hardware. We present a complete flow from algorithm to hardware as an alternative to embedded microprocessors. The flow supports an essentially unlimited class of programming languages, and generates circuits that are correct *by construction*.**

The flow uses an intermediate representation called AHIR, which is an orthogonal factoring of the program behaviour into control, data and memory aspects. This allows optimisations that are scalable to very large systems. We demonstrate the flow using examples ranging from stream ciphers to database and linear algebra applications, and show that the resulting circuits are competitive with processor based implementations.

I. Introduction

The design of large hardware systems is expensive due to two reasons: the need for trained manpower, and verification at every step of the process. One way to overcome this difficulty is to deploy low-power embedded microprocessors so that training and verification is moved to the software domain. We present a high-level synthesis flow as an alternative that provides a verifiable and optimisable path from executable specifications[3] to efficient hardware. The synthesis flow is made competitive by the following features:

1) It is independent of the programming language used.
2) It guarantees correctness of the implementation.
3) The intermediate representation used supports optimisations that can scale with the size of the program.

Such a flow eliminates the need for design expertise and also the need to verify the resulting hardware. It is sufficient to verify the input specification (the program), which can be done using standard software techniques.

The synthesis flow introduces an intermediate step in the form of a representation called AHIR (A Hardware Intermediate Representation). The representation is independent of the programming language used, and can be routinely translated to a hardware implementation, while also supporting scalable optimisations[1][2].

II. Related work

Attempts at creating a path from high-level programs to hardware descriptions, can be loosely categorised as follows:

A. Improvements over RTL

Efforts like Bluespec[4] raise the abstraction in an RTL description in order to support higher-level constructs. Such a language can be very powerful in expressing the architecture of the hardware, but the target user is a hardware designer who can effectively utilise this expressive power.

B. Modified high-level languages

Some efforts reinterpret programming languages as hardware descriptions, and also extend them with special features. The language SA-C[5], for example, is a purely functional subset of C that disallows pointers. On the other hand, Handel-C[6] is a language that guarantees complete ISO-C compatibility and also provides additional primitives.

In both examples, the designer must use specific features to generate efficient hardware, instead of the compiler inferring a hardware implementation. This forces the programmer to revaluate standard programming practises.

C. High-level programs as hardware specifications

Some efforts simply interpret a program as a behavioural specification, which is mapped to a hardware implementation using an intermediate representation.

For example, the Phoenix project uses an intermediate representation called Pegasus[7] for a compiler flow from C to hardware[8]. A description in Pegasus can be implemented as a micropipeline. The representation allows the compiler to natively implement a number of high-level transformations.

The SPARK[9] project uses an internal representation based on hierarchical task graphs (HTG). The compiler uses a heuristic to combine high-level transformations on the HTG — such as code motion and speculation[10] — with scheduling and resource binding to produce efficient hardware.

Our work is similar, since the goal is to transparently compile programs into hardware. AHIR differs from both Pegasus and SPARK since it factorises the system into three separate components: control flow, data flow and memory. This 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.

III. AHIR

A system in AHIR is a collection of modules connected to a memory subsystem as shown in Fig. 1. Each module represents one function from the input program. Function

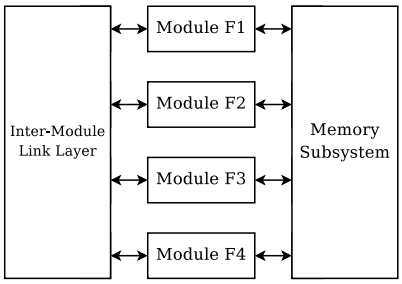


Fig. 1. An AHIR system.

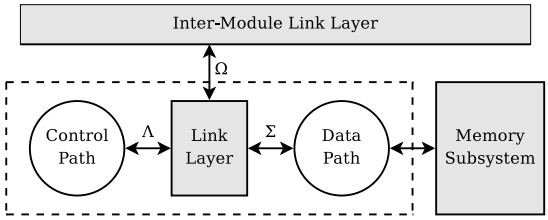


Fig. 2. An AHIR module.

calls are implemented through an *inter-module link layer*. The architecture of the memory subsystem is not defined in AHIR. It is only required to service every request *eventually*.

A. A module in AHIR

Two flows are described in an AHIR module — control-path and data-path — that communicate through an *intra-module link layer* as shown in Fig. 2. The control-path is a petri-net that specifies the ordering of events in the module. The data-path is a pool of hardware resources connected by wires.

Communication through the link layers is specified as an exchange of symbols. The set of symbols associated with a component is called its alphabet. The data-path uses alphabet Σ, while the control-path uses alphabet Λ. The interaction with the inter-module link layer is represented by the alphabet Ω.

B. Data-path

The data-path is a directed hyper-graph, where the edges represent values, and the nodes represent operations on these values. Each edge is a hyper-edge with a single tail and one or more heads. The tail drives a value on the edge, which reaches all the heads instantaneously.

A data-path node is described by a state machine with an idle state, and one or more busy states. When a request req_i arrives at an idle node, the node samples all its incoming data-edges and changes state to busy_i. The node then operates on the sampled values and updates the outgoing data-edges. On completion, the node emits an acknowledge symbol ack_j, and then returns to the idle state.

The data-path uses load and store operators to communicate with external memory. Additionally, there are input and output ports that are used to exchange arguments and return values with the inter-module link layer during a function call.

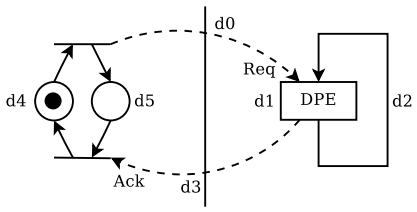


Fig. 3. Delays in an AHIR specification.

C. Control-path

The control-path is a petri-net that expresses the sequence in which events occur in a module. It is required to be in a class called "Type-2 Petri-nets", as defined in Section IV.

The control-path has a single marked place in the initial marking. This enables a single transition called init, which responds to a request symbol in alphabet Ω and begins execution of the module. The end of execution is represented by the fin transition that emits an acknowledgement symbol in Ω and marks the initially marked place.

D. The Intra-module Link Layer

The intra-module link layer translates symbols generated by the control-path (in Λ) to symbols for the data-path (in Σ) or the inter-module link layer (in Ω), and *vice versa*. It is defined as a set of functions that instantaneously consume symbols presented to them and generate new symbols.

E. Handshakes and delay constraints

Operations in AHIR are managed by symbolic handshakes. The control-path emits a $\mathrm{request}$ in Λ, which triggers an operator in the data-path. When done, the operator emits an $\mathrm{acknowledge}$ in Σ, which causes further events in the control-path. This *request-acknowledge handshake* encapsulates any delays in the implementation.

An implementation must satisfy a pair of one-sided delay constraints for the handshake in order to ensure correctness. Fig. 3 shows a hypothetical example with associated delays. When the control-path emits the request symbol Req, it must update its state before the arrival of the acknowledgement symbol Ack. Hence we have:

$$d_5 \leq d_0 + d_1 + d_3$$

Similarly, when the data-path emits an Ack, it eventually receives a Req. The data inputs must have stabilised before this request arrives. Hence we have:

$$d_2 \leq d_3 + d_4 + d_0$$

Note that the term $d_0 + d_3$ is common to both expressions. An implementation can always guarantee timing correctness by sufficiently padding these delays to satisfy the constraints.

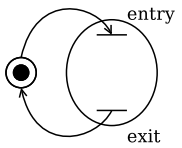


Fig. 4. A TPR and a Type-1 petri-net

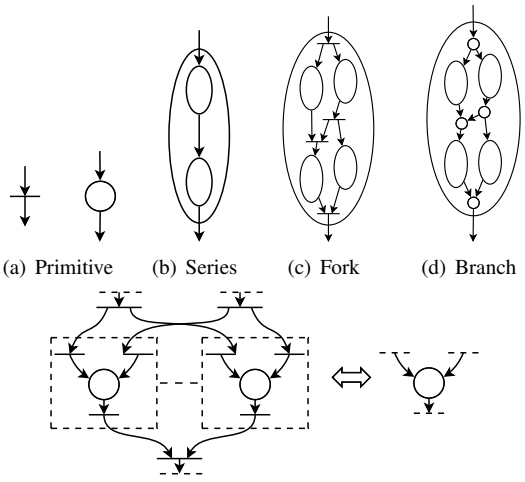


Fig. 5. Type-2 construction rules.

F. Execution model

AHIR uses a synchronous execution model. The control-path responds instantaneously to the arrival of symbols, while data-path elements take one or more cycles to execute. Values in the data-path are also propagated instantaneously. Clearly, this satisfies the delay constraints, since d_1 is finite, while other delays are zero.

G. The Inter-module Link Layer

The inter-module link layer is used to route function calls between modules. It has one arbiter for each module, that manages the flow of input arguments and return values between the calling module and the called module.

IV. Type-2 petri-nets

We propose a class of petri-nets called Type-2, based on a set of standard construction rules. The structure of a Type-2 petri-net is designed to enable analyses and transformations that are scalable with the size of the petri-net.

Definition 4.1: A **simple place (transition)** is a place (transition) with one incoming edge and one outgoing edge.

Definition 4.2: A **token-preserving region (TPR)** is a petri-net P that can be augmented with one simple place $\hat{p}$ and a sufficient number of simple transitions and edges, to produce a live and safe petri-net P' such that $\hat{p}$ is the only marked place in the initial marking (as shown in Fig. 4).

Definition 4.3: A **Type-1 Petri-net** is a live and safe petri-net that marks one simple place in the initial marking. Clearly, a Type-1 petri-net is constructed by augmenting a TPR.

```
d = m + n
b = m - n
if (b > 0):
  a = b + c
  d = e + a
x = d + 2
```

(a) Pseudo-code.

```
d1 = m + n
b = m - n
if (b > 0):
  a = b + c
  d2 = e + a
d3 = φ(d1,d2)
x = d3 + 2
```

(b) SSA form.

Fig. 6. A code fragment and its SSA form.

Definition 4.4: A **Standard TPR (STPR)** is a TPR constructed using the standard set of rules (defined below).

Definition 4.5: A **Type-2 petri-net** is a Type-1 petri-net created by augmenting a Standard TPR.

Type-2 construction rules:

1) A simple place or transition is a *primitive* STPR.
2) A *series region* is an STPR formed by joining two standard STPRs in series.
3) A connected *acyclic* subgraph made of STPRs, forks and joins is itself an STPR called a *fork region*.
4) A connected (possibly cyclic) subgraph made of STPRs, branches and merges is an STPR called a *branch region*.
5) Replacing a merge place in a branch region with parallel merges as shown in Fig. 5(e) also results in a standard TPR. The set of parallel merges introduced by this replacement is called a *parallel-merge region*.

The Type-2 construction rules are illustrated in Fig. 5. The branch region allows cycles in order to express arbitrary branch and loop structures. The fork region does not allow cycles since that introduces further conditions for liveness. But this does not affect the expressive power of Type-2 petri-nets. Parallel-merge regions implement the runtime selection of values at the exit of a branch, such as variable `d` in Fig. 6.

V. The synthesis flow

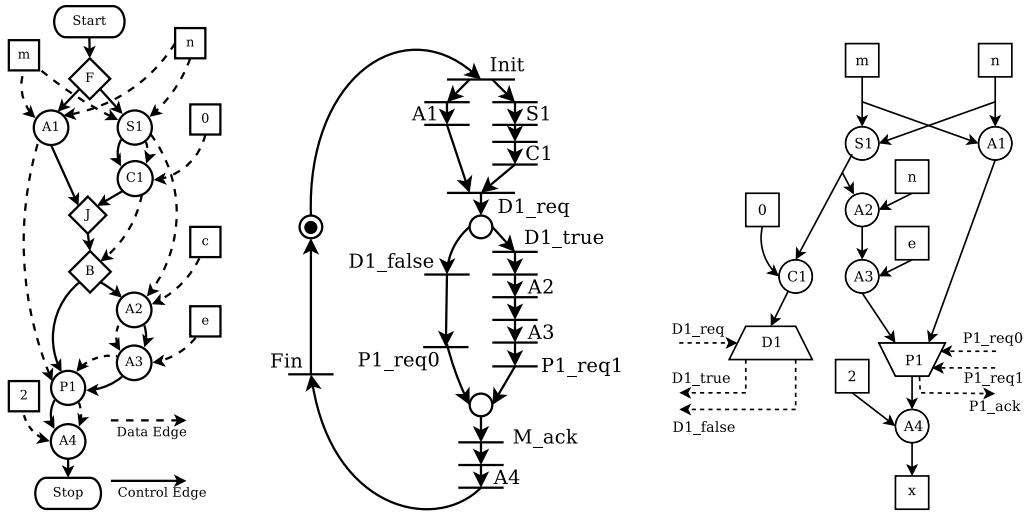
The synthesis flow uses the LLVM framework to parse and optimise the input program. This is translated to an AHIR specification using a CDFG as an intermediate step.

A. C to CDFG

The C program is first converted to LLVM bytecode, which is based on the SSA form[11]. This is a purely functional form that removes the notion of individual *variables* from a program. Every assignment to a variable is a distinct value; multiple assignments that occur in branches are handled by a special instruction called the ϕ-function, as shown in Fig. 6.

The LLVM bytecode is then translated to a control data flow graph (CDFG)[12]-[16], as shown in Fig. 7(a). The CDFG represents instructions as nodes connected by control and data flow edges. The edges in the CDFG arise from three kinds of dependences in the original program:

- Data dependences that create control and data flow.
- Control structures that create control flow.
- External dependences (in storage) that create control flow.



(a) CDFG. (b) Control Path. (c) Data Path.

Fig. 7. Translating a CDFG to AHIR.

B. Generating AHIR from a CDFG

The AHIR specification is generated by piece-wise translation. Each node or edge in the CDFG is replaced by an equivalent *AHIR fragment*, and the fragments are connected to obtain the control and data paths. Fig. 7 shows the input CDFG and the resulting AHIR specification for our example. Elements that are obvious from the context have been hidden. Two structures are shown in detail — a decoder element (`D1`) that examines the condition for a branch, and a multiplexer element (`P1`) that implements a ϕ-function.

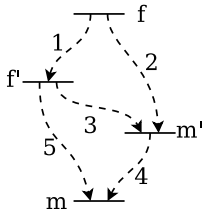
C. Correctness

The method used by our synthesis flow guarantees that the generated circuit specification is correct *by construction*. The first step of translating a C program to a CDFG is a routine one that does not need to be verified separately. The second step of generating an AHIR specification A from the CDFG G is shown to be correct by recovering a second CDFG G' from A such that:

1) A is an implementation of G'.
2) G' is isomorphic to G.

Statement (1) is true since G' was recovered from A. To demonstrate the isomorphism in Statement (2), we use a simple labelling scheme as follows.

Every element (node and edge) in G is assigned a unique label. When generating A, the labels in G are used to label the corresponding AHIR fragments. This results in a labelled AHIR specification A, where the fragments can be identified by their labels. Each such fragment is replaced with a CDFG element to obtain a new CDFG G'. Each element in G' is assigned a label derived from the corresponding fragment. These labels can now be used to demonstrate that G' is isomorphic to the original CDFG G.



(a) A fork region

	1	2	3	4	5
1	Y	-	Y	Y	Y
2	-	Y	-	Y	-
3	Y	-	Y	Y	-
4	Y	Y	Y	Y	-
5	Y	-	-	-	Y

(b) Compatibility

Fig. 8. Compatibility in a Type-2 petri-net.

VI. Arbiterless sharing of data-path operators

We describe an optimisation that reuses a data-path operator for multiple operations that are never active at the same time. This avoids arbitration overheads since there is no contention. The optimisation uses an almost linear static analysis of the control-path to identify sharing opportunities.

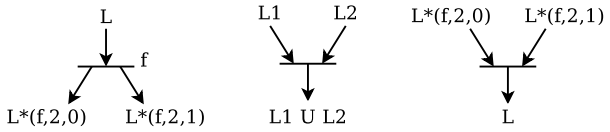
Definition 6.1: An operator is said to be **active** at a given instant of time if and only if it has received a request, but not yet emitted an acknowledgement.

Definition 6.2: Two operators M_1 and M_2 are said to be **compatible** if and only if M_1 does not receive a request while M_2 is active, and *vice versa*.

A. Compatibility in Type-2 petri-nets

In a Type-2 petri-net, compatibility of two transitions is determined by the nature of the smallest STPR that contains them, which we term as their *nearest common ancestor* (NCA). Two operations can potentially be incompatible only if the NCA is a fork region; operations in a branch or series NCA region are always compatible.

Fig. 8 shows a fork region in a Type-2 petri-net with numbered segments and their compatibility with each other. Segments 1 and 2 are *incompatible* since they can execute concurrently. But segments 1 and 4 are compatible, since a sequence is enforced by the path through segment 3.



(a) Fork. (b) Union at a join. (c) Reduction at a join.

Fig. 9. Labelling scheme.

Definition 6.3: Two elements e_1 and e_2 in a Type-2 petri-net are compatible if and only if one of the following is true:

1) their NCA is *not* a fork region
2) there is a path within the NCA fork region, joining the two elements

B. Compatibility labelling

We use a labelling scheme to record the paths reaching a petri-net element from the init transition. The labelling is a symbolic execution of the Type-2 petri-net. Two elements can be compared for compatibility using their labels instead.

A label is a set $L = \{l_0, l_1, \ldots\}$ where each $l \in L$ is a sequence of n label elements $l = [a_0, a_1, \ldots, a_{n-1}]$. A label element is a 3-tuple (f, k, i) made of a fork identifier f, the fan-out k of the fork, and an index i into the fan-out. A label element $a = (f, k, i)$ is said to *indicate* the fork f.

Common operators such as equality $(A = B)$, concatenation $(a.b)$ and prefix $(a \leq b)$ are defined to have sensible meanings. The prefix operator defines a partial order on label sequences. The longest common prefix (LCP) of two sequences is the longest sequence that is a prefix of both the sequences. The product of two labels $(A * B)$ is defined as the concatenation of pairs of sequences from the two labels: $A * B = \{a.b | \forall a \in A, \forall b \in B\}$. For convenience, the product operator is overloaded to represent the product of a label with a single element: $A * b = A * \{[b]\}$.

C. Labelling scheme

Parallel-merge regions are first reduced to simple merges, which simplifies the labelling without affecting compatibility. The $init$ transition is assigned an empty label. The label of every element is computed from its predecessors. Only forks and joins result in a new label; other elements are assigned the same label as their predecessors.

1) Labelling at a fork: If L is the label assigned to a fork f with $k(f)$ successors, then each successor s_i is assigned the label $L * \big(f, k(f), i\big)$, as shown in Fig. 9(a).

2) Labelling at a join: In the general case — such as transition m' in Fig.8(a) — a join is assigned the union of the labels assigned to all its predecessors, as shown in Fig. 9(b). But when the join receives all the tokens starting from a particular fork — such as transition m in Fig.8(a) — the union is reduced to remove the label elements indicating that fork, as shown in Fig. 9(c). Usually, only a subset of the union is reduced, since paths from unrelated forks may reach a join.

The reduction at the join ensures that the labelling scheme is "closed" — all the extensions created within a fork region disappear at the exit of a fork region[2]. Finally, the fin transition is assigned an empty label.

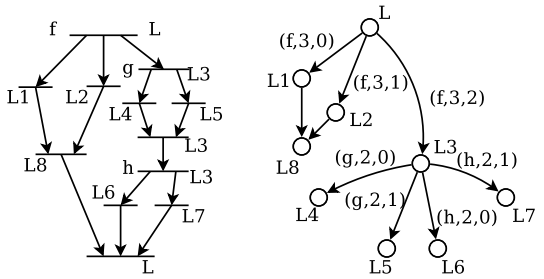


Fig. 10. Label Representation Graph.

D. Label Representation Graph (LRG)

The compatibility label is a record of every path reaching that element from the $init$ transition, which results in an exponential size. Comparing two labels for compatibility also has exponential complexity, since every sequence in one label has to be compared with every sequence in the other label.

We eliminate the complexity by using a *label representation graph* (LRG) as shown in Fig. 10. The LRG represents labels as nodes, where edges represent the manner in which a label is constructed from other labels. The LRG is a directed acyclic graph with a single root node that represents the empty label.

Let $l(u)$ be the label represented by a node u. An edge (u, v) in the LRG may itself be labelled with a label element e, in which case it represents the product operation $l(v) = l(u) * e$. If multiple incoming edges are incident at a node v, then it represents a label that is the union of all its predecessors. In a well-formed LRG, multiple incoming edges are incident on a node if and only if they are all unlabelled.

The LRG is a compact representation of compatibility labels in a Type-2 petri-net. Each path reaching a node u from the root of the LRG represents one label sequence in the label $l(u)$. The following result is proved in [2].

Theorem 6.1: Two operations with labels represented by nodes u and v in the LRG are said to be compatible, if and only if one of the following is true:

1) There is a path from u to v or *vice versa*.
2) There exists a node a in the LRG, from which u and v are reachable along non-intersecting paths such that one of the following is true:
   a) One or both paths begin with an unlabelled edge.
   b) The labels on the first edges in the paths indicate different forks.

This allows us to check two operations for compatibility using a DFS-based algorithm defined on the LRG that is almost linear in complexity.

E. Shared operators

We have used the LRG to implement arbiterless sharing in the AHIR synthesis flow. We use a simple greedy algorithm to generate cliques of pair-wise compatible operations that are

mapped to a single operator in the data-path. The incoming data-edges are routed through multiplexers; the registers for the outgoing data-edges are not shared.

This scheme for arbiterless sharing is quite effective in reducing hardware costs, demonstrating support for scalable optimisations in AHIR. Synthesis results show improvements in the throughput-area ratio (measured in $\mathrm{Hz}\,/\,\mathrm{slice}$) in the range of 15–190% depending on the application[2].

VII. Evaluation of the end-to-end synthesis flow

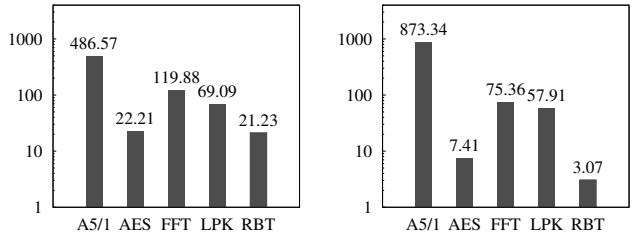
We have implemented a complete C-to-ASIC tool-flow based on commercial synthesis tools from Cadence and Synopsys. The flow translates C programs to ASIC implementations for the TSMC 180nm technology, using the OSU standard cell library and CACTI models for SRAM[17]. We compare the performance of the ASIC implementations with an industry-standard low-power microprocessor, the Intel Atom N270.

The same set of C programs was run on the Intel Atom, and also translated to hardware: A5/1 stream cipher, AES block cipher, 64-point FFT, Linpack and Red-Black Trees. The delay in each case is the time taken to finish one task specific to that program — generating one key bit in A5/1, encrypting one block in AES, generating one 64-point FFT, solving a 100×100 system in Linpack, and inserting one thousand nodes in a Red-Black Tree. The values for area, frequency and power dissipation for the Atom processor were obtained from the corresponding datasheet. The measurements for the AHIR circuits were scaled from the 180nm data to match the 45nm technology used in the Atom processor.

TABLE I
Comparison with the Intel Atom N270

	Area (mm^2)	Freq (MHz)	Delay (ms)	Power (mW)	Energy (μJ)	E×D
A5/1-Atom	25	1600	0.12μs	2500	298.44 nJ	35.63
A5/1-AHIR	0.10	285	0.07μs	9.22	0.61 nJ	0.04
AES-Atom	25	1600	0.036	2500	89.362	3.194
AES-AHIR	0.41	285	0.107	37.56	4.023	0.431
FFT-Atom	25	1600	0.022	2500	55.64	1.238
FFT-AHIR	0.32	180	0.035	13.11	0.464	0.016
LPK-Atom	25	1600	7.90	2500	19740	155875
LPK-AHIR	1.69	165	9.42	30.33	285	2691
RBT-Atom	25	1600	0.36	2500	891.89	318.19
RBT-AHIR	1.13	165	2.47	17.00	42.01	103.80

In Table I, we show the results of experiments that compared the generated circuits with the Intel Atom N270 processor. The energy used for completing a job is equivalent to the throughput achieved for each watt of power supplied, commonly termed as "performance per watt". The comparative graph in Fig. 11(a) shows that the AHIR circuits are better than the Intel Atom by at least an order of magnitude.



(a) Energy $\left(\frac{\text{Atom}}{\text{AHIR}}\right)$ (b) Energy × Delay $\left(\frac{\text{Atom}}{\text{AHIR}}\right)$

Fig. 11. Comparison of AHIR circuits with the Intel Atom N270

VIII. Conclusion

We have established a competitive pathway from high-level programs to hardware that is correct by construction and also scalable to large systems. The scalability is made possible by an explicit factorisation of the intermediate representation into three components: control, data and storage. Our experiments demonstrate that the circuits generated using our compiler are competitive with industry-standard low-power microprocessors, thus providing a scalable and efficient alternative for implementing large systems in hardware.

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 Int. Conf. on VLSI Design*, January 2007.

[2] S. D. Sahasrabuddhe, "A competitive pathway from high-level programs to hardware." Ph.D. dissertation, IIT Bombay, 2009.

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

[4] 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.

[5] S.-B. Scholz, "Single Assignment C – Efficient Support for High-level Array Operations in a Functional Setting." in *Journal of Functional Programming*. Cambridge University Press, 2003, pp. 1005–1059.

[6] I. Page, "Closing the Gap between Hardware and Software: Hardware-software cosynthesis at Oxford," in *IEE Colloquim on Hardware-Software Cosynthesis for Reconf. Systems*, February 1996, pp. 2/1–2/11.

[7] M. Budiu and S. C. Goldstein, "Pegasus: An Efficient Intermediate Representation," School of Comp. Sci., CMU, Tech. Rep., April 2002.

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

[9] 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.

[10] 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, no. 2, February 2004.

[11] S. S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[12] M. Rim and R. Jain, "Representing Conditional Branches for High-Level Synthesis Applications," in *Proc. of the 29th DAC*, 1992, pp. 106–111.

[13] G. G. Jong, "Data Flow Graphs: System Specification With the Most Unrestricted Semantics," in *Proceedings of the European Conference on Design Automation*. IEEE, 1991, pp. 401–405.

[14] J. T. van Eijndhoven and L. Stok, "A Data Flow Graph Exchange Standard," in *Proc. of the Euro. Conf. on Des. Auto.*, 1992, pp. 193–199.

[15] D. D. Gajzki and A. Orailoglu, "Flow Graph Representation," in *Proceedings of the 23rd DAC*. IEEE, 1986, pp. 503–509.

[16] S. Amellal and B. Kaminska, "Functional Synthesis of Digital Systems with TASS," in *IEEE Trans. on Computer-Aided Design of Int. Circuits and Systems*. IEEE, May 1994, vol. 13, no. 5, pp. 537–552.

[17] K. P. Ghosh and S. Subramanian, "Power / Delay Estimation of Auto Generated Circuits," Department of Electrical Engineering, IIT Bombay, Tech. Rep., April 2009.