A C-to-RTL Flow as an Energy Efficient Alternative to Embedded Processors in Digital Systems

5 authors, including:

Kavi Arya
Indian Institute of Technology Bombay
19 PUBLICATIONS   **112** CITATIONS

Madhav Desai
Indian Institute of Technology Bombay
60 PUBLICATIONS   **761** CITATIONS

A C-to-RTL flow as an energy efficient alternative to embedded processors in digital systems

Sameer D. Sahasrabuddhe, Sreenivas Subramanian, Kunal P. Ghosh, 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— **We present a high-level synthesis flow for mapping an algorithm description (in C) to a provably equivalent register-transfer level (RTL) description of hardware. This flow uses an intermediate representation which is an orthogonal factorization of the program behavior into control, data and memory aspects, and is suitable for the description of large systems. We show that optimizations such as arbiter-less resource sharing can be efficiently computed on this representation. We apply the flow to a wide range of examples ranging from stream ciphers to database and linear algebra applications. The resulting RTL is then put through a standard ASIC tool chain (synthesis followed by automatic place-and-route), and the performance and power dissipation of the resulting layout is computed. We observe that the energy consumption (per completed task) of each resulting circuit is considerably lower than that of an equivalent executable running on a low-power processor, indicating that this C-to-RTL flow offers an energy efficient alternative to the use of embedded processors in mapping algorithms to digital VLSI systems.**

I. INTRODUCTION

The design of complex digital systems is expensive due to two reasons: the need for trained manpower, and the difficulty of verification at every step of the design process. Thus, it is often the case that low-power embedded microprocessors are used to implement complex algorithms in digital systems, so that the design and verification problems are moved to the software domain. A high-level synthesis flow which maps an algorithm directly to a hardware description can be a potential alternative if it provides a verifiable and optimizable path from executable specifications[1] to hardware.

In addition to the ease of use, performance measures and energy considerations are standards by which a high-level synthesis flow should be judged relative to the competing methodologies — custom designed RTL and embedded processors. It is very critical that the results of the high-level synthesis flow are characterized with respect to their area/power/delay/energy, both in absolute terms and relative to competing methodologies. As far as comparisons with custom designed hardware are concerned, a high-level synthesis flow which starts from a C/C++ description of the algorithm is unlikely to be competitive, mainly because the amount of parallelism expressible in a C/C++ program is not sufficient. It should be possible to address this shortcoming by starting with algorithms described in parallel programming languages, but such discussions are beyond the scope of this paper.

We provide comparisons between the hardware generated by our high level synthesis flow and processor implementations of an algorithm described in C. Note that it is not entirely obvious that algorithm-specific hardware generated by a high-level synthesis flow will be superior to embedded processor implementations of the same algorithm. One can argue that the algorithm-specific hardware can exploit the characteristics of the algorithm to a greater degree than a general purpose processor. On the other hand, when a C/C++ program is compiled to a processor, the platform (that is, the processor) is usually a heavily optimized circuit in which each functional element has been tweaked to a high level, whereas the algorithm-specific circuit is generated by automated tools whose optimization abilities will most probably not match those of a human designer. Finally, since it is not possible to express parallelism directly in a C program (although a compiler can infer it), it is not too clear whether a high-level synthesis system can identify enough parallelism to make the resulting hardware competitive with a heavily optimized processor. Concrete data which throws light on these aspects is valuable.

Our high-level synthesis flow is based on a factored intermediate representation which guarantees the correctness of the implementation, and supports optimizations that can scale with the size of the program. 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 optimizations[2][3].

In order to characterize the energy efficiency of the resulting circuits, we select a wide range of applications (from stream ciphers to databases and linear algebra) and run them through our C-to-RTL flow. The resulting RTL is then mapped to an ASIC using industry standard automatic synthesis tools. This ASIC is characterized for power, delay, area and energy, and the results are compared with those obtained when the same program is mapped to a low power processor. The results indicate that even in its current state, our high-level synthesis flow is an energy competitive alternative to the use of embedded processors in the design of complex systems.

II. RELATED WORK

There have been several approaches to the creation of a path from high-level programs to hardware descriptions, which can be loosely categorized as follows:

A. Improvements over RTL

Efforts such as 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 utilize 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 reevaluate standard programming practices.

C. High-level programs as hardware specifications

Some efforts interpret a program as a behavioral 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 micro-pipeline. 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 factorizes the system into three separate components: control flow, data flow and memory. The components can be optimized and implemented separately as long as specified constraints are satisfied.

Several commercial tools which translate a C program to hardware are also available, such as Catapult-C[1] from Mentor Graphics, C-to-Silicon[2] from Cadence, and Synopsys HLS[3] from Synopsys. Currently, we have no access to the internal details of these proprietary tools.

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 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*.
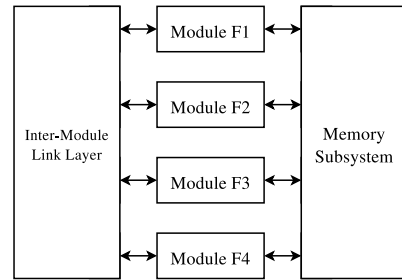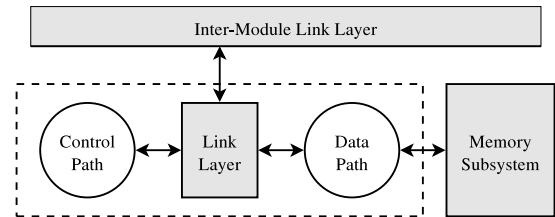
Fig. 1.   An AHIR system.

Fig. 2.   An AHIR module.

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.

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 circuit-level implementation paradigm to be used for an AHIR

Fig. 3.   Delays in an AHIR specification.

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

(a) Primitive   (b) Series   (c) Fork   (d) Branch

(e) Parallel merges

Fig. 5.   Type-2 construction rules.
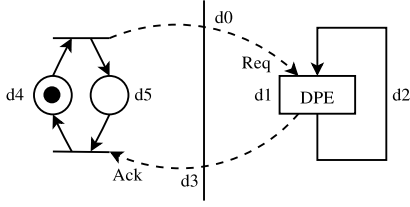
module is not specified (the implementation may be synchronous or asynchronous). Thus, petri-nets provide a neutral, powerful and compact mechanism to describe parallelism in the control flow of an algorithm. As we shall see in Section IV, a "Type-2" petri-net provides an easily verifiable, yet powerful enough mechanism to describe control flow.

The transitions in the control path are associated with input and output symbols. An *input transition* is associated with an input symbol (that is, the transition is assumed to have fired when the input symbol is received by the control path), and an *output transition* is associated with an output symbol (the output symbol is emitted when the transition fires).

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 acknowledgment 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 request in Λ, which triggers an operator in the data-path. When done, the operator emits an 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 acknowledgment symbol Ack. Hence we have:

$$d_5 \le d_0 + d_1 + d_3$$

Similarly, when the data-path emits an Ack, it eventually receives a Req. The data inputs must have stabilized before this request arrives. Hence we have:
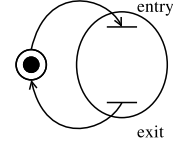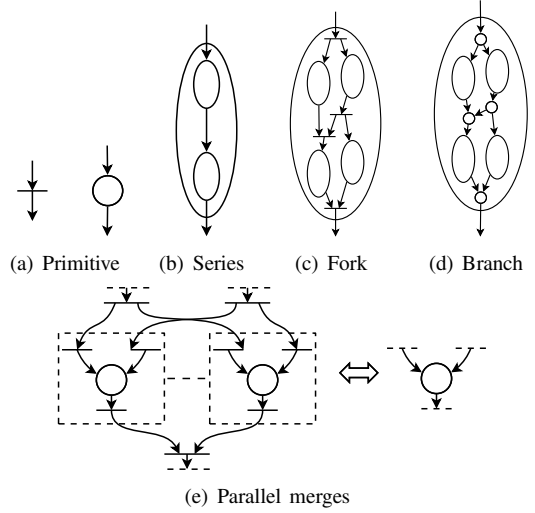
$$d_2 \le d_3 + d_4 + d_0$$

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.

F. Execution model

AHIR uses the following execution model. The control-path responds instantaneously to the arrival of symbols, while data-path elements take some finite non-zero amount of time 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. In principle, RTL implementations of AHIR can be synchronous or asynchronous. In our current flow however, we generate synchronous RTL implementations from AHIR.

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

Control paths in AHIR are instances of Type-2 petri-nets (introduced in [3]), which are defined using a set of standard construction rules, which ensure that the resulting petri-net is live and safe. Further, the structure of a Type-2 petri-net enables 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.

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* sub-graph made of STPRs, forks and joins is itself an STPR called a *fork region*.
4) A connected (possibly cyclic) sub-graph 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 petrinets. Parallel-merge regions implement the run-time selection of values at the exit of a branch, such as variable d in Fig. 9.

V. Arbiter-less sharing of data-path operators

We describe an optimization 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 optimization uses an almost linear static analysis of the control-path to identify sharing opportunities.

Definition 5.1: 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 acknowledgment.

Definition 5.2: 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. 6 shows a fork region in a Type-2 petri-net with numbered segments and their compatibility with each other.
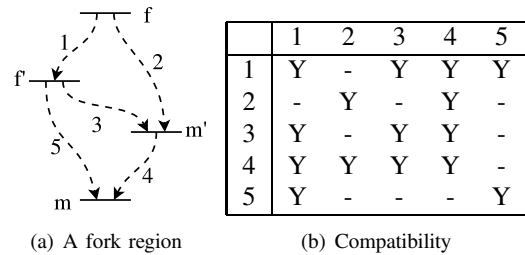

	1	2	3	4	5
1	Y	-	Y	Y	Y
2	-	Y	-	Y	-
3	Y	-	Y	Y	-
4	Y	Y	Y	Y	-
5	Y	-	-	-	Y

(a) A fork region   (b) Compatibility

Fig. 6. Compatibility in a Type-2 petri-net.

(a) Fork.   (b) Union at a join.   (c) Reduction at a join.
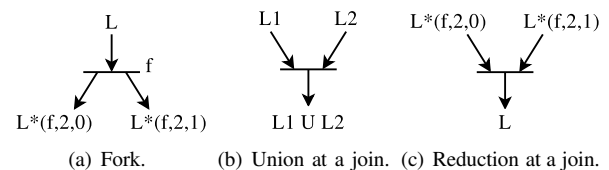
Fig. 7. Labeling scheme.

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.

Definition 5.3: Two elements e_1 and e_2 in a Type-2 petrinet are compatible if their NCA is *not* a fork region or if there is a directed path within the NCA region which joins the two elements.

B. Compatibility labeling

We use a labeling scheme to record the paths reaching a petri-net element from the init transition. The labeling 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.

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. Labeling scheme

Parallel-merge regions are first reduced to simple merges, which simplifies the labeling 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) Labeling 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 $L * (f, k(f), i)$, as shown in Fig. 7(a).

2) Labeling at a join: In the general case — such as transition m' in Fig.6(a) — a join is assigned the union of the labels assigned to all its predecessors, as shown in Fig. 7(b).
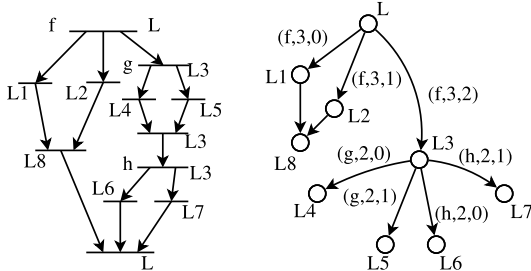
Fig. 8.   Label Representation Graph.

(a) Pseudo-code. (b) SSA form.

Fig. 9.   A code fragment and its SSA form.

But when the join receives all the tokens starting from a particular fork — such as transition m in Fig.6(a) — the union is reduced to remove the label elements indicating that fork, as shown in Fig. 7(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 labeling scheme is "closed" — all the extensions created within a fork region disappear at the exit of a fork region[3]. Finally, the fin transition is assigned an empty label.

D. Label Representation Graph (LRG)

The compatibility label is a record of every path reaching that element from the *init* transition, and can thus have an exponential size (the number of elements in the label set can be exponential in the size of the petri-net). Comparing two labels for compatibility can also have exponential complexity, since every sequence in one label has to be compared with every sequence in the other label.

We eliminate the exponential complexity by using a *label representation graph* (LRG) as shown in Fig. 8. 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 labeled 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 unlabeled.

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 [3].

Theorem 5.1: 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 unlabeled edge.

b) The labels on the first edges in the paths indicate different forks.

E. Shared operators

We have used the LRG to implement arbiter-less 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 arbiter-less sharing is quite effective in reducing hardware costs, demonstrating support for scalable optimizations in AHIR. Synthesis results (targeted at field programmable gate arrays) show improvements in the throughput-area ratio (measured in Hz / slice) in the range of 15–190% depending on the application[3].

VI. THE SYNTHESIS FLOW

The synthesis flow uses the LLVM Compiler Framework[4] to parse and optimize the input C program. The resulting optimized LLVM bytecode is then translated to an AHIR specification using a CDFG as an intermediate step.

A. C to CDFG

The starting point is a C program. In the current implementation, the only restrictions on the C program are that the call graph should not contain any cycles, and that function pointers should not be used (pointers to data can be used in the usual way)[5]. The C program is first converted to LLVM byte-code, 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. 9.

The LLVM bytecode is then translated to a control data flow graph (CDFG)[12]-[16], as shown in Fig. 10(a). The CDFG is a hypergraph where the nodes represent instructions and edges represent control and data flow. A data edge is a hyperedge with a single tail and one or more heads, that represents a value in the program, defined by the tail and used by the heads. The control edge has a single tail and a single head, and represents the passing of control from the tail to the head.

---

[4] http://llvm.org/

[5] These restrictions are not fundamental to the approach, but we have not gotten around to eliminating them so far.

Each value in the program that is defined by some instruction u and used by a set of instructions s_v is mapped a data edge in the CDFG where the tail n_u corresponds to the instruction u and the set of heads has a one to one correspondence with the set s_v. Control edges are created in a manner that exposes instruction-level parallelism in the program. Sequence is enforced only where the execution of a node may affect execution of some other node. For example, when a node v uses the output of a node u, a control edge is introduced to ensure that v is executed only when u is finished. Similarly, control edges are used to eliminate hazards between pairs of load instructions, store instructions and function calls that may access the same memory location, as determined by a memory reference analysis.

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. 10 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 correctness of the second step (CDFG to AHIR) is shown as follows[3]: suppose that A was the AHIR specification generated from the reference CDFG G. Then every execution sequence in A is a member of the set of possible execution sequences of G. Now, from A, we construct a new CDFG G' such that every execution sequence of G' is an element of the set of execution sequences of A. Now, G' and G are shown to be isomorphic, so that their sets of execution sequences are in a one-to-one correspondence with each other. This shows that the sets of execution sequences in A and G are in a one-to-one correspondence.

D. AHIR to VHDL

The conversion of an AHIR circuit to VHDL is straightforward. Currently, we produce a system which is synchronous and uses a single clock (positive edges only). Each symbol exchanged between the control path and the data path is coded by a pulse which is sampled high by exactly one clock edge. Symbols are associated with transitions as indicated earlier.

Places in the AHIR control path are handled as follows:

1) If a place has a single predecessor transition which is an output transition and a single successor which is an input transition, then it is optimized away.
2) If a place has a single successor transition which is an output transition, and if this successor transition is not a join, then the place is modeled by an OR gate whose inputs are its predecessor transitions.
3) In all other cases, the place is modeled as a finite state machine with two states, with the state being set by any of its predecessors, and reset by any of its successors.

The number of places that need to be modeled by state machines is usually a small fraction of the total number of places. Output transitions are modeled by AND gates, with each predecessor place to the transition providing an input to the AND gate. The cost of implementing the control path is thus minimal in relation to the implementation of the data path and memory subsystem.

Operators in the data path compute their results in a single clock cycle, and the outputs are registered. Shared operators are constructed using multiplexers in a standard manner. The instantiation of operators in the data path, the interconnections within the data path and those between the data path and the control path mirror those in the AHIR representation, except for the fact that several operators in the AHIR data path can be mapped to the same physical operator (through arbiterless sharing). The inter module link layer is completed using the call graph corresponding to the source code.

The memory sub-system is implemented as a multiple-bank (each bank is a single-cycle synchronous SRAM) pipelined memory which offers as many load/store ports as there are load/store operators across the data paths. The memory sub-system is constructed using a request-complete protocol and ensures that read/write operations complete in the order that they were requested. The number of memory banks in the memory subsystem, and the degree of pipe-lining are parameters/generics that can be selected based on the memory reference characteristics of the application.

VII. EXPERIMENTAL 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. We selected the following set of applications for the experiments (the task used to characterize the energy dissipation is also indicated).

1) A5/1, a simple stream cipher (basic task: produce one key bit).
2) AES, the standard AES-128 block cipher (basic task: encrypt a 128-bit block).
3) LPK, the standard Linpack benchmark (basic task: LU factorization of 100×100 matrix with floating point entries).
4) FFT (basic task: 64 point floating point FFT).
5) RBT, the red-black tree data structure (basic task: insertion of 1000 elements into the tree).

In each case, the VHDL generated by the C-to-RTL flow was synthesized using Synposys Design Compiler, using the OSU Standard Cell Library[6] for the TSMC 0.18μ technology. Memories were modeled using CACTI SRAM models[17][7].
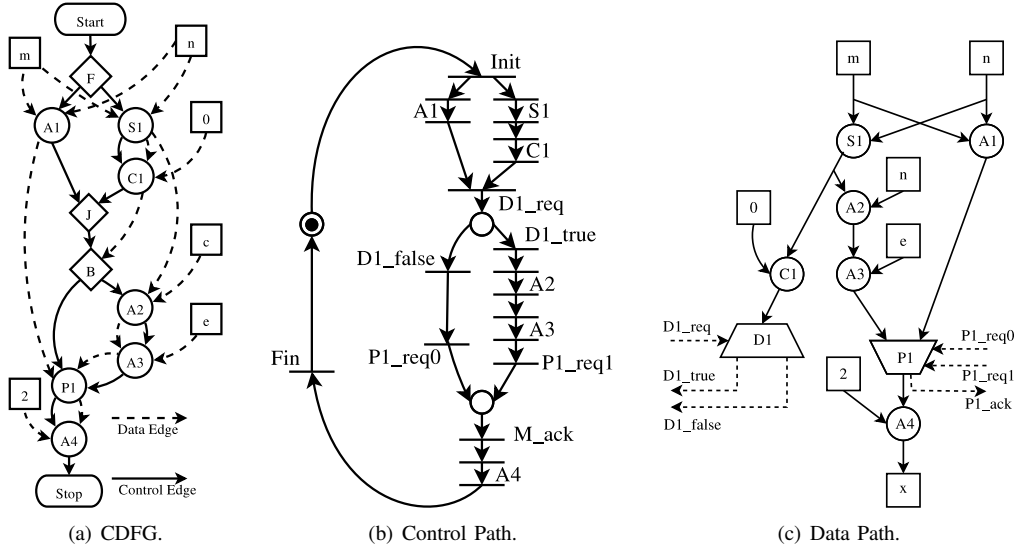

(a) CDFG.   (b) Control Path.   (c) Data Path.

Fig. 10.   Translating a CDFG to AHIR.

The resulting net-list was mapped to layout using the Cadence SOC Encounter tool. No human intervention was used in the entire process from C to the final layout (other than specifying the operating frequency to the automatic place-and-route tools). The total time needed for the entire process (from C to layout) was of the order of a few minutes for A5/1 to a few hours for the LPK case. A simulation based toggle file was then used with the extracted net-list to estimate power dissipation in each of the generated circuits. The reciprocal of the energy used for completing a task is equivalent to the throughput achieved for each watt of power supplied, commonly termed as *performance per watt*

TABLE I

AREA/DELAY/POWER/ENERGY DATA FOR AHIR CIRCUITS IN THE TSMC 0.18μ TECHNOLOGY

	Area (mm^2)	Freq (MHz)	Delay (ms)	Power (mW)	Energy (μJ)
A5/1	1.45	71.4	0.28μs	73	20.44 nJ
AES	6.5	71.4	0.428	338	144.7
FFT	5.1	41.7	0.155	95	14.7
LPK	27	41.7	37.7	273	10300
RBT	18	41.7	9.9	153	1511

As a reference to judge the energy efficiency of the AHIR generated circuits, we consider a popular low power processor (henceforth denoted by P), namely the Intel Atom N270 processor[8], which is built in Intel's $45nm$ technology. This processor is readily available, combines high performance with low power dissipation, has full floating point support, and is a part of many low power computing devices. The sample set of C programs was run on the processor P (exactly the same program used in the hardware generation was run on the

processor). In each case, the number of cycles needed to finish one task (as described above) was measured. The data sets for the programs were small enough to fit into the processor cache, and the cycle count was averaged across multiple runs. These cycle counts were then used to find the delay and energy consumption for the processor. All values for area, frequency and power dissipation for the processor were obtained from the corresponding data-sheet.

TABLE II

AREA/DELAY/POWER/ENERGY RESULTS FOR THE PROCESSOR P BUILT IN A $45nm$ PROCESS

	Area (mm^2)	Freq (MHz)	Delay (ms)	Power (mW)	Energy (μJ)
A5/1	25	1600	0.12μs	2500	298.44 nJ
AES	25	1600	0.036	2500	89.362
FFT	25	1600	0.022	2500	55.64
LPK	25	1600	7.90	2500	19740
RBT	25	1600	0.36	2500	891.89

We find that the energy consumed per task by the $180nm$ AHIR circuits is appreciably lower than that for the $45nm$ processor P (except for AES and RBT). This is significant because the processor P is implemented in a technology which is four generations ahead of the AHIR circuits.

Now, suppose that we scale the energy numbers assuming migration of the AHIR circuits from $180nm$ to the $45nm$ technology node, using the following scaling rules (assume that $S = 45/180 = 0.25$):

- The operating voltage is scaled from $1.8V$ to $0.9V$.
- The internal capacitances are assumed to be scaled by S.
- The operating frequency of the generated circuit is scaled by $1/S$.
- Half of the total power dissipation of the $45nm$ circuits is due to leakage[18], and the other half is due to switching

power. Note that this is a conservative estimate, and the actual leakage is likely to be lower at the power levels corresponding to the AHIR circuits. The leakage power is a negligible fraction of the total power dissipation in the $180nm$ technology.

- The area is scaled by S^2.

Thus, the per task energy consumption due to switching activity alone will scale by $1/16$, and since we are assuming that leakage power dissipation is half the total, the total per task energy consumption of the circuit will scale by $1/8$. The area of the circuits will scale by $1/16$. The power dissipation will scale by $1/2$ (this would have been $1/4$ without leakage). This scaling, though approximate, can be justified since the area of the resulting circuits (in each case) is small, so that interconnect effects are not likely to be pronounced.

TABLE III
AREA/DELAY/POWER/ENERGY RATIOS (PROCESSOR P VALUES RELATIVE TO THE SCALED AHIR VALUES)

	Area	Freq	Delay	Power	Energy
A5/1	275.8	5.6	1.7	68.5	116.6
AES	61.5	5.6	0.34	14.8	4.9
FFT	78.4	9.6	0.57	52.6	30.3
LPK	14.8	9.6	0.84	18.3	15.3
RBT	22.2	9.6	0.15	32.7	4.7

In Table III, we tabulate the ratios of the performance parameters area, frequency, delay, power and energy for the processor P relative to the scaled values for the AHIR circuits assuming the scaling rules shown above. From the table, we see that across all applications, the energy consumed by AHIR circuits is considerably lower than the processor (the energy per task for the processor is between $4.7X$ and $116.6X$ that of the corresponding AHIR circuit). The improvement is highest in the case of bit-level manipulation applications such as A5/1, and lowest in the case of random "pointer-chasing" code. Note that the improvement in the case of floating point programs (FFT and LPK) is also substantial. The delays in completing the tasks are lower for the processor P in almost all the cases. This is mainly due to the fact that currently, the operators used in the AHIR generated RTL are not pipelined or optimized in any manner during the logic synthesis and physical design. This limits the operating clock frequency of the circuits synthesized from the AHIR generated RTL.

VIII. CONCLUSION

We have presented a high-level synthesis flow that converts an algorithm described in C into a VHDL description. The synthesis flow uses a factored control-data-storage internal representation and is correct by construction. On this internal representation, optimizations such as arbiter-less operator sharing can be carried out in an efficient manner. The flow can thus be applied to large programs describing a wide variety of algorithms. Our experiments demonstrate that across this variety of algorithms, an ASIC implemented using the RTL generated by this high-level synthesis flow uses a significantly lower energy per task (or equivalently, has higher performance per watt) than industry-standard low-power microprocessors. Thus, the range of algorithms which can be handled, and the ease of obtaining an implementation which is likely to be more energy efficient than a processor indicate that this high-level synthesis flow provides an alternative to embedded processors for implementing complex algorithms in hardware.

REFERENCES

[1] 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.

[2] 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*, January 2007, pp. 245–250.

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

[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 Reconfigurable Systems*, February 1996, pp. 2/1–2/11.

[7] M. Budiu and S. C. Goldstein, "Pegasus: An Efficient Intermediate Representation," School of Computer Science, Carnegie Mellon University, 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 *International Workshop on Logic & Synthesis*, 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 *Proceedings of the 29th ACM/IEEE Design Automation Conference*, 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 *Proceedings of the 3rd European Conference on Design Automation*. IEEE, 1992, pp. 193–199.

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

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

[17] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An Integrated Cache Timing, Power and Area Model," Western Research Laboratory, Palo Alto, Tech. Rep., August 2001.

[18] D. J. Frank, R. Puri, and D. Toma, "Design and CAD challenges in 45nm CMOS and beyond," in *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*. New York, NY, USA: ACM, 2006, pp. 329–333.