High-Level Synthesis of Asynchronous Systems by Data-Driven Decomposition

Catherine G. Wong
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125
wongca@async.caltech.edu

Alain J. Martin
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125
alain@async.caltech.edu

ABSTRACT

We present a method for decomposing a high-level program description of a circuit into a system of concurrent modules that can each be implemented as asynchronous pre-charge half-buffer pipeline stages (the circuits used in the asynchronous R3000 MIPS microprocessor). We apply it to designing the instruction fetch of an asynchronous 8051 microcontroller, with promising results. We discuss new clustering algorithms that will improve the performance figures further.

Categories and Subject Descriptors

B.7.1 [**VLSI**]; B.5.2 [**Automatic synthesis**]

General Terms

Design

Keywords

Asynchronous VLSI, high-level synthesis

1. INTRODUCTION

One obstacle to the widespread use of asynchronous VLSI is the lack of tools that can generate high-performance systems. The Caltech synthesis method begins by describing circuits using a high-level language, CHP [1]. Successive semantics-preserving program transformations are then applied, each generating a lower-level description of the circuit. The final output is a transistor netlist. This method is correct by construction, and every chip designed using this approach (including a 2M-transistor asynchronous MIPS R3000 microprocessor [8]) has been functional on first silicon.

The most difficult transformation, and one with a large effect on the speed and energy efficiency of the final system, is the first: process decomposition. In process decomposition, the original sequential CHP description of the circuit is broken up into a system of communicating modules (still expressed in CHP) that are then individually synthesized at lower levels. The inter-module communications mapped out during this
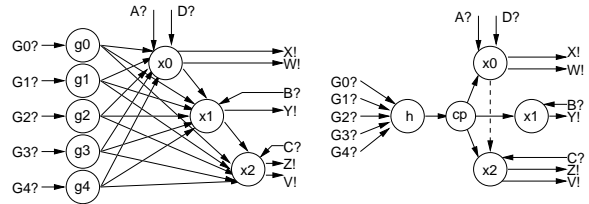
Figure 1: The system on the left is unoptimized while the one on the right was generated from the same program using the optimizations for energy efficiency introduced in this paper.

step consume the bulk of energy in the finished asynchronous system as data must be not only sent, but also validated and acknowledged. Designers rely greatly on experience and intuition to create an energy-efficient system.

CAD tools exist for process decomposition but they are mostly syntax-directed [3, 4, 5], or they begin with a lower-level specification than CHP [6, 7]. The designers of the asynchronous MIPS R3000 saw that very fine-grained pipeline stages were required to achieve high throughput, and abandoned the syntax-directed approach because it could not generate modules that were small enough. The approach described in this paper is data-driven decomposition (DDD), a high-level synthesis technique based not on syntax but on dataflow [2]. DDD generates modules that are small enough to be synthesized further into fine-grained pipeline stages; the decomposed systems are correct under the assumption of slack-elasticity [12].

This paper presents optimizations to the basic DDD method that create energy-efficient systems. The data dependency analysis is now more complex but results in significant improvements. A demonstration of the improvements made is given in figure 1. DDD has been applied to the design of the instruction fetch unit for an asynchronous 8051 microcontroller. The synthesized system runs at 102 MIPS and consumes 178 pJ per instruction. Automated tools implementing DDD are currently under development, and this paper also discusses new clustering algorithms to improve the performance figures further. Although developed for the synthesis of asynchronous systems, the method can also be useful for synchronous VLSI and FPGA implementations.
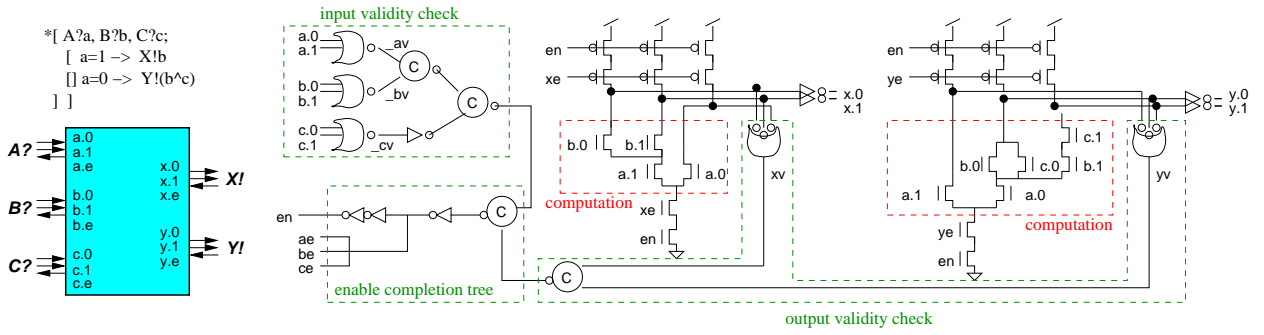
2. ASYNCHRONOUS CIRCUIT TEMPLATES

The fine-grained asynchronous pipeline stages employed in

Figure 2: A simple circuit that fits into the half-buffer template. The enable nodes (ae, be, ce, xe and ye) are inverted channel acknowledgment rails that are intially high.

the Caltech design style most commonly belong to the pre-charge half-buffer (PCHB) family [9]. The name of the circuit family refers to the specific interleaving of the input and output four-phase handshakes (set data; wait for acknowledge; reset data; wait for acknowledge reset) used to synchronize communications in lieu of a global clock signal. Circuits belonging to the PCHB family are not limited to simple buffers — they can be complex and include numerous channels, arithmetic functions, state, and both conditional inputs and outputs. Physical constraints (e.g., the maximum number of transistors allowed in series) combined with performance targets (e.g., cycle time) usually dictate the size of a single pipeline stage.

At the CHP level, all deterministic programs that fulfill the following conditions can be fit into the PCHB template: no nested loops; all input communications in the main loop appear before output communications begin; only one communication is allowed on each channel during every main loop iteration[1]. Communications on the same channel may appear in different branches of a selection statement (**if**-statement), as long as only one branch is ever executed per main loop iteration. If more communications are desired per iteration, state must be introduced.

An example of a simple process that fits this circuit template is given in figure 2. The communication channels are implemented using dual-rail code. Note that all of the computation in the circuit, including conditions, is handled by ten transistors in two pulldown networks. The rest of the circuit (transistors that pre-charge output nodes, check channel validities, and form completion trees for input enable rails and the local circuit enable) can be considered the communications overhead of the computation. It is clear why reducing the number of communication channels in a decomposed system also greatly reduces the system's energy consumption.

3. DATA-DRIVEN DECOMPOSITION

This section outlines the major steps of basic data-driven decomposition. In an effort to limit the size of the decomposed modules in DDD, each module implements all of the assignments to a single variable in the sequential program. Input communications are considered assignments to the input variable, and output communications are considered assignments to the output channel.

3.1 Dynamic Single Assignment Form

The sequential program is first converted to Dynamic Single Assignment (DSA) form. This conversion can reduce the number of operations performed on a single variable and thus help create DDD modules that fit into the PCHB circuit template. *A DSA program is one in which each variable is assigned a value at most once during a single iteration of the program's main loop.* Variables may have assignments in different branches of the same selection since only one branch is executed per iteration. However, if a variable has multiple assignments within the same branch, it must be split into separate DSA variables. All CHP programs can be systematically converted to DSA form.

As an example, a variable x with two assignments in series ($x := a$ and $B?x$) will be split into two new DSA variables (x_0, x_1):

$P \quad \equiv *[A?a; x := a, y := \neg a; X!x, Y!y; \quad B?x; Z!x]$
$P_{DSA} \equiv *[A?a; x_0 := a, y := \neg a; X!x_0, Y!y; \quad B?x_1; Z!x_1]$

Now x_0 and x_1 can each be decomposed into their own PCHB module.

3.2 Projection

Once the program is in DSA form, the technique of projection can be applied to break it up into a concurrent system of smaller modules. This involves assigning every variable and communication channel in the sequential program to a *projection set*. When the program is projected onto a set, a new module is created containing only the statements involving the variables and channels of that set.

For example, when the program P_{DSA} is projected onto the sets $\{A?, a, x_0, y, X!, Y!\}$ and $\{B?, x_1, Z!\}$, the resulting modules are $P1_{DSA} \equiv *[A?a; x_0 := a, y := \neg a; X!x_0, Y!y]$ and $P2_{DSA} \equiv *[B?x_1; Z!x_1]$. The correctness of the projection technique requires that the program be slack-elastic (i.e., that any amount of buffering can be added to any channel without affecting correctness). All deterministic programs and many others meet this criterion. DDD systematically creates intermediate channels and chooses projection sets through data dependency analysis described in [2]. Briefly, if variable x depends upon variable y and is also used in the computation of variable z, DDD introduces channels to send the value of y to the module computing x and to send the value of x to the module computing z. These channels are then included in the projection set for the module computing x.

To illustrate the mechanics of projection, consider the simple example program $P1_{DSA}$. Since we plan on projecting x_0 out

---

[1]Circuits repeat their specified behaviour indefinitely, and so the CHP programs that describe them are enclosed in a non-terminating loop.

onto its own module, the projection set for that module (Px_0) must contain all of the variables upon which the value of x_0 depends, including any input channel that inputs a value to x_0. Since both x_0 and y depend upon a, a must have two copies so that one can be projected out onto each process. Thus we rewrite $P1_{DSA}$ as

$$*[A?a; \quad a_{x_0} := a, a_y := a; \quad x_0 := a_{x_0}, y := \neg a_y; \quad X!x_0, Y!y]$$

(We refer to the assignments $a_{x_0} := a$ and $a_y := a$ as projection assignments since they have been added because of projection.) Then, we rewrite the projection assignments to introduce the intermediate channels that will be required between the modules in the decomposed system:

$$*[\quad A?a; \quad (Ax0!a \| Ax0?a_{x_0}), (Ay!a \| Ay?a_y);$$
$$x_0 := a_{x_0}, \quad y := \neg a_y; \quad X!x_0, \quad Y!y \quad]$$

Finally, we can perform projection on the sets $\{A?, a, Ax0!, Ay!\}$, $\{Ax0?, a_{x_0}, x_0, X!\}$ and $\{Ay?, a_y, y, Y!\}$. The resulting system is functionally equivalent to the original sequential program: $*[A?a; Ax0!a, Ay!a] \parallel *[Ax0?a_{x_0}; x_0 := a_{x_0}; X!x_0] \parallel *[Ay?a_y; y := a_y; Y!y]$.

4. SYNTHESIS FOR ENERGY EFFICIENCY

Reducing the number of communications in a system can greatly reduce the energy consumption. Of course, the specification of the original sequential program cannot be changed, and so the communications on external channels must remain the same. However, communications on internal channels introduced by process decomposition can be made conditional. This may decrease energy consumption in three ways: by reducing the wire load that is switched per cycle; by making entire modules conditional; by decreasing the actual number of channels in the system.

The first way is obvious. In the second way, if all of the input and output communications of a module only occur under a certain condition, then the module can stop computing the condition and simply perform the main computation when its data inputs all arrive. For example, the program $*[G?g; [g \rightarrow X?x; Y!(x + 1) \| \neg g \rightarrow \textbf{skip}]$ can be simply rewritten as $*[X?x; Y!(x+1)]$, since x will only be sent when a computation is required anyways. Finally, the above transformation is also an example of the third way, as the channel $G?$ has been eliminated from the decomposed system, along with its validity-check circuitry.

All examples below are simplified versions of situations that occur in the asynchronous microcontroller currently being designed at Caltech (Section 5).

4.1 Creating Conditional Communications

Consider the following program:

$$COND \equiv *[\quad G?g, \quad A?x; \quad Y!x;$$
$$[\quad g = 0 \longrightarrow B?b; x := f_1(x, b); \quad C?c; x := f_2(x, c)$$
$$\| \quad g = 1 \longrightarrow B?b; \quad W!b$$
$$\| \quad g = 2 \longrightarrow \textbf{skip}$$
$$]; \quad Z!x$$
$$]$$

First, when rewriting the program in DSA form, if a variable x is split within a selection statement into multiple DSA variables x_i, then only the last of these variables actually needs to have a defined value at the end of the selection. Intermediate DSA variables can be undefined in guarded commands where they are not required.

The DSA version of $COND$ is therefore

$$COND_{DSA} \equiv *[\quad G?g, \quad A?x_0; \quad Y!x_0;$$
$$[g = 0 \longrightarrow B?b; x_1 := f_1(x_0, b); C?c; x_2 := f_2(x_1, c)$$
$$\| g = 1 \longrightarrow B?b; W!b, \quad x_2 := x_0$$
$$\| g = 2 \longrightarrow x_2 := x_0$$
$$]; \quad Z!x_2$$
$$]$$

Variable x_2 is used outside of the selection statement and therefore must always be assigned a value. Intermediate DSA variable x_1 is only required in the first branch, however, and so is undefined in the other two branches. Now after applying projection in the normal fashion, the process implementing variable x_1 is

$$Px1 \equiv *[\quad Gx1?g, \quad X0x1?x_0;$$
$$[g = 0 \longrightarrow Bx1?b; \quad X1x2!f_1(x_0, b) \quad \| \quad \textbf{else} \longrightarrow \textbf{skip}] \quad]$$

We have eliminated communications on intermediate channel $X1x2$ when $g \neq 0$ with no overhead cost — even if $X1x2$ were to remain unconditional, guard variable g would be required in both processes $Px1$ and $Px2$.

The next task is to ensure that defined values of variables are sent only when they are actually used in the receiving module's computation. To illustrate, note that in $COND_{DSA}$ both x_1 and $W!$ depend upon b. However, x_1 is only assigned a value when $g = 0$ and $W!b$ is only executed when $g = 1$. Therefore, place the projection assignments for intermediate channels $Bx1$ and BW as follows:

$$*[\quad G?g, \quad A?x_0; \quad Y!x_0;$$
$$[\quad g = 0 \longrightarrow \quad B?b; \quad (Bx1!b \| Bx1?b_{x_1});$$
$$x_1 := f_1(x_0, b_{x_1}); \quad C?c; \quad x_2 := f_2(x_1, c)$$
$$\| \quad g = 1 \longrightarrow \quad B?b; (BW!b \| BW?b_W); W!b_W, x_2 := x_0$$
$$\| \quad g = 2 \longrightarrow \quad x_2 := x_0$$
$$]; \quad Z!x_2$$
$$]$$

After projection, the process implementing assignments to variable b is

$$Pb \equiv *[\quad Gb?g;$$
$$[\quad g = 0 \longrightarrow \quad B?b; \quad Bx1!b$$
$$\| \quad g = 1 \longrightarrow \quad B?b; \quad BW!b$$
$$\| \quad g = 2 \longrightarrow \quad \textbf{skip}$$
$$] \quad]$$

4.2 Encoding Guards

The last technique encodes guard conditions (branch conditions of selection statements) in fewer variables. The purpose of the transformation is to reduce the number and size of physical channels required in the decomposed system, given that every variable assigned a value within a selection statement depends upon the variables in guard conditions. For example, consider the following process.

$$ENCex \equiv *[\quad G_0?g_0, G_1?g_1, G_2?g_2, G_3?g_3; \quad A?a, B?b, C?c;$$
$$[\quad f(g_0, g_1, g_2, g_3) \longrightarrow \quad X!(a \wedge b), Y!(b \wedge c), z := a \wedge c$$
$$\| \neg f(g_0, g_1, g_2, g_3) \longrightarrow \quad z := b \vee c$$
$$]; \quad Z!(a \vee z)$$
$$]$$

Would there be more or fewer channels in the decomposed system if the guard conditions were encoded as follows?

$$*[\quad G_0?g_0, G_1?g_1, G_2?g_2, G_3?g_3; \quad A?a, B?b, C?c;$$
$$h := f(g_0, g_1, g_2, g_3);$$
$$[\quad h \longrightarrow \quad X!(a \wedge b), \quad Y!(b \wedge c), \quad z := a \wedge c$$
$$\| \neg h \longrightarrow \quad z := b \vee c$$
$$]; \quad Z!(a \vee z)$$
$$]$$

The answer depends on the size of the variables g_i, the number of variables assigned a value in the selection (three), and the number of guarded commands in the selection statement (two).

To begin, encode guards by assigning a communications cost to every variable in the sequential code. A variable that can hold K different values can be communicated on a 1of$K$ internal channel. (A 1of$K$ code comprises K data wires encoded in a one-hot style, and a single acknowledge wire.) For practical purposes, break large channels up into a group of channels of manageable size (e.g., 1-byte variables are not communicated on a 1of256 channel but rather upon four 1of4 channels).

Choose some base channel-size 1of$B$. Normally, $B = 4$ but any reasonable value (say, $B \leq 8$) can be chosen for this purpose. If a variable x can assume K different values, then define $V(x) = K$. The internal channel required to communicate x can be implemented as $\lceil \log_B K \rceil$ different 1of$B$ channels. This variable is therefore assigned a communications cost of $C(x) = \lceil \log_B(V(x)) \rceil$.

In summary, scanning through the sequential program then, for every selection statement: let G be the set of all guard variables in the selection; let N be the number of conditions in the selection; let A be the number of variables assigned a value within the selection. Let h be the variable that encodes the guard conditions. Now, compute E (the communications cost when guard conditions are encoded in h), and U (the cost when they are left unencoded). Then:

$$C(G) = \sum_{\forall g_i \in G} C(g_i) \qquad V(h) = N$$
$$E = C(G) + C(h) * A \qquad U = C(G) * A$$

If $E < U$ then encode the guard conditions of the selection in question. If not, leave the selection unencoded. The systems in figure 1 demonstrate the possible communications savings when guards using g_i are encoded in h using the technique described here.

Returning to our example, $N = 2$ and $A = 3$. Let $V(g_i) = 4$ for $\forall g_i \in G$. Then $C(G) = 4$, $C(h) = 1$, $U = 12$, and $E = 7$. In this case, encoding the guard conditions reduces the communications cost of the selection by almost half. In contrast, when $V(a) = V(b) = 4$, the process $*[A?a, B?b; [a \wedge b \rightarrow x\uparrow[]\text{else} \rightarrow x\downarrow]]$ is an example of a selection for which it is better not to encode the guard conditions ($U = 2$, $E = 3$).

4.3 Summary

The method for process decomposition is: convert the sequential CHP into DSA form; where possible, flatten nested selection statements and then encode guard conditions when energy consumption can be decreased; create and place projection assignments, including modifications for conditional communications; analyze data dependencies to create projection sets; perform projection. Each of these steps can be performed by an automated tool, and early prototypes (that take CHP programs as input) exist.

5. EXPERIMENTAL RESULTS

We synthesized the instruction fetch unit of an asynchronous 8051 microcontroller [9] using DDD as outlined in the previous section. The original CHP program describing this unit is given in figure 4 and was designed to optimize instruction throughput. It includes decoding (of variable length instructions), interrupt handling, and a 16-bit incrementer. The unit control is complicated by the fact that although instructions can be one to three bytes in length, they are always fetched from memory two bytes at a time. The unit is the limiting factor on the instruction throughput of the entire microcontroller, and in a custom design consumes roughly 12% of the energy of the microcontroller core.

This section compares the results of DDD with a decomposition performed by hand on the same program. The hand decomposition (which took one month for designers to finalize) is the one being sent for fabrication, and so every effort was made to optimize it for both speed and energy. We ran digital simulations in a 0.18 μm 1.8 V technology using the `esim` tool [13]. (Traditional methods of low-level synthesis were applied to transform the DDD modules into transistor netlists for simulation.) Historically, `esim`'s estimates are within 10% of the performance of the fabricated chip. The simulation results are summarized in figure 3.

In digital simulations, the system generated by DDD has roughly half the instruction throughput (102 MIPS) of the system designed by hand. With many more modules, the DDD system consumes roughly 2.74 times more energy per instruction (178pJ/instr). The DDD results are promising, since extended DDD is only the first half of high-level synthesis, with further optimizations to be introduced by clustering modules (Section 6).

For the purposes of comparison, consider the Philips asynchronous 80C51 microcontroller that was compiled in a syntax-directed manner. When performance figures are adjusted to the 0.18 μm technology, the Philips 80C51 runs at 11 MIPS [10]. We were not able to compare our energy figures to those of the Philips instruction fetch unit; in any case, the original specifications of the units are different.

6. IMPROVEMENTS BY CLUSTERING

We envision DDD as the first half of the synthesis method. The second half is clustering, which recomposes DDD modules into larger modules to further improve energy and performance. DDD modules can be clustered together with different goals in mind: reducing energy consumption, reducing forward latency, or reducing the time- and energy-efficiency metric Et^2 [11].

Clustering is implemented in two stages. In the first stage, DDD modules along the critical path of a system are repeatedly clustered in series until they are too large to be implemented as single PCHB stages. The second stage requires a global optimization algorithm (we are considering quadratic programming, simulated annealing, and genetic algorithms) to both cluster DDD modules in parallel and add slack-matching buffers to improve performance.

6.1 How Much Clustering Is Too Much?

The modules produced by DDD are often smaller than necessary, creating concurrent systems with extra communications overhead. To reduce overhead, therefore cluster modules together into larger CHP processes that still fit into a single PCHB asynchronous pipeline stage. Of course, an entire microprocessor can theoretically be implemented in a single pipeline stage by creating one large state machine. But limits both physical (e.g., the number of transistors in series) and self-imposed (e.g., cycle time) enforce a maximum size for pipeline stages.

It can be determined from the CHP alone whether a clustered module exceeds the maximum limits using the template for PCHB circuits. To check whether there will be too many transistors in series in the pulldown network, we can scruti-

Method	No.Modules	Instr.Throughput	Energy/Instr.	CycleSpeed	Energy/Cycle
Custom	23	208 MIPS	65 pJ	300 MHz	54.6 pJ
DDD	42	102 MIPS	178 pJ	330 MHz	59.0 pJ

Figure 3: Results of digital simulations (0.18μm, 1.8V). Ported to the same technology, the Philips asynchronous 8051 microcontroller runs at 11 MIPS. DDD performance can be improved by clustering (section 6).

```
FETCH ≡
    pc := initpc,  IMemPC!initpc,  OP!r,  C0!I0;
    *[ I?i;  iLen := i.len;  DIL!iLen,
        [ i.newpc ⟶ newpc := getpc ▯ i.movCR ⟶ newpc := read ▯ i.movCW ⟶ newpc := write ▯ else ⟶ newpc := false ];
        [ pc[0] = 0 ⟶
            [ iLen = 1  ∧  newpc = false   ⟶ pc := pc + 1
            ▯ iLen = 1  ∧  newpc = getpc ⟶ pc := pc + 1,  C1!X;  PtrPC!pc
            ▯ iLen = 2  ∧  newpc = false   ⟶ pc := pc + 2,  C1!I1
            ▯ iLen = 2  ∧  newpc = getpc ⟶ pc := pc + 2,  C1!I1;  PtrPC!pc
            ▯ iLen = 3  ∧  newpc = false   ⟶ pc := pc + 2,  C1!I1;  IMemPC!pc,  OP!r;  pc := pc + 1,  C0!I2
            ▯ iLen = 3  ∧  newpc = getpc ⟶ pc := pc + 2,  C1!I1;  IMemPC!pc,  OP!r;  pc := pc + 1,  C0!I2,  C1!X;  PtrPC!pc
            ▯                newpc = read   ⟶ pc := pc + 1,  C1!X;  PtrPC!pc
            ▯                newpc = write ⟶ pc := pc + 1,  C1!X
            ]
        ▯ pc[0] = 1 ⟶  pc := pc + 1;
            [ iLen = 1  ∧  newpc = false   ⟶   skip
            ▯ iLen = 1  ∧  newpc = getpc ⟶   PtrPC!pc
            ▯ iLen = 2  ∧  newpc = false   ⟶   IMemPC!pc,  OP!r;  pc := pc + 1,  C0!I1
            ▯ iLen = 2  ∧  newpc = getpc ⟶   IMemPC!pc,  OP!r;  pc := pc + 1,  C0!I1,  C1!X;  PtrPC!pc
            ▯ iLen = 3  ∧  newpc = false   ⟶   IMemPC!pc,  OP!r;  pc := pc + 2,  C0!I1,  C1!I2
            ▯ iLen = 3  ∧  newpc = getpc ⟶   IMemPC!pc,  OP!r;  pc := pc + 2,  C0!I1,  C1!I2;  PtrPC!pc
            ▯                newpc = read   ⟶   PtrPC!pc
            ▯                newpc = write ⟶   skip
            ]
        ];
        IG?ig;  DIG!ig,  [ ig.active ⟶ IOK?irpt ▯ else ⟶ irpt := none ];
        [ newpc = false ⟶ [ irpt ≠ none  ∧  pc[0] = 0 ⟶ newpc := getpc,  PtrPC!pc
                          ▯ irpt ≠ none  ∧  pc[0] = 1 ⟶ newpc := getpc,  PtrPC!pc,  C1!X
                          ▯ irpt = none ⟶ skip ]
        ▯ newpc = getpc ⟶ [ irpt ≠ none ⟶ Addr?pc;  PtrPC!pc ▯ irpt = none ⟶ skip ]
        ▯ newpc = read   ⟶ Addr?a;  IMemPC!a,  OP!r,  [ a[0] = 0 ⟶ C0!A,  C1!X ▯ a[0] = 1 ⟶ C0!X,  C1!A ];
                          [ irpt ≠ none ⟶ newpc := getpc;  PtrPC!pc ▯ irpt = none ⟶ skip ]
        ▯ newpc = write ⟶ Addr?a;  IMemPC!a,  OP!w;  [ irpt ≠ none ⟶ newpc := getpc;  PtrPC!pc ▯ irpt = none ⟶ skip ]
        ];
        [ newpc = getpc ⟶ Addr?pc ▯ newpc ≠ getpc  ⟶ skip ];  pcz := (pc[0] = 1);
        [¬pcz ⟶ IMemPC!pc, OP!r, C0!I0 ▯ pcz ∧ newpc ≠ false ⟶ IMemPC!pc, OP!r, C0!X, C1!I0 ▯ pcz ∧ newpc = false ⟶ C1!I0]
    ]
```

Figure 4: Sequential CHP for the fetch unit of the asynchronous 8051 microcontroller.
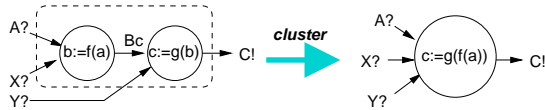

Figure 5: Clustering DDD modules in sequence.

nize modules with complicated boolean functions. To check if the cluster exceeds the specified cycle time, we examine modules with many wide communication channels. (Such modules can result in circuits with deep validity-check and completion trees.)

6.2   Clustering DDD Modules in Sequence

If two DDD modules appear in sequence, clustering them together may decrease the forward latency of the concurrent system. We must first check that their combined computation is simple enough that it fits into a single stage (i.e., count the transistors in series). Clustering two modules in sequence also reduces overall energy consumption by eliminating the channels (and validity checks) in between them.

The asynchronous systems we design achieve peak performance when all of the inputs to a module arrive at the same time. This is the motivating principle of "slack-matching," and it dictates that all paths from any primary input (external input channel in the sequential code) to a given module traverse the same number of pipeline stages. Slack-matching therefore helps determine the optimal number of pipeline stages along a path. Previous research shows that the Et^2 efficiency of a system is highest when there is an equal amount of power distributed to each pipeline stage [11]. Combining this requirement with the results of slack-matching, we see that a path that optimally has N pipeline stages is more efficient when all N stages perform some computation than when only two stages perform computation and the other $N-2$ are simple buffers there simply to add slack.

We therefore cluster modules together in sequence only if they appear upon the longest path of the decomposed system (i.e., if clustering reduces the forward latency of the system as a whole). There is no point in clustering modules in sequence along shorter paths, since buffers will be added during slack-matching to improve performance anyways. When the forward latency of the system has been decreased, we begin a new iteration and attempt to cluster modules along the new longest path. Only when no new clusters will fit in a single pipeline stage do we slack-match the system.
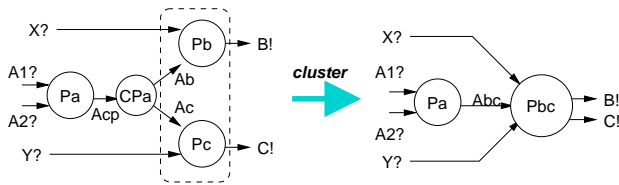
Figure 6: Clustering DDD modules in parallel.

Figure 7: A slack-matched concurrent system viewed as a table of columns.

Slack-matching is similar to the retiming problem in synchronous circuits [14], but in our asynchronous design style, every module combines computation logic and latching instead of keeping them as separate units. Also, slack-elasticity allows us to add buffers (the equivalent of retiming registers) on any channel without worrying about adding an equal number to each input of a combinational logic block to preserve system correctness. Finally, since we are slack-matching for Et^2 efficiency and time is not our only criterion, we must be careful not to over-slack-match systems.

6.3 Clustering DDD Modules in Parallel

Clustering modules that operate in parallel reduces energy consumption only when the modules share inputs (when both their computations depend on the same variable). When such modules are merged, their separate input communication channels are collapsed into one. This reduces the number of wires switched per cycle and eliminates the redundant input validity trees that used to grace each module. The two modules Pb and Pc in Figure 6 illustrate such a scenario.

Slack-matching also plays a role here. When we combine the two modules in parallel, we still want all inputs to the new module to arrive at the same time. We therefore can think of the system as a table with columns, where all modules in the same column are the same number of pipeline stages away from the primary inputs. For optimal performance, we only cluster modules that share inputs when they appear in the same column.

On shorter paths that use simple buffers to match the lengths of longer paths, we consider swapping the columns of these buffers with computation modules so that more computation modules can be clustered together. This is illustrated in figure 7), where external channels $A?$, $B?$ and $C?$ are primary inputs while $X!$ is a primary output. Modules $P6$ and $P7$ are in the same column and share inputs from $CP4$: as such, they can be clustered. Modules $P2$ and $P3$ both share inputs from $CP1$ but are in different columns. If $P3$ were swapped with the buffer before it, then it could be clustered with $P2$. But if the output channel of $P3$ is much larger than its input channel, the swapping and clustering can increase the overall energy consumption in the system.

It can be seen that the combination of clustering DDD modules and slack-matching for performance is a global optimization problem. Slack-matching systems that contain cycles is more complicated, and is discussed elsewhere [9].

7. CONCLUSION

We have presented Data-Driven Decomposition (DDD), the first half of a method for the high-level synthesis of energy-efficient asynchronous circuits. DDD employs data-dependency analysis to decompose a sequential program describing the circuit into a concurrent system of communicating modules. This is the first high-level synthesis method to target circuits in the
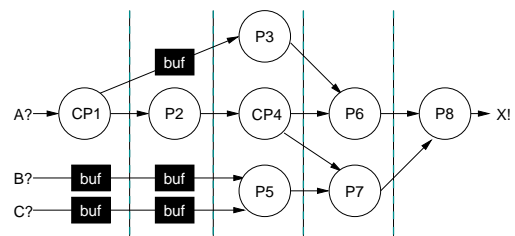
PCHB family of fine-grain asynchronous pipeline stages. Circuits from this family were used to build the asynchronous MIPS R3000 that, when performance figures are ported to the same technologies, is (to our knowledge) the fastest asynchronous microprocessor to date.

We also introduced the second half of our envisioned synthesis method: clustering DDD modules for further improvements in energy consumption, performance, or both. We set up the framework for clustering algorithms that target the same asynchronous PCHB circuits as DDD. Early prototypes of tools that automatically perform DDD exist, and we are also developing tools that automatically perform clustering.

8. REFERENCES
[1] A.J. Martin. "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits," in C.A.R. Hoare, ed., *Developments in Concurrency and Communication, UT Year of Programming Series*, Addison-Wesley, 1990.

[2] C.G. Wong and A.J.Martin. "Data-driven Process Decomposition For the Synthesis of Asynchronous Circuits," *Proc. ICECS*, 2001.

[3] S.M. Burns and A.J. Martin. "Synthesis of Self-Timed Circuits by Program Transformation," In G.J. Milne, ed., *The Fusion of Hardware Design and Verification*, North-Holland, 1988.

[4] A. Bardsley and D.A. Edwards. "The Balsa Asynchronous Circuit Synthesis System," *Forum on Design Languages*, 2000.

[5] C.H. van Berkel and R.W.J.J. Saeijs. "Compilation of Communicating Processes Into Delay-Insensitive Circuits," *Proc. ICCD*, pp. 157-162, 1988.

[6] J. Cortadella, M. Kishinevsky et al. "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. Information and Systems*, Vol. E80-D, No. 3, pp. 315-325, March 1997.

[7] R.M. Fuhrer, S.M. Nowick et al. "MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines," Columbia University CS Tech Report CUCS-020-99, 1999.

[8] A.J. Martin, A. Lines et al. "The Design of an Asynchronous MIPS R3000," *Proc. ARVLSI*, pp. 164-181, September 1997.

[9] A.J. Martin, M.Nyström et al. "The Lutonium: A Sub-Nanojoule Asynchronous 8051 Microcontroller," to appear in *Proc. ASYNC*, May 2003.

[10] H. vanGageldonk, K. van Berkel, and A. Peeters. "An Asynchronous Low-Power 80C51 Microcontroller," *Proc. ASYNC*, April 1998.

[11] A.J. Martin, M.Nyström, and P.Penzes. "ET2: A Metric For Time and Energy Efficiency of Computation," In R.Melhem and R.Graybill ed., *Power-Aware Computing*, Kluwer, 2001.

[12] R. Manohar and A.J. Martin. "Slack Elasticity in Concurrent Computing," *Proc. 4th Intl Conf. on the Math of Program Construction, Lecture Notes in CS*, Springer-Verlag, 1998.

[13] P. Penzes and A.J. Martin. "An Energy Estimation Method for Asynchronous Circuits with Application to an Asynchronous Microprocessor," *Proc. DATE*, 2002.

[14] C. Leiserson, F. Rose, and J. Saxe. "Optimizing Synchronous Circuitry by Retiming," *3rd Caltech Conference on VLSI*, 1993.