Peephole Optimization of Asynchronous Networks Through Process Composition and Burst-mode Machine Generation [1]

Ganesh Gopalakrishnan
Prabhakar Kudva

UUCS-93-020

Department of Computer Science
University of Utah
Salt Lake City, UT 84112, USA

September 22, 1993

Abstract

In this paper, we discuss the problem of improving the efficiency of macromodule networks generated through asynchronous high level synthesis. We *compose* the behaviors of the modules in the sub-network being optimized using Dill's trace-theoretic operators to get a single behavioral description for the whole sub-network. From the composite trace structures so obtained, we obtain interface state graphs (ISG) (as described by Sutherland, Sproull, and Molnar), encode the ISGs to obtain encoded ISGs (EISGs), and then apply a procedure we have developed called Burst-mode machine reduction (BM-reduction) to obtain burst-mode machines from EISGs. We then synthesize burst-mode machine circuits (currently) using the tool of Ken Yun (Stanford). We can report significant area- and time-improvements on a number of examples, as a result of our optimization method.

[1]This work was supported in part by NSF Award MIP-9215878

Peephole Optimization of Asynchronous Networks through Process Composition and Burst-mode Machine Generation

GANESH GOPALAKRISHNAN* (*ganesh@cs.utah.edu*)
PRABHAKAR KUDVA (*pkudva@cs.utah.edu*)
Department of Computer Science, University of Utah, Salt Lake City, Utah 84112, USA

1 Introduction

In the last decade, asynchronous[2]circuit and system design has seen significant growth in the number of practitioners and a corresponding broadening of the basic understanding at both the practical and theoretical levels [1, 2, 3] [4, 5, 6]. The result is that there are numerous design styles, many of which are supported by reasonable synthesis and analysis tools. Asynchronous systems have been shown to exhibit a number of inherent advantages: more robust behavior (in terms of process and environmental variations) [7], a capability for higher performance operation [8], decreased power consumption [9, 10, 11] which makes asynchronous circuits attractive for portable applications, inherently higher reliability in high speed applications [12, 13], and the ease with which coordination and communication between systems (that involve arbitration, message routing, *etc.*) can be supported. These developments warrant serious consideration of asynchronous circuits by the high level synthesis community which has, hitherto, largely focussed their efforts on synchronous circuits [14, 15, 16, 17].

Due to the considerable degree of attention that synchronous circuits have received, they would, in many cases, prove to be superior to asynchronous circuits if evaluated using many practical criteria. However, asynchronous system design has been making significant strides forward in recent times. In addition, considerable potential for improvement still remains untapped, and also enormous potential exists for *mixing* the synchronous and the asynchronous styles to derive the best advantages of both the styles [18, 19, 20, 21, 22, 23, 24].

In this paper, we shall discuss the problem of improving the efficiency of circuits generated through asynchronous high level synthesis using a tool such as SHILPA [25, 26, 27] or Brunvand's OCCAM compiler [28, 29]. These tools take a description in a concurrent process description language ("hopCP", or "Occam" in our examples) and translates it into a network of *macromodules* [30, 31]. Macromodules are hardware primitives that have an area complexity of anywhere from one to several tens of and-gate-equivalents, and are designed to support common control-flow constructs: the *procedure call* element (CALL), the *control-flow merge* element (MERGE), the *control-flow join* element (JOIN), the *sequencer* element (SEQUENCER), the *toggle* element (TOGGLE), various arbiters (ARBITER), modules that alter control-flow based on Boolean conditions (SELECT), and modules that help implement finite-state machines (such as DECISION-WAIT) are all examples of commonly used macromodules. The set of macromodules considered by us in this paper are largely those reported in [32, 33, 31], but include a few additional ones documented in [34]. One notable feature of these macromodules is that they employ transition-style (*non return-to-zero*) signaling, which is well known for its advantages, among which ease of understanding and reduced power consumption are prominent [31]. In the rest of the paper, whenever we discuss macromodules, we shall tacitly assume transition-style signaling.

For several reasons, macromodules are a popular choice as target for asynchronous circuit compilation. Their behavior is much easier to understand than that of corresponding Boolean gate networks. They have efficient circuit realizations in many technologies. However, when employed as the target of asynchronous circuit compilers, many instances of the same macromodule subnetwork tend to recur. For example, in

*Supported in part by NSF Award MIP 9215878

[2]Asynchronous circuits are those that do not employ a global clock, but instead, use handshake signals to achieve synchronization.

compiling process-oriented languages such as hopCP, the JOIN-CALL combination occurs frequently [3]. Retaining these sub-networks in the final circuit can lead to area- and time-inefficient circuits.

The problem we address in this paper is how to optimize these subnetworks in such a way that the optimized networks are valid replacements for the original networks in any behavioral context—*i.e.*, the optimizations are *context-free* in the behavioral sense. The particular approach we take in this paper to generate the logic for the optimized network, however, requires that the environment of the sub-network should allow the sub-network to operate in the *fundamental mode*, *i.e.*, allow sufficient time for the sub-network to stabilize after each excitation. Thus, the optimizations work in any behavioral context, subject only to simple timing assumptions. We call this problem the *peephole-optimization problem* for macromodule networks.

The peephole optimization problem for macromodule networks has been addressed in the past. In [35], this problem has been mentioned, though the details of the author's solution have not been published. In [36], we have addressed the problem of *verifying* peephole optimizations. The approach of [7] altogether avoids the peephole optimization problem by synthesizing logic equations directly from a textual intermediate form called *production rules*. The generality of Martin's logic synthesis method is not well understood. We believe that asynchronous high-level synthesis approaches that generate macromodule networks as intermediate form possess several advantages, among which the advantages of retargetability, understandability of the generated circuits, and ease of validation of the compiler are prominent. Therefore, we prefer the approach of generating macromodule networks and later optimizing them.

Our approach to peephole optimization is based on *process composition*. We *compose* the behaviors of the modules in the sub-network being optimized to get a single behavioral description for the whole sub-network[4]. We model behaviors in the *trace theory* of Dill [37], and use the composition operator on trace structures (Section 2.) Dill's parallel composition (or process composition) operator which enjoys a number of desirable properties, which we exploit (detailed in Section 2). The behavior *inferred* in this fashion is often quite succinct, as it leaves out many combinations of the behaviors of the submodules that can never arise, or can lead to *internal hazards* in the circuit. (Note: The parallel composition operator is usually used in conjunction with the *hide* operator that can also suppresses irrelevant information.) Taking the example of the JOIN-CALL combination, we will compose the automatons[5]describing the behaviors of a JOIN and a CALL element to get a single automaton that describes the behavior of the combination. To the best of our knowledge, the use of process composition as a step in the peephole optimization process is new.

Recall that the behavior of our macromodules is expressed in the transition signaling style. Before we synthesize logic corresponding to the automata inferred through process composition, we convert them into encoded interface state graphs (EISGs) [38]—automata that label their moves with polarized signal transitions (*e.g.*, "a rising" (a+), "b falling" (b-), *etc.*). (EISGs are very similar to the *state graphs* of Chu [39].)

Now we address the problem of logic synthesis for implementing the EISGs. Asynchronous logic synthesis is inherently harder than that for synchronous systems primarily due to the need to provide *hazard covers*. Synthesizing hazard-free implementations of asynchronous automata with unrestricted behaviors is still an open problem (and is of dubious value, as totally unrestricted behaviors are rare). Recent research in asynchronous system design has resulted in many synthesis procedures that impose mild restrictions on the class of asynchronous automaton descriptions allowed, and synthesize hazard-free efficient implementations for the members of these sub-classes. We choose one such class proposed by Davis [8] called *burst-mode machines*. Efficient tools for synthesizing burst-mode machines are becoming available [23, 40, 8]. We then discuss how EISGs are converted into Burst-mode machines through a process called *BM-reduction*, and synthesized using the tool of Yun [40].

In Section 3, our optimizer is illustrated on a simple example. We also present general details about parallel composition, EISG generation and BM-reduction.

[3] It results from multiple invocations of communication on a channel from different places in the program text.

[4] Analogous to composing two resistors in parallel to get an *equivalent resistance*, for instance. However automaton composition is more involved than resistance composition!

[5] We use the terms "asynchronous automata" and "asynchronous state machines" interchangeably.

Our peephole optimizer has been applied on a number of examples and we are encouraged by our results to date. We should note that our optimizer imposes some restrictions on the kind of macromodules it allows at its input. It allows only *deterministic* components (*e.g.*, no arbiters), because burst-mode machines are deterministic. Secondly, it handles only *delay insensitivity* [41] modules, because the BM-reduction process capitalizes on Udding's syntactic conditions about *delay insensitivity* [41]. Fortunately these restrictions are mild, and are obeyed by most of the sub-networks generated by SHILPA (or similar compilers). In Section 4, we discuss our experimental results and provide concluding remarks.

2 Basic Definitions and Concepts

In this section, we begin by stating basic definitions about asynchronous circuits, our assumptions, as well as key results needed here (detailed elsewhere, for example [37, 42, 43]).

2.1 Basics of Burst-mode Machines

Burst-mode machines are a *restricted* class of asynchronous finite-state machines developed at HP-laboratories [8]. They are capable of modeling a large variety of practical systems (control-oriented behaviors) succinctly, and have been used widely in significant projects [8, 44].

A burst-mode machine is a "Mealy-style" finite-state machine in which every transition is labeled by pairs (I, O) (written in the usual "I/O" notation). I is a non-empty set of polarized signal transitions (an example: {a+,b-}), and is called the *input burst*. According to the original definition of burst-mode machines [8], O is a possibly empty set of polarized signal transitions called the *output burst*. *We shall, however, assume that O is non-empty.* Doing so is consistent with the assumption of delay insensitivity that we make regarding our macromodules.

When in a certain state, a burst-mode machine awaits its input burst to be completed (all the polarized transitions to arrive), then generates its output burst, and proceeds to its target state. The environment can also be given a burst-mode specification by *mirroring* [37] a given burst-mode machine. The environment's specification will consist of I/O pairs where the Is are output-bursts for the environment and the Os are input-bursts for the environment. Thus, it can be seen that the environment must wait for the output burst to be complete before it can generate new inputs. We call this the *burst-mode assumption*.

Burst-mode machines obey a number of restrictions. For input bursts Is_1 and Is_2 labeling any two transitions leaving a state s, neither must be a subset of the other. This enforces determinacy. The set of input bursts need not be exhaustive. Those input bursts that are not explicitly specified are assumed to be illegal, and cause undefined behaviors when invoked.

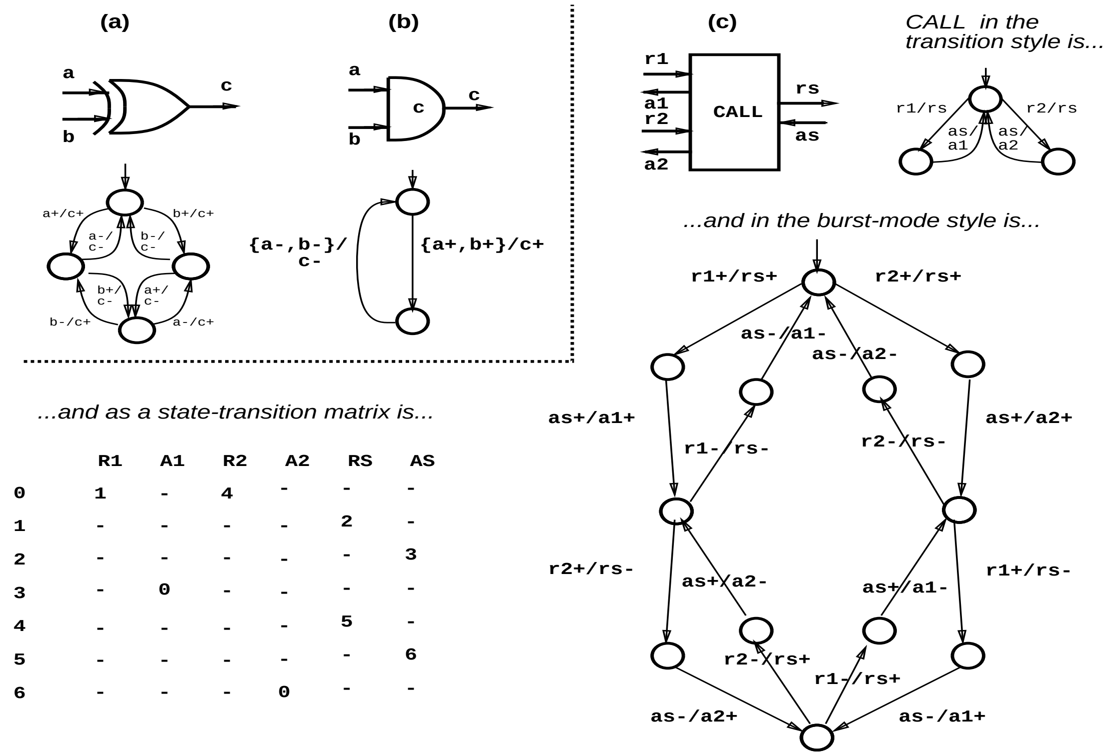
If a state of a burst-mode machine can be entered via two separate transitions, then the output bursts associated with these transitions must be compatible (for instance, one output-burst should not include b+ while the other output-burst includes b-).

Figures 1(a),(b), and (c) model a MERGE (an "XOR-gate"), a Müller C-ELEMENT (or a "join"), and a CALL element, respectively. These diagrams specify the *intended usages* of these modules. For completeness, we provide the behavior of the CALL element in the transition style as well as a state-transition matrix. As an example, the state-transition matrix is to be read as follows: in state 0, when R1 is obtained, go to state 1; in state 1, generate an RS and go to state 2; in state 2, when an AS is obtained, go to state 3; in state 3, generate an A1 and go back to state 0.

2.2 Assumptions about Macromodule Networks Being Optimized

The macromodule network being optimized by our optimizer must not contain *arbiters*, or *arbiter*-like components, because our optimizer generates burst-mode machines as output, and burst-mode machines cannot model arbiters. The only use of non-determinism permitted in the network input to the optimizer is to model concurrency through non-deterministic interleaving.

The network of macromodules being considered by the optimizer should initially be in a quiescent state[6].



	R1	A1	R2	A2	RS	AS
0	1	-	4	-	-	-
1	-	-	-	-	2	-
2	-	-	-	-	-	3
3	-	0	-	-	-	-
4	-	-	-	-	5	-
5	-	-	-	-	-	6
6	-	-	-	0	-	-

Figure 1: Burst-mode machines for MERGE(a), C-ELEMENT(b), and CALL(c)

This is because burst-mode machines must have *non-empty* input bursts, and therefore, cannot produce an output without receiving any inputs immediately after "power up".

The network of macromodules being considered by the optimizer should also never *diverge* (in other words, the network should be infinitely often in a quiescent state). This is because each "circle" (state) in a burst-mode machine models a quiescent situation, and every transition ("arc") must be incident on a "circle", in order to be a well-formed burst-mode machine graph.

We assume that our macromodules are *delay insensitive* (DI). The behavior of a DI module does not depend on its computational delays or delays on wires used to communicate with it. If the behavior is invariant over module-delays but not over wire delays, the module is classified as *speed independent* (SI).

2.3 Basics of Trace Theory

Trace theory[37] is a formalism for modeling, specifying, and verifying speed-independent and delay-insensitive circuits. In this paper, we employ trace-theory to provide us with semantically well specified operators (namely *parallel composition*, *renaming*, and *hiding*) that allow us to obtain the composite behavior of networks of delay-insensitive macromodules, in the process of optimizing them.

Trace-theory is based on the idea that the behavior of a circuit can be described by a regular set of *traces*, or sequences of transitions. Each trace corresponds to a partial history of signals that might be observed at the input and output terminals of a circuit.

A *simple prefix-closed trace structure*, written *SPCTS*, is a three tuple (I, O, S) where I is the *input alphabet* (the set of input terminal names), O is the output alphabet (the set of output terminal names), and S is a prefix-closed regular set of strings over $\alpha = I \cup O$ called the *success* set.

In the following discussion, we assume that S is a non-empty set. We associate a SPCTS with a module that we wish to describe. Roughly speaking, the success set of a module described through a SPCTS is the

[6] A system is in a quiescent state if it is blocked waiting for external inputs.

set of traces that can be observed when the circuit is "used properly". With each module, we also associate a *failure* set, F, which is a regular set of strings over α. The failure set of a module is the set of traces that correspond to "improper uses" of the module. A failure set of a module is completely determined by the success set: $F = (SI - S)\alpha^*$. Intuitively, $(SI - S)$ describes all strings of the form xa, where x is a success and a is an "illegal" input signal. Such strings are the minimal possible failures, called *chokes*. Once a choke occurs, failure cannot be prevented by future events; therefore F is suffix-closed. (Note: When we specify a SPCTS, we generally specify only its success set; its input and output alphabet are usually clear from the context, and hence are left out.)

As an example, consider the SPCTS associated with a unidirectional BUFFER with input a and output b. In this context we view a buffer as a component that accepts signal transitions at a and produces signal transitions at b after an unspecified delay. If we were to use BUFFER properly, its successful executions will include one where it has done nothing (*i.e.*, has produced trace ϵ), one where it has accepted an a but has not yet produced a b (*i.e.*, the trace a), one where it has accepted an a and produced a b (*i.e.*, the trace ab), and so on. More formally, the success set of BUFFER is

$$(\{a\}, \{b\}, \{\epsilon, a, ab, aba, \ldots\}).$$

This is a record of all the partial histories (including the empty one, ϵ), of successful executions of BUFFER. An example of an improper usage of BUFFER—a *choke*—is the trace "aa". Once input "a" has arrived, a second change in "a" is illegal since it may cause unpredictable output behavior. A buffer of this type can be used to model a wire with some delay.

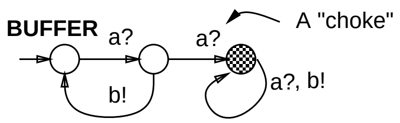


Figure 2: The Finite Automaton corresponding to BUFFER

We can denote the success set of a SPCTS by using a state diagram such as the one in in Figure 2. In this diagram, constructs such as $a?$ denote incoming transitions (rising or falling) and constructs such as $b!$ denote outgoing transitions (rising or falling). This diagram also shows the choke of BUFFER.

There are two fundamental operations on trace structures: *compose* ($\|$) finds the concurrent behavior of two circuits that have some of their terminals of opposite directions (the directions are input and output) connected, and *hide* makes some terminals unobservable (suppressing irrelevant details of the circuit's operation). A third operation, *rename*, allows the user to generate modules from templates by renaming terminals. For the purposes of this paper, the following property of these operators is important: parallel-composition, renaming, and hiding preserve delay insensitivity.

To transform a speed-independent circuit into a delay insensitive circuit (in the context of Dill's trace theory), buffers can be placed in series with the terminals of the speed-independent circuit. However, placing buffers in series with the terminals of a delay insensitive circuit has no effect whatsoever on the behavior of the circuit.

2.4 Udding's Conditions for Delay Insensitivity

Udding [41, 45] has done foundational work on delay insensitivity, and has provided four syntactic conditions that are necessary and sufficient to guarantee delay insensitivity. These conditions, which form the basis of BM-reduction, are now briefly explained (and illustrated in Figure 3(a) through (d)).

Udding's first condition is: "*the same signal cannot transition twice in a row.*" If a signal were to so transition, then the second transition, which may be applied too soon, can nullify the effect of the first transition applied on the same wire, giving rise to a "runt pulse". An example is provided in Figure 3(a).

Udding's second condition is "*if a module accepts (generates) two inputs a and b in the order ab, it must also accept (generate) them in the order ba.*" If this were not so, the wires leading to the module (which

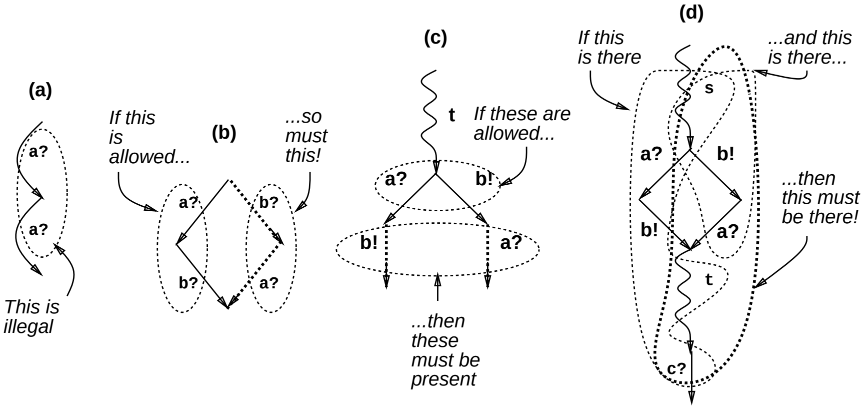


Figure 3: Udding's Conditions for Delay Insensitivity

can have arbitrary delays) can reorder the transitions and present them to the module in the wrong order. Notice that the words "accepts" and "generates" are used symmetrically above, because the environment can also be treated as a module, through the process of mirroring. An example is provided in Figure 3(b).

Udding's third condition is *"for input symbol a and output symbol b, and for arbitrary trace t, if the behaviors ta and tb are legal for the module, then the behaviors tab as well as tba must also be legal."* This is explained as follows. After processing t, the module has the choice of generating a b and awaiting an a, or vice-versa (and like-wise the environment). Suppose the module chooses to generate the output b. The environment has no immediate way of knowing that this choice was taken by the module (due to arbitrary wire delays). In fact, the environment may "think" that the module is waiting for an a (which is also legal for the module to do after a t). Therefore, the environment can go ahead and generate an a. The module will, therefore, end up seeing the sequence tba, which better be legal for the module. The argument can be completed using symmetry (via mirroring). An example is provided in Figure 3(c).

Udding's fourth condition is *"for symbols a, c which are both inputs and b which is an output, and arbitrary traces s and t, if sabtc as well as sbat are legal for the module, then sbatc should also be legal."* The argument goes as follows. The existence of $sabtc$ in the success set of the module says that after s, the *environment of the module* has a choice of causing its a output followed by awaiting its b input. The existence of $sbat$ in the success set of the module says that the module has the choice of causing its b output followed by awaiting its a input. Therefore the following scenario is possible: the environment and the mechanism together engage in trace s. Then, concurrently, the environment emits a while the module emits b as the next signal transition. Depending on the wire delays, a could arrive at the module before the module emits b, or could arrive later. Suppose a did not arrive at the module in a "timely way". Also suppose that b did not arrive at the environment in a "timely way". The module "thinks that" it is engaged in behavior sba and pursues it, while the environment "thinks that" it is engaged in sab and pursues it. The module and the environment then continue with behavior t which is a legal extension to both sab and sba. Now the environment can emit a c. The module, after processing the trace $sbat$, must find c to be a legal input.

In effect, after s, the module and the environment end up "talking at the same time" and hence lose track of whose action came first. An example is provided in Figure 3(d).

3 Details of the Optimizer With Examples

3.1 A Simple Example "Control-Block Sharing"

The input to our synthesis procedure is a macromodule network. An example appears in Figure 4. The MERGE component always receives a `START` transition at power-up. This places a "call" on the CALL element through the `R2` input. This enables the `A` input of the C-ELEMENT. When a transition on `A_IN` arrives, `A_OUT` is produced. `A_OUT` also "returns" the call to the CALL element, which re-enables a call, now through the `R1`

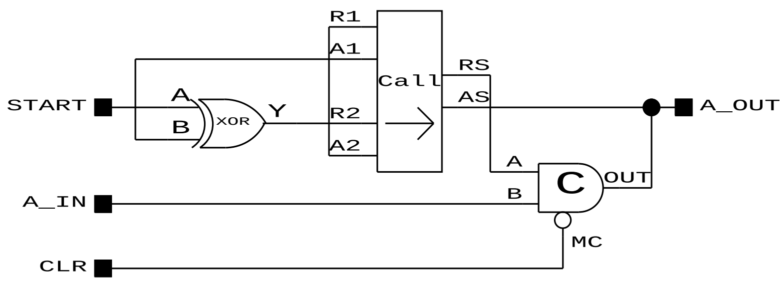


Figure 4: Circuit Illustrating Control Block Sharing

input. Thus, this circuit should reduce to a wire between A_IN and A_OUT.

Feeding this circuit through Dill's parallel composition operator gives us the following state transition matrix:

```
      A_OUT  A_IN
  0:      -     1
  1:      2     -
  2:      -     3
  3:      4     -
  4:      -     5
  5:      6     -
  6:      -     7
  7:      8     -
  8:      -     9
  9:      2     -
```

These signal transitions are converted into an EISG, and into a burst-mode machine, which indeed generates a wire!

```
0    2        A_IN+ |      A_OUT+
6    0        A_IN- |      A_OUT-
4    6        A_IN+ |      A_OUT+
2    4        A_IN- |      A_OUT-
```

3.2 Detailed Explanation of the Optimizer

We now go through the details of the optimizer, giving all the steps from inputting a macromodule circuit upto obtaining and verifying a burst-mode machine. Each step is explained in a separate section.

3.2.1 Obtaining a Macromodule Network

As explained before, the input macromodule network is obtained from any asynchronous high-level synthesis tool that meets our criteria. For specific discussions, we will use SHILPA as our example high-level synthesis tool.

3.2.2 Identifying a Sub-network to Optimize

Currently we identify the sub-network manually, although, in future implementations, we could obtain the sub-network as a result of performance studies. We could also determine burst-mode circuits corresponding to standard "macromodule network idioms" once and for all, and store them. Also, since Dill's parallel composition operator is, essentially, exponential in its execution time, it often pays to compose a large network by composing its sub-sub-networks first, and then composing the results. Determining a suitable set of partitions (and sub-partitions) is important for the overall efficiency. Currently we do this by hand, by picking clusters of most closely related components.

Once the sub-network to optimize is identified, it is to be guaranteed that its environment will obey the burst-mode assumption with respect to it. That is, the environment must, after supplying every input burst, wait for the sub-network to produce its output burst. This calls for path-delay analysis which is currently done using conventional simulation tools.

3.2.3 Imposing Environmental Constraints

Once a sub-network is identified, the environment of the sub-network must be suitably specified, to avoid obtaining too general a result. For example, each sub-network can interact with its environment through either an *active* channel or a *passive* channel. An active channel involves the output of a *request* transition followed by the receipt of an *acknowledge* transition. A passive channel awaits a *request* transition and then generates an *acknowledge* transition. Channel connections to the environment must not be left "dangling", for, this would cause impossible behaviors to be considered by the parallel composition process. For example, for an active channel, we must stipulate that *acknowledge* will come only after a *request*. If this is not specified, the parallel composition operator will allow for the possibility of an *acknowledge* even before a *request*. These constraints are expressed by introducing fictitious modules that possess the required I/O traces and effectively "close off" the dangling channel connections properly.

SHILPA generates connections to datapath elements by treating them as active channels (*i.e.*, it places a request for a computation on the datapath element and awaits an acknowledge). Datapath elements are not considered in their entirety by our optimizer; only their *control-aspects* are considered. Therefore the abstraction for a datapath element, as far as our optimizer is concerned, is an active channel. Therefore, connections to datapath elements are modeled exactly as "dangling" active channels are modeled.

One more preprocessing step is to be applied: any MERGE element with a `START` input (as in Figure 4) is replaced by an IWIRE—initialized wire—that acts like a buffer with input a and output b except that it begins operation by generating a b. Notice that IWIRE is non-quiescent. However, it can be guaranteed that SHILPA-generated circuits that involve the IWIRE are quiescent. Once the above steps are completed, parallel composition can be invoked on the network (usually recursively, on the partitions, as mentioned in Section 3.2.2).

3.2.4 Composite SPCTS to EISG

Composite SPCTS are converted into EISGs by exhaustively "simulating" all possible moves until all reachable configurations are covered. In Figure 1, this process is illustrated for a CALL element. This process also can result in state explosion. For instance, behaviors in which many nested branches are involved can expand into very large EISGs. Fortunately, this has not proved to be a problem so far.

3.2.5 BM-reduction

Once EISGs are obtained, they are to be converted into equivalent burst-mode machines. This algorithm, and its correctness, are briefly outlined below.

Input: An EISG, which is a state graph with circles denoting states, and arcs between states labeled by a single polarized transition of an input signal or an output signal. Only those EISGs obtained by composing macromodules obeying restrictions stated earlier are considered.

Output: A burst-mode machine.

Method:

1. Mark all states as "not visited", and call the starting state *current*.
2. If the *current* state has not been visited, and has an exit through at least one output transition, mark *current* as visited, and retain any arbitrary output transition leaving that state. Eliminate all other transitions. Call the destination of the retained transition as *current*. Continue with Step 2.
3. We reach this step when the "*current*" state has no exits through an output transition. Retain all the exits through input transitions from this state. Consider all the destination states reached through

these input transitions as "*current*", in turn, and continue with Step 2 for these states.

4. We reach here after the initial "transition elimination" portion of the algorithm is over. Now, remove unreachable portions of the state graph.
5. Set the starting state of the state graph as "*current*".
6. Go to the *current* state. It will have exits only through input transitions. (This invariant is initially true due to the quiescence of the starting state, and is preserved by the way the following loop will work.)

   Take any path out of the "*current*" state and traverse the path, collecting input transitions encountered along the way into a set "*input-burst*". (We will never encounter a state in the interim that has both an input exit as well as an output exit because the loop starting in Step 2 would then have eliminated all the input exits, and all but one output exit.) Continue collecting input transitions, till we encounter a state with exactly one output exit. Call this state "*intermediate*".
7. Continue traversing from state "*intermediate*" collecting output transitions into a set "*output-burst*" till a state which has no exits through an output transition is encountered. Call this state "*next*".
8. Construct a burst-mode machine arc from "*current*" going to "*next*" labeled by "*input-burst/output-burst*".
9. Repeat the procedure from Step 6 for all paths emanating from "*current*".
10. Recurse, now treating all the states marked "*next*" as "*current*", and till all states have been considered.
11. Eliminate all duplicate transitions in the burst mode machine.

We illustrate the above algorithm on the following state transition matrix (corresponding to "QR42": a four-phase to two-phase converter with *quick return linkage* [45]). In this example, we do not consider polarized transitions, to avoid notational clutter; the same method applies, whether the transitions are polarized or not. (The various kinds of parenthesizations, `[]`, `<>`, etc. are explained momentarily.)

```
Input wires:  R4 A2
Output wires:  R2 A4
Start state number: 0
        R4    R2    A2    A4
  0:    [1]    -     -     -
  1:     -    [8]    -     2
 <2:>    3     6     -     -
 <3:>    -     4     -     -
  4:     -     -    [5]    -
  5:     -     -     -    [0]
  6:    [4]    -    [7]    -
  7:    [5]    -     -     -
  8:     -     -     9    [6]
 <9:>    -     -     -     7
```

The transitions enclosed in `[]` are the ones selected in the loop starting at Step 2. The states enclosed in `<>` are the ones referred to in Step 4 as being the "unreachable portions of the state graph". In state `1`, we could have retained the transition going to either state `8` or state `2` because both transitions are through outputs, namely `R2` and `A4`; we arbitrarily choose the transition going to state `8`. In state `8`, we *must* retain the transition going to state `6` because this move is through the output transition `A4`; the transition going to state `9` is not retained, as that transition is through the input `A2`. In state `6`, both transitions must be retained because both of them are input transitions.

After Steps 1 through 4, we are left with the following graph (we now remove the `[]` decoration):

```
Input wires:  R4 A2
Output wires:  R2 A4
Start state number: 0
        R4    R2    A2    A4
  0:     1     -     -     -
  1:     -     8     -     -
  4:     -     -     5     -
  5:     -     -     -     0
  6:     4     -     7     -
  7:     5     -     -     -
  8:     -     -     -     6
```

We now take state `0` as current, and traverse till state `1`, forming the set *input-burst* {`R4`}. State `1` is called **`intermediate`**. From state `1`, the traversal is continued, forming set *output-burst* {`R2,A4`}, reaching state **`next`**, which is state `6`. From state `6`, two arcs are erected to state `0`, both with *input-burst* {`R4,A2`} (paths `6,4,5,0` and `6,7,5,0`, with *current* being state `6`, *intermediate* being state `5` and *next* being state `0`); only one is retained. The following burst-mode machine is now constructed:

```
Input wires:  R4 A2
Output wires:  R2 A4
Start state number: 0

  0 -- R4/{R2,A4} --> 6

  6 -- {R4,A2}/A4 --> 0
```

The above burst-mode machine is realized through the following equations, generated by Yun's tool [40].

```
A4 =  R4 +  A2' R2 +  A2 R2'

R2 =  R4' R2 +  R4 QO' +  R2 QO'

QO =  A2 R4' +  A2 QO +  R4 QO
```

The steps in the algorithm can be justified as follows.

- When a state s has both out-going output- and input-transitions, due to adherence to Udding's conditions, these inputs and outputs will be offered in all combinations. The environment (due to the *burst-mode assumption*) will allows enough time for the machine to produce all these outputs in some order before it considers applying any of the inputs being offered. This justifies the step that retains an arbitrary output transition in preference to all other transitions, in Step 2.
- By the same token, when a sequence of inputs is being collected to form the set *input-burst*, we can be assured that these inputs will appear in all permutations (due to delay insensitivity). Furthermore, the behavior following this input burst (irrespective of the path taken to form it) will be the same. Thus, we can be assured that all input bursts will lead to the same output burst. This will give rise to opportunities to eliminate duplicate transitions.

3.2.6 Correctness of BM-reduction

The above reasoning shows that BM-reduction results in a burst-mode machine that has the same behavior as the original macromodule network when that network is operated under the burst-mode assumption. The following well-formedness conditions of burst-mode machines are also guaranteed:

Circuits	Burst-mode machine size	Macromodule machine size	Burst-mode machine speed	Macromodule machine speed
Merge	2a,1o	2a, 1o	an xor delay	an xor delay
QR42 (hand design)	8a, 5o	10a, 3o	10nS	15nS
QR42 (version 1)	8a, 5o	48a, 26o	10nS	20nS
QR42 (version 2)	8a, 5o	40a, 20o	10nS	15nS
Two-input Call	12a, 5o	14a, 7o	4nS	8nS
Control-Block Sharing	0a, 0o	20a, 11o	0nS	11nS
Call-C Idiom	19a, 8o	18a, 9o	10nS	11nS
Decision Wait	18a, 8o	12a, 6o	8nS	15nS
Simple GVT (part 1)	9a, 6o	12a, 6o	4nS	4nS
Simple GVT (part 2)	5a, 3o	8a, 4o	10nS	14nS
Call3-Merge Optimization	50a, 18o	30a, 15o	11nS	16nS

Figure 5: Performance of our Optimizer

Non-empty Input Bursts: The fact that all input bursts are non-empty is guaranteed by the *quiescence* requirement and by the way the BM-reduction process works.

Subset Property: The subset property requires that no input burst can be a subset of another. This is guaranteed as follows. Consider the traversal made by algorithm BM-reduction beginning at step 6, when it forms the set *input-burst*. Suppose the sequence $i_1, i_2, \ldots, i_n$ is encountered by the time state *intermediate* is reached. Due to Udding's condition 2 (Figure 3(b)), we are assured that this sequence will also appear in all its permutations between state *current* and *intermediate*. Furthermore, the sequence of outputs encountered during the traversal from *intermediate* to *next* will also appear in all its permutations. Thus, we will end up getting many transitions with identical *input-burst* and *output-burst* sets—specifically as many such burst-mode transitions as the product of the number of permutations that the inputs and the outputs have. Thus, we can get *duplicate* burst-mode transitions, but never two burst-mode transitions that violate the subset property. (Duplicate burst-mode transitions will be eliminated in Step 11.)

Unique Entry: This is guaranteed by the way an EISG is generated (essentially a *state* of an EISG includes the state of the interface signals; hence, there cannot be a state-conflict in the burst-mode machine because the EISG will allocate two separate states for non-compatible interface-signal assignments.

4 Results and Concluding Remarks

We have an implementation for all the phases of our optimizer described here, and these phases have been integrated to some extent. The parallel composition tool was developed by Dill and Nowick [42]. The

EISG generation algorithm is described in [38]. The BM-reduction algorithm has been implemented by us. For burst-mode machine generation, we use the tool developed by Yun [40]. Yun's tool also generates a Verilog description of the circuit; however, for comparing burst-mode machine outputs with macromodule networks (which we have in the Viewlogic tool database), we end up translating combinational logic equations describing burst-mode machines into VHDL (the version of Viewlogic that we use does not compile Verilog into circuits), and use the VHDLDesigner tool to generate schematics. Looking back, it is interesting to observe the great extent to which we end up using tools/techniques developed for *synchronous high level synthesis*!

Our results fall into different categories. In general, the burst-mode circuits generated by us are often smaller and faster, as shown in Table 5. Here is how area figures were obtained: we obtained gate-style implementations for the individual macromodules, and added their sizes to obtain the area of the macromodule circuit. The output of the burst-mode machine is in AND/OR form for which we obtain a gate-count straight-forwardly. In both cases, we reduce the whole description to two-input AND/OR gates and report their count in the form "#a, #o" for "number of ands and ors", respectively. In this table, the circuits *Call-C Idiom*, *Simple GVT* (part 1 and 2), and *Call3-Merge* are various networks produced by the Occam or SHILPA compilers; others have been mentioned earlier.

Obtaining speed estimates can be trickier. Here, we focus on throughput, and not on latency. For throughput, the notion of a *critical path* does not apply. Instead, the notion of *cycle time* [46] applies. Cycle time is the period of one cycle of the repetitious behavior exhibited by the asynchronous circuit when the circuit's environmental connections are suitably "closed off", to make the asynchronous circuit into an oscillator [7]. For each of our test circuit we close-off the environment so as to use the burst-mode circuit as aggressively as possible, short of breaking the burst-mode assumption. The same environment was then used for simulating the macromodule circuit.

Burst-mode circuits do not need special provisions for being reset; merely holding the interface signals low after power-up achieves resetting. In contrast, the macromodule circuits require an explicit reset signal. Automated techniques to avoid providing an explicit reset input to macromodules are not known to us (though briefly discussed in [3] by Furber).

We have experimentally determined that our optimizer subsumes virtually every *macromodule-network to macromodule-network* optimization proposed by Brunvand in [28]. Our optimizer, in effect, achieves macromodule-network optimization and burst-mode machine generation in one phase.

Burst-mode circuits are believed to be often easier to test than ordinary macromodule-based circuits [47], especially if implemented using Nowick's *locally clocked* style [23]. They can also be synthesized as complex gates, as done by Stevens [8].

In conclusion, replacing macromodule-networks by burst-mode machine networks often seems to have many advantages. In practice, one may carry out macromodule subnetwork replacement till the required degree of performance is achieved. Then, one may leave some macromodules (at the "top level") unaltered, for, they make the control organization of a large system quite clear. In the process of replacing macromodule sub-network after sub-network, incremental algorithms for repeatedly re-verifying the burst-mode assumption seem highly desirable.

Acknowledgements. The authors are very grateful to Nick Michell for providing assistance in every phase of our work. Thanks to Ken Yun for timely help with his code. Thanks also to Erik Brunvand and Steve Nowick for numerous discussions and tool/library support.

References

1. *Workshop on "Asynchronous and Self-Timed Systems," University of Manchester, March 1993. (Organizer: Steve Furber).*

2. *Articles in the Minitrack "Asynchronous and Self-Timed Circuits and Systems," of the Computer Ar-*

[7]Our simulator always tried to achieve "stabilization of the network within a clock cycle" and after so many "ticks" quit, saying that the network is unstable; but by then we had finished our observation of the cycle-time!!

chitecture Track of the 26th Hawaiian International Conference on System Sciences, January, 1993. (Minitrack Organizers: Ganesh Gopalakrishnan and Erik Brunvand.).

3. *VII Banff Workshop on "Asynchronous Hardware Design", Banff, Canada, August 28-September 3, 1993. (Organizer: Graham Birtwistle).*

4. *Special Issue on Asynchronous and Self-Timed System Design, edited by Ganesh Gopalakrishnan and Erik Brunvand for the journal "Integration", planned for the end of 1993.*

5. *Special Issue on Asynchronous and Self-Timed System Design, edited by Tam-Anh Chu and Rob Roy for the journal "IEEE Design & Test of Computers", planned for 1994.*

6. *Special Issue on Asynchronous and Self-Timed System Design, edited by Teresa Meng and Sharad Malik for the journal "Journal of VLSI and Signal Processing", Volume 6, Issue 3, November 1993.*

7. Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In editor C.A.R. Hoare, editor, UT Year of Programming Institute on Concurrent Programming. Addison-Wesley, 1989.

8. Al Davis, Bill Coates, and Ken Stevens. The Post Office Experience: Designing a Large Asynchronous Chip. In T.N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1*, pages 409–418, January 1993.

9. Jaco Haans, Kees van Berkel, Ad Peeters, and Frits Schalij. Asynchronous multipliers as combinational handshake circuits. In *Proceedings of the IFIP Working Conference on Asynchronous Design Methods, Manchester, England, 31 March - 2 April, 1993*, 1993. *Participant's edition.*

10. Kees van Berkel. *Handshake Circuits: An Asynchronous Architecture for VLSI Programming.* Cambridge University Press, 1993.

11. C. H. van Berkel, C. Niessen, M. Rem, and R.W. J. J. Saeijs. VLSI programming and silicon compilation: a novel approach from Philips Research. In *Proc ICCD*, New York, 1988.

12. Y. Watanabe, Y. Nakasha, Y. Kato, K. Odani, and M. Abe. A 9.6-Gb/s HEMT ATM Switch LSI with Event-Controlled FIFO. *IEEE Journal of Solid State Circuits*, 28(9):935–940, September 1993.

13. Router chip used in Intel's hypercube architectures. Personal communication with Loc Nguyen of Intel, Banff, September, 1993.

14. Michael C. McFarland, Alice C. Parker, and Raul Camposano. The high-level synthesis of digital systems. *Proceedings of the IEEE*, 78(2):301–318, February 1990.

15. Raul Camposano and Wayne Wolf. *High Level VLSI Synthesis.* Kluwer Academic Publishers, 1991.

16. Daneil D. Gajski (Ed.). *Silicon Compilation.* Addison-Wesley, 1988.

17. Daniel Gajski, Nikil Dutt, Allen Wu, and Steve Lin. *High-Level Synthesis; Introduction to Chip and System Design.* Kluwer Academic Publishers, 1992. ISBN 0-7923-9194-2.

18. Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Department of Computer Science, Stanford University, October 1984.

19. M.J.Stucki and J.R.Cox, Jr. Synchronization strategies. In *Proceedings of the Caltech Conference on VLSI*, pages 375–393, January 1979.

20. Mark E. Dean. *STRIP: A Self-Timed RISC Processor.* PhD thesis, Department of Computer Science, Stanford University, July 1992.

21. Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pein Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, 37(9):1005–1018, September 1988.

22. Cherrice Traver. A testable model for stoppable clock asics. In *Proceedings of the Fourth Annual IEEE International ASIC Conference and Exhibit*, pages P6–3.1, 1991.

23. Steven M. Nowick, Kenneth Y. Yun, and David L. Dill. Practical Asynchronous Controller Design. In *Proceedings of the International Conference on Computer Design*, pages 341–345, October 1992.

24. Ganesh Gopalakrishnan and Lüli Josephson. Towards amalgamating the synchronous and asynchronous styles. In *TAU 93: Timing Aspects of VLSI, Malente, Germany.* ACM, September 1993. *To Appear.*

25. Ganesh Gopalakrishnan and Venkatesh Akella. Vlsi asynchronous systems: Specification and synthesis. *Microprocessors & Microsystems*, 16(10), October 1992.

26. Venkatesh Akella and Ganesh Gopalakrishnan. SHILPA: A High-Level Synthesis System for Self-Timed Circuits. In *International Conference on Computer-aided Design, ICCAD 92*, pages 587–591, November 1992.

27. Venkatesh Akella. *An Integrated Framework for High-Level Synthesis of Self-timed Circuits.* PhD thesis, Department of Computer Science, University of Utah, 1992.

28. Erik Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits.* PhD thesis, Carnegie Mellon University, 1991.

29. Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer Design (ICCAD), IEEE*, pages 262–265, nov 1989.

30. S. M. Ornstein, M. J. Stucki, and W. A. Clark. A functional description of macromodules. In *Spring Joint Computer Conference.* AFIPS, 1967.

31. Ivan Sutherland. Micropipelines. *Communications of the ACM*, June 1989. *The 1988 ACM Turing Award Lecture.*

32. Erik Brunvand. Parts-r-us. *a chip aparts(s)...*. Technical Report CMU-CS-87-119, Carnegie Mellon University, May 1987.

33. Erik Brunvand. A cell set for self-timed design using actel FPGAs. Technical Report 91-013, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1991.

34. Ajay Khoche. An extended cell-set for self-timed designs. Technical Report 93-003, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1993. *Addendum to report* 91-013.

35. Kees van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI.* PhD thesis, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.

36. Ganesh Gopalakrishnan, Nick Michell, Erik Brunvand, and Steven M. Nowick. A correctness criterion for asynchronous circuit verification and optimization. *IEEE Transactions on Computer-Aided Design*, 1992. *Accepted for Publication.*

37. David L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-independent Circuits.* MIT Press, 1989. *An ACM Distinguished Dissertation.*

38. Ivan Sutherland and Bob Sproull. Chapter 6 and chapter 7 of ssa notes # 4702 and # 4703, volume 1, on interface state graphs. Technical memo, Sutherland, Sproull, and Associates, 1986.

39. Tam-Anh Chu. *Synthesis of Self-timed VLSI Circuits from Graph Theoretic Specifications.* PhD thesis, Department of EECS, Massachusetts Institute of Technology, September 1987.

40. Kenneth Y. Yun, David L. Dill, and Steven M. Nowick. Synthesis of 3d asynchronous state machines. In *Proceedings of the International Conference on Computer Design*, pages 346–350, October 1992.

41. Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits and systems. *Distributed Computing*, (1):197–204, 1986.

42. David L. Dill, Steven M. Nowick, and Robert F. Sproull. Specification and automatic verification of self-timed queues. *Formal Methods in System Design*, 1(1):29–62, July 1992.

43. Stephen Nowick. *Automatic Synthesis of Burst-mode Asynchronous Controllers.* PhD thesis, Stanford University, 1993. Ph.D. Thesis.

44. Steve Nowick, Mark Dean, David Dill, and Mark Horowitz. The design of a high-performance cache controller: A case study in asynchronous synthesis. In T.N. Mudge, V. Milutinovic, and L. Hunter, editors, *Proceedings of the 26th Annual Hawaiian International Conference on System Sciences, Volume 1*, pages 419–427, January 1993.

45. Ganesh Gopalakrishnan and Prabhat Jain. Some recent asynchronous system design methodologies. Technical Report UUCS-TR-90-016, Dept. of Computer Science, University of Utah, Salt Lake City, UT 84112, 1990.

46. Steven Burns and Alain Martin. Performance analysis and optimization of asynchronous circuits. In Carlo Sequin, editor, *Advanced Research in VLSI : Proceedings of the 1991 University of California Santa Cruz Conference*, pages 71–86. The MIT Press, 1991. *ISBN 0-262-19308-6.*

47. Steven M. Nowick. *Personal Communication, 1992.*

A Details of the Examples Optimized

We now provide details of how, for each significant circuit considered in Table 5, the network to be optimized is specified, along with the constraints necessary to obtain a compact SPCTS and (in some cases) how the partitioning is done.

Various versions of QR42 were considered. The "trick" in specifying a QR42 module in Occam or hopCP is to treat each channel initially as if it were a wire, and then to ignore the acknowledgement handshake emitted by the channels, to get a "kosher" QR42. This trick works when wire communications alternate.

The following was the specification used for the hand-design for QR42:

```
(defun qr42-imp-w-channel-constraints-di ()
  (teval
   (hide '(x1 x2 x3)
 (compose (toggle r4 x3 x1)
  (buffer x3 r2) ;-- to delay-insensitize
  (join x1 a2 x2)
  (merge-element x3 x2 a4)
  (fourph-channel r4 a4) ;-- constraint on 4-phase interface
  (twoph-channel r2 a2)  ;-- constraint on 2-phase interface
   ))))
```

The circuit obtained from the Occam compiler, after a few hand-transformations, is the following (called Version 1 in the paper):

```
(defun mmqr42 ()
  (teval
   (hide '(rs1 as1 rs2 r11 r12 a12 a22 r22 r2i a4i)
   (compose
    (buffer r2i r2) ;-- the buffers are to delay-insensitize the QR42
    (buffer a4i a4) ;   prior to verification

    (fourph-channel r4 a4)  ;-- four-phase channel constraint
    (twoph-channel r2 a2)   ;-- two-phase channel constraint
    (join r4 rs1 as1)
    (join as1 rs2 a4i)
    (call r11 r2i r12 a12 rs1 as1)
    (call r2i r12 r22 a22 rs2 a4i)
    (join a12 a2 r22)
    (iwire a22 r11)
   ))))
```

The circuit obtained from SHILPA (Version 2) is

```
(defun qr42-shilpa ()
  (teval
   (hide '(a2_out a21 r22 a2i a11 a22 r11 r21 rs1 r4_out r12 rs2)
   (compose
    (join a2_out a21 r22)
    (iwire r22 a2i)
    (join a2i a11 r12)
    (iwire a22 r11)
    (call r11 a11 r21 a21 rs1 r4_out)
    (call r12 r21 r22 a22 rs2 a4);
    (buffer rs2 a4)
    (buffer r12 r2)
    (join r2 a2 a2_out)
    (join rs1 r4 r4_out)
    (twoph-channel r4 a4)
    (twoph-channel r2 a2)
   ))))
```

One thing nice about all the QR42's is that the parallel composition operator could bridge many of the macromodule to macromodule peephole optimizations identified by Erik Brunvand [28].

The circuit for Call-C Idiom is

```
(defun call-c-idiom ()
  (teval
   (hide '(rs ci)
   (compose
    (call r1 a1 r2 a2 rs as)
    (join rs b ci)
    (buffer ci c)
   ))))
```

The specification for decision-wait is:

```
; Molnar's decision-wait - 2x1
;
;    pref *[ r1? ; a1! | r2? ; a2! ]
; || pref *[ s? ; (a1! | a2!) ]

(defmacro decision-wait (r1 a1 r2 a2 s)
  `(petri-to-spcts 5 5
   '( (0 (0)   (1))
      (1 (1 3) (0 4))
      (2 (0)   (2))
      (3 (2 3) (0 4))
      (4 (4)   (3)))
   '(0 4)
   '(,r1 ,a1 ,r2 ,a2 ,s)
   '(,r1 ,r2 ,s)
   '(,a1 ,a2)))
```

A larger decision-wait was tried, but took oodles of time, so was aborted:

```
; Decision-wait needed in Udding-Josephs stack - 3x2
;
;    pref *[ r1? ; (a11! | a12!)
;          | r2? ; (a21! | a22!)
;          | r3? ; (a31! | a32!) ]
;
; || pref *[ s1? ; (a11! | a21! | a31!)
;          | s2? ; (a12! | a22! | a32!) ]

(defmacro decision-wait-3x2 (r1 r2 r3 s1 s2 a11 a12 a21 a22 a31 a32)
  `(petri-to-spcts 7 11
   '( (0 (0)   (1))
      (1 (0)   (2))
      (2 (0)   (3))
      (3 (4)   (5))
      (4 (4)   (6))
      (5 (1 5) (0 4))
      (6 (1 6) (0 4))
      (7 (2 5) (0 4))
      (8 (2 6) (0 4))
      (9 (3 5) (0 4))
      (10(3 6) (0 4)))
   '(0 4)
   '(,r1 ,r2 ,r3 ,s1 ,s2 ,a11 ,a12 ,a21 ,a22 ,a31 ,a32)
   '(,r1 ,r2 ,r3 ,s1 ,s2)
   '(,a11 ,a12 ,a21 ,a22 ,a31 ,a32)
   ))
```

The GVT (Global Virtual Time) computation arises in Time Warp simulation. The hopCP specification used was:

```
  GVT1 [] <=  ((rcin?z, lcin?y) -> pout!(min y z) ->  GVT1 [])
||
  GVT2 [] <=  (pin?x -> (lcout!x,rcout!x) -> GVT2[])
```

The circuit when compiled gives essentially two independent circuits, one for process GVT1 and the other for process GVT2. Since the circuits are so independent, it is foolish to try and compose them together! So, we do them separately, as simplegvt-part1 and simplegvt-part2, below:

```
(defun simplegvt-part1 ()
  (teval
   (hide '(t1 t11 t12)
 (compose
  (iwire pout_in t1)
  (buffer t1 t11)
  (buffer t1 t12)
  (join t11 lcin_in r2)
  (join t12 rcin_in r1)
  (buffer a2 lcin_out)
  (buffer a1 rcin_out)
  (join a1 a2 rmin)
  (buffer amin pout_out)

  (twoph-channel r1 a1)
  (twoph-channel r2 a2)
  (twoph-channel rmin amin)
  (twoph-channel pout_out pout_in)
  (twoph-channel lcin_in lcin_out)
  (twoph-channel rcin_in rcin_out)
   ))))
```

```
(defun simplegvt-part2 ()
  (teval
  (hide '(t1 t2)
 (compose
  (join lcout_in rcout_in t1)
  (iwire t1 t2)
  (join t2 pin_in r3)
  (buffer a3 pin_out)
  (buffer a3 lcout_out)
  (buffer a3 rcout_out)

  (twoph-channel lcout_out lcout_in)
  (twoph-channel rcout_out rcout_in)
  (twoph-channel pin_in pin_out)
  (twoph-channel r3 a3)
   ))))
```

Call3-merge is another optimization discussed in Erik Brunvand's PhD dissertation [28]. Erik's optimization rule is that call3-merge-optimization can be optimized to optimized-call3-merge. It was observed that call3-merge-optimization and optimized-call3-merge both reduced to the same composite automaton, and so, they both gave rise to the same burst-mode circuit.

```
(defun call3-merge-optimization ()
  (teval
   (hide '(a2 a3)
   (compose
    (call3 r1 a1 r2 a2 r3 a3 rs as)
    (merge-element a2 a3 out)))))

(defun optimized-call3-merge ()
  (teval
   (hide '(int)
   (compose
    (call r1 a1 int out rs as)
    (merge-element r2 r3 int)))))
```

One final circuit that passed Dill's composition operator but not the rest of the algorithm (*yet, that is*) is given below. Note my use of hierarchy even to make Dill's code run at an acceptably fast rate:

```
; This circuit appears on page 159 of Venkatesh's thesis.
; I've replaced the XORs by iwires
; also eliminated registers and in their place have r1 output a1 input, etc...

(defun multicast1 ()
  (teval
   (hide '(c1 a)
 (compose
  (join a2 a3 c1)
  (iwire c1 a)
  (join a a_in r1)
  (buffer a3 d_out)
  (buffer a2 c_out)
  ))))

(defun multicast2 ()
  (teval
   (hide '(in1 in2)
 (compose
  (bcel2 a1 in1 r2 in2 r3)
  (iwire c_in in1)
  (iwire d_in in2)
  (buffer a1 a_out)
  ))))

(setf *multicast1* (multicast1))
(setf *multicast2* (multicast2))


(defun hier-multicast ()
  (teval
   (compose
    *multicast1*
    *multicast2*

    (twoph-channel r1 a1)
    (twoph-channel r2 a2)
    (twoph-channel r3 a3)

    (twoph-channel c_out c_in)
    (twoph-channel d_out d_in)
    )))
```