Projection: A Synthesis Technique for Concurrent Systems*

Rajit Manohar
School of Electrical Engineering
Cornell University
Ithaca, NY 14853

Tak-Kwan Lee† and Alain J. Martin
Computer Science 256-80
California Institute of Technology
Pasadena, CA 91125

Abstract

We present a process decomposition technique for the design of pipelined asynchronous circuits. The technique is simple to use, and is based on projecting a program on different sets of variables. We provide conditions under which the technique can be applied, and show how it can be used to decompose complex concurrent programs.

1. Introduction

The formal synthesis approach to designing asynchronous VLSI systems begins with a simple, sequential description of the specification to be implemented. The end result of synthesis is a highly complex concurrent system which is a valid implementation of the original sequential specification. The transformation of a sequential specification to the final concurrent system is done using semantics-preserving transformations; therefore, if we know that the original sequential specification is correct, the final concurrent implementation is correct as well [6].

The main program transformation used in the design of the Caltech microprocessor [7] was *process decomposition* [6]. The transformation was used to decompose a sequential program into a concurrent one in a systematic manner.

In the design of an asynchronous MIPS microprocessor, we used a number of program transformations geared toward increasing the amount of concurrency in the system by finely pipelining various parts of the computation [8]. In this paper, we present a framework in which we can analyze a large class of program transformations that were used in the design of the MIPS processor.

We present a method for the decomposition of programs into parts based on the notion of *projection*. The idea is inspired by examining the data-dependencies in the computation, and attempting to eliminate all unnecessary synchronization. In the simplest case, we take a program and syntactically project various variables into separate concurrent processes that are unsynchronized. For instance, we would decompose process

$$*[\ L?x; P?y;\ \ R!x; Q!y]$$

into

$$*[\ L?x; R!x\ ]\ \|\ *[\ P?y; Q!y\ ]$$

since the two parts of the original process are not data-dependent. (The processes are written in CHP notation, a brief description of which is provided in the appendix.) Such transformations are not always semantics-preserving, and we examine the conditions under which we can apply them while preserving the correctness of the computation. The conditions amount to demonstrating that certain programs are *locally slack elastic*—a property which can be established for a relatively large class of programs [5].

We present the design of the *writeback* process in the asynchronous MIPS processor using the notion of projection. The derivation of the final concurrent implementation is instructive, in that it shows the types of program transformations that are necessary in order to achieve high performance in an asynchronous design.

2. Slack Elastic Programs

We specify a distributed computation using CHP, a variant of CSP [3]. (A brief description is provided in the appendix.) The processes in the computation interact by exchanging messages over first-in first-out channels. Each channel in the computation has a fixed amount of *slack*, or buffering, which specifies the maximum number of outstanding messages on a channel. The *probe* of a communication action is a boolean that is true when the communication action can complete without suspending.

*The research described in this report was sponsored in part by the Defence and Advanced Research Projects Agency and monitored by the Office of Army Research

†Now with Epson Research and Development, San Jose, CA 95134

Slack elastic programs are those whose correctness is preserved when the slack on channels in the system is increased [5]. We only consider programs that are deadlock-free, where no communication action can remain suspended forever. Increasing the slack on a synchronization channel is a transformation that can be used to increase the amount of concurrency in the system. A particular instance of increasing concurrency is adding pipelining. For example, consider:

$p1 \equiv$ `*[` $L?x;\ \ R!g(f(x))$ `]`

If we can increase the slack on channel R, we can replace this program with

$p2 \equiv$ `*[` $L?x;\ \ R'!f(x)$ `]` $\|$ `*[` $R'?y;\ \ R!g(y)$ `]`

in which the computation $g(f(x))$ has been decomposed into two pipeline stages. As far as the rest of the computation is concerned, the only observable difference between $p1$ and $p2$ is that the slack on channel R (or L) has been increased by one. Slack elasticity is a property of closed systems, since we must consider the process as well as its environment in order to determine if a system is slack elastic.

Since increasing the slack on a communication channel will usually increase the number of possible traces [9], we do not use traces directly to compare executions. Instead, all properties of the system are specified in terms of the sequence of messages sent and received on communication channels. Information about the relative order of communication actions on different channels is not considered when comparing computations.

When a program is deterministic and does not probe any channels, the system is slack elastic. This is because all communication actions only depend on the data values that are transferred on channels—intuitively, all communication actions are controlled by local variables in processes. In this case, changing the slack of a channel does not affect what the system computes. The notion of what happens when the slack of a communication channel is increased is formalized in Theorem 2 from [5]. Theorem 2 states that increasing the slack on a channel can change the behavior of a system only if it causes the introduction of non-deterministic choices within some process in the system. (This theorem is applicable only when examining systems in which processes do not share variables.) If we can show that we cannot introduce any non-determinism in the system, we are guaranteed that the system will not change its behavior when the slack on a channel is increased.

To analyze a component of a concurrent system, we introduced the concept of *locally slack elastic* systems. Given an open system $\mathcal{S}$ we define the concept of local slack elasticity of $\mathcal{S}$ as follows. We examine the parallel composition of $\mathcal{S}$ with any other system $\mathcal{S}'$ such that $\mathcal{S}\|\mathcal{S}'$ is closed. If adding slack to any channel in the closed system does not introduce any non-determinism in $\mathcal{S}$, $\mathcal{S}$ is said to be locally slack elastic. Once again, it should be clear that if $\mathcal{S}$ does not probe any channels, $\mathcal{S}$ is locally slack elastic. If we compose a collection of locally slack elastic systems to construct a closed system, the resulting system is locally slack elastic as well.

We state the following result which provides sufficient conditions for local slack elasticity:

Theorem 1 (determinism) *Let $\mathcal{S}$ be an open system in which the guards of selection statements are syntactically mutually exclusive and there are no probed channels. Then $\mathcal{S}_\emptyset$ is locally slack elastic.*

The interested reader is referred to [5] for more details and proofs of other sufficient conditions for slack elasticity.

In this paper we focus on locally slack elastic systems, and when we say that two systems are equivalent, we mean they are equivalent when composed with other locally slack elastic systems.

3. Projection

We would like to be able to decompose a process into parts and reduce the synchronization among parts that are not data-dependent. However, as was described above, reducing synchronization is not guaranteed to preserve the correctness of the original computation.

To address this issue, we adopt the following strategy. We begin by decomposing a process into parts, and keeping the parts tightly synchronized so as to preserve the semantics of the original computation. This decomposition is done via a *partition* function that systematically separates a CHP program fragment into two parts based on the structure of the program. The purpose of this part of the transformation is to syntactically divide the program fragment into two parts in a manner that guarantees correctness.

The second part of the transformation uses the concept of slack elasticity to introduce concurrency among the two parts that were decomposed by partitioning. We will loosen the synchronization constraints by assuming that we are designing our system by composing locally slack elastic components.

3.1 Partitions

A *partition* of a statement is a pair of statements such that their concurrent composition with a special *sequencer* process behaves exactly like the original statement, assuming that the two partitioned parts have infinite execution traces.

We introduce the program templates $LSync(\cdot)$ and $RSync(\cdot)$ to clarify the presentation of partitions. These template will be used to synchronize the two parts of the partition by using a "fork and join" synchronization mechanism when composed with the sequencer. These are defined as follows:

$$\begin{aligned} LSync(S) &\equiv \quad C_1?; S; C_1? \\ RSync(S) &\equiv \quad C_2?; S; C_2? \end{aligned}$$

where C_1 and C_2 are fresh channels. Given a partition of the form $\langle LSync(A)\ , RSync(B)\rangle$, the concurrent composition of the two parts would be

$$C_1?; A; C_1? \ \| \ C_2?; B; C_2?$$

which, when composed with $C_1! \bullet C_2!; C_1! \bullet C_2!$, would implement a "fork and join" synchronization.

We define a function $\mathbf{part}(S)$ that maps statement S to a set of pairs, where each pair is a valid partition of S. This function is defined by structural induction on the body of S as follows:

- $\mathbf{part}(A) = \{\langle LSync(\mathbf{skip})\ , RSync(A)\rangle,$
  $\langle LSync(A)\ , RSync(\mathbf{skip})\rangle\}$
  for any elementary statement A. We will define the partition function so that if $\langle p\ , q\rangle$ is in the set of valid partitions, so is $\langle q\ , p\rangle$ with $Lsync$ and $Rsync$ interchanged.

- $\mathbf{part}(S\|T) = \{\langle LSync(A)\ , RSync(\mathbf{skip})\rangle,$
  $\langle LSync(\mathbf{skip})\ , RSync(A)\rangle,$
  $\langle LSync(S)\ , RSync(T)\rangle,$
  $\langle LSync(T)\ , RSync(S)\rangle\}$
  where A represents the statement $S\|T$. Parallel composition can be partitioned without introduction of any additional synchronization. Note that, for simplicity, parallel composition is one of the terminal symbols in our structural definition of partitions.

- $\mathbf{part}(S; T) = \{s, t \mid s \in \mathbf{part}(S) \wedge t \in \mathbf{part}(T) \triangleright$
  $\langle \mathbf{l}(s); \mathbf{l}(t)\ , \mathbf{r}(s); \mathbf{r}(t)\rangle\}$
  The partition of the sequential composition of two statements is derived by examining all possible partitions of the two statements, and sequentially composing their components. The functions $\mathbf{l}(\cdot)$ and $\mathbf{r}(\cdot)$ are used to extract the left and right components of a pair. (An explanation of the set notation can be found in the appendix.)

Example: The partitions of $x := 0$ are

$$\{\langle LSync(x := 0)\ , RSync(\mathbf{skip})\rangle,$$
$$\langle LSync(\mathbf{skip})\ , RSync(x := 0)\rangle\}$$

∎

Example: The set of partitions of $A; B; C$ where A, B, C are elementary statements contains the following set:

$$\begin{aligned} \{&\langle LSync(A); LSync(B); LSync(C), \\ &\quad RSync(\mathbf{skip}); RSync(\mathbf{skip}); RSync(\mathbf{skip})\rangle, \\ &\langle LSync(A); LSync(B); LSync(\mathbf{skip}), \\ &\quad RSync(\mathbf{skip}); RSync(\mathbf{skip}); RSync(C)\rangle, \\ &\langle LSync(A); LSync(\mathbf{skip}); LSync(C), \\ &\quad RSync(\mathbf{skip}); RSync(B); RSync(\mathbf{skip})\rangle, \\ &\langle LSync(A); LSync(\mathbf{skip}); LSync(\mathbf{skip}), \\ &\quad RSync(\mathbf{skip}); RSync(B); RSync(C)\rangle\} \end{aligned}$$

The rest of the partitions can be obtained by the symmetric closure of the set (i.e., for each pair $\langle x\ , y\rangle$, add $\langle y\ , x\rangle$ to the set of partitions with $LSync$ and $RSync$ interchanged). Observe that for any pair $\langle x\ , y\rangle$ in the set of partitions, both components x and y of the partition locally preserve the ordering among actions in the original program. ∎

Partitions consist of two pieces of the original computation that, when composed in parallel with a sequencer process, precisely implement the original computation. We formalize this observation by the following theorem.

Theorem 2 (partition) *Let $E(\cdot)$ be a program template, and consider any pair $\langle L\ , R\rangle \in \mathbf{part}(P)$. Assume that in any deadlock-free execution of $E(P)$, action P is encountered infinitely often without any parallel duplication of P. Then,*

$$E(L) \ \| \ *[\ R\ ] \ \| \ *[\ C_1! \bullet C_2!\ ] \ \equiv \ E(P)$$

Proof: (Sketch) Note that the requirement of infinite communication on channels C_1 and C_2 simply ensures that the process $*[C_1! \bullet C_2!]$ does not deadlock. If we permit process $*[C_1! \bullet C_2!]$ to deadlock, we can lift the infinite communication restriction.

Observe that L begins with $C_1?$ and R begins with $C_2?$—by definition of $\mathbf{part}(\cdot)$. Therefore, L cannot begin execution until R begins execution, and vice versa. Similarly, L cannot finish execution until R finishes (and vice versa), because they both end with $C_1?$ and $C_2?$ respectively. Therefore, it suffices to show that when $L\|R$ are composed with a process that repeatedly executes $C_1! \bullet C_2!$, they implement P. This can be shown by structural induction on the definition of $\mathbf{part}(\cdot)$.

Let $\langle L\ , R\rangle \in \mathbf{part}(P)$. The key observation is that the following properties hold: (a) L and R begin with $C_1?$ and $C_2?$ respectively; (b) L and R execute a matching number of communications on C_1 and C_2 (observe that this holds in the definition of $\mathbf{part}(\cdot)$ given above); (c) The environment executes $C_1! \bullet C_2!$, which enforces that the number of completed communication actions on channels C_1 and C_2 is equal.

The proof of equivalence is straightforward for the cases when P is an elementary statement or the parallel composition of two statements. We consider the case of sequential composition. Given $P = S;T$, we have $L = Ls;Lt$ and $R = Rs;Rt$ where Ls, Lt, Rs, Rt are obtained from the partitions of S and T by the appropriate substitution of $Sync(\cdot)$ actions with communications on C_1 and C_2. By induction, $Ls \| Rs$ implements S and $Lt \| Rt$ implements T (when composed with the process $*[C_1! \bullet C_2!]$). Since the number of C_1 actions in Ls equals the number of C_2 actions in Rs, we are guaranteed that the number of completed C_1 actions equals the number of completed C_2 actions at the semicolon between Ls and Lt—and at the semicolon between Rs and Rt. Since both Lt and Rt begin with a communication action on C_1 and C_2 respectively, they begin simultaneously—concluding the proof. ∎

Partitioning a process is a generalized form of process decomposition. In process decomposition, a CHP program was split into two parts by using the following transformation:

$$*[\ldots;\ S;\ \ldots] \;=\; *[\ldots;\ A;\ldots] \;\|\; *[[\overline{A}];S;A]$$

where A is a slack zero communication action. We can use partitioning to obtain a similar decomposition:

$$\begin{aligned} *[\ldots;\ S;\ \ldots] \;=\; & (*[\ldots;\ C_1?;C_1?;\ \ldots] \\ & \|*[\ C_2?;S;C_2?\ ] \\ & \|*[\ C_1! \bullet C_2!\ ]) \end{aligned}$$

We can replace $C_1?$ actions with $C_2!$ actions and eliminate the sequencer process, since this does not change any synchronization behavior. We obtain:

$$\begin{aligned} *[\ldots;\ S;\ \ldots] \;=\; & (*[\ldots;\ C_1!;C_1!;\ \ldots] \\ & \|*[\ C_1?;S;C_1?\ ]) \end{aligned}$$

(Indeed, when transformed into handshaking expansions, using a two phase protocol on channel C_1 would result in a reshuffled version of the system obtained through process decomposition.)

3.2 Projection

Given a statement S, we restrict our attention to *valid* partitions $\langle A\,,B\rangle \in \textbf{part}(S)$—partitions such that A and B do not share any variables or communication ports. Disallowing shared variables permits us to use the theory of slack elastic systems to reason about partitions. Allowing the actions from A and B to overlap in any manner would not be permissible without this condition. For example, consider the process

$$x\uparrow;\ \ x\downarrow$$

One possible partition of it is shown below:

$$\langle LSync(x\uparrow);LSync(\textbf{skip}),\ \ RSync(\textbf{skip});RSync(x\downarrow)\rangle$$

The "$Sync$" actions ensure that the two processes do not modify variable x simultaneously. If we attempted to remove the synchronizations, both $x\uparrow$ and $x\downarrow$ might execute concurrently, causing erroneous executions.

We introduce concurrency by adding slack on channels C_1 and C_2 that implement the "$Sync$" action. As we increase the slack on these channels, we loosen the synchronization constraints we have imposed through "$Sync$" operations. This transformation cannot be justified in general. We therefore restrict ourselves to considering systems such that the partitioned system (with the sequencer process) is locally slack elastic, and where channels C_1 and C_2 are slack elastic. The theorems stated earlier provide sufficient conditions that would ensure that the system is locally slack elastic.

We now let the slack on channels C_1 and C_2 be infinite. Notice that the sequencer process can run arbitrarily ahead of the the two partitions that it is synchronizing. As a result, the $LSync$ and $RSync$ actions no longer serve any purpose, and can be eliminated! The final result is a decomposition of the original process in which the two parts can execute concurrently.

Theorem 3 (projection) *Let $E(\cdot)$ be a program template, and let $\langle L\ ,R\rangle$ be a valid partition of process P. Replace $LSync(A)$ by A in L (call this process L') and $RSync(B)$ by B in R (call this process R'). Assume that: (a) in any deadlock-free execution of $E(P)$, action P is encountered infinitely often without any parallel duplication of P; (b) $E(L)\|*[R]\|*[C_1!\bullet C_2!]$ is locally slack elastic; (c) $E(L)$ and R do not share variables. Then, $E(L')\|*[R']$ is a valid implementation of $E(P)$.*

Proof: (Sketch) By Theorem 2, we are guaranteed that $E(L)\|*[R]\|*[C_1! \bullet C_2!]$ is equivalent to $E(P)$. By assumption (b), the system is locally slack elastic. Therefore, we can increase the slack on channels C_1 and C_2 without affecting the correctness of the entire system. When the slack on C_1 and C_2 is infinite, we can eliminate all communication actions on C_1 and C_2, because eliminating them does not affect the possible execution traces that could occur—concluding the proof. ∎

By the nature of this construction, we have a transformation that effectively *projects* a process onto two disjoint sets of variables.

Example: Consider the following process:

$$*[\ L?x;R!x;P?y;Q!y\ ]$$

A valid partition of the body of the loop would be:

$$\langle LSync(L?x); LSync(R!x); LSync(\textbf{skip}); LSync(\textbf{skip}),$$
$$RSync(\textbf{skip}); RSync(\textbf{skip}); RSync(P?y); RSync(Q!y)\rangle$$

The two components of the partition do not share variables or ports. The system is locally slack elastic, because there are no selection statements. Therefore, we can apply Theorem 3 to argue that the original process is equivalent to:

$$*[L?x; R!x; \textbf{skip}; \textbf{skip}] \;\|\; *[\textbf{skip}; \textbf{skip}; P?y; Q!y]$$

which is the same as:

$$*[\;L?x; R!x\;] \;\|\; *[\;P?y; Q!y\;]$$

We can think of this as a transformation that syntactically projected the original process onto two sets of variables—$\{L?, R!, x\}$ and $\{P?, Q!, y\}$—to obtain the final result. ∎

4. Examples

In this section we use projection to show how various parts of a computation can be decomposed so as to increase the amount of concurrency in the final implementation.

4.1 Simple Pipelining

Consider the earlier example where we pipelined the computation of $g(f(x))$. The original program was

$$*[\;L?x;\;\; R!g(f(x))\;]$$

This program is equivalent to

$$*[\;L?x;\;\; y := f(x);\;\; R!g(y)\;]$$

We can replace the assignment with communication actions using the communication axiom [3], to obtain

$$*[\;L?x;\;\; (R'!f(x)\|R'?y);\;\; R!g(y)\;]$$

We project the program onto the two disjoint sets $\{L?, x, R'!\}$ and $\{R'?, y, R!\}$ to obtain

$$*[\;L?x;\;\; R'!f(x)\;] \;\|\; *[\;R'?y;\;\; R!g(y)\;]$$

which is the pipelined implementation. This example illustrates that the introduction of new variables and internal communication actions permits the systematic decomposition of a process into multiple parts that exchange data as required. The first step was to introduce variable y to hold the intermediate result $f(x)$. Next, the assignment to y was replaced with a communication action. Finally, we applied projection to obtain the pipelined implementation.

4.2 Data-dependent Synchronization

Consider the following program. It receives a bit on channel C, and depending on the value either discards the input received on channel A, or sends it out on channel X along with copying the value received on B to Y.

$$\begin{array}{l} *[\;\; A?x;\;\; C?c; \\ \quad [\;\; c = \textbf{true} \;\longrightarrow\; B?y; X!x,\, Y!y \\ \quad [\!]\;\; c = \textbf{false} \longrightarrow \textbf{skip} \\ \quad] \\ ] \end{array}$$

Observe that the value communicated on channel Y in the first guard of the selection statement is received on B, and does not use the values c or x. We use the environment template

$$\begin{array}{l} E(P) \equiv \\ \quad *[\;\; A?x; C?c; \\ \quad\quad [\;\; c = \textbf{true} \longrightarrow \;\; P \\ \quad\quad [\!]\;\; c = \textbf{false} \longrightarrow \textbf{skip} \\ \quad\quad] \\ \quad] \end{array}$$

We project $B?y; X!x, Y!y$ onto two disjoint sets $\{X!, x\}$ and $\{B?, Y!, y\}$; the two parts after projection are $X!x$ and $B?y;\; Y!y$ respectively. Since $E(X!x)$ does not share any variables with $B?y;\; Y!y$, we obtain the following decomposition:

$$\begin{array}{l} *[\;\; A?x; C?c; \\ \quad [\;\; c = \textbf{true} \longrightarrow X!x \;\; [\!] \;\; c = \textbf{false} \longrightarrow \textbf{skip}\;\;] \\ ] \\ \| \\ *[B?y;\; Y!y] \end{array}$$

Note that both processes are simpler to implement than the original process. Also, the actions on channels A and B are no longer synchronized. The second process has no internal selection statement, and does not even need to examine the value of variable c. Since our model of computation does not consider the relative order of actions on channels C and B, we can decompose the original computation into these two processes. Note that an environment where a process always attempts to send a message on channel B, but where C always receives a false value is ruled out because that would imply that the send action on B (in the environment) in the original computation was suspended forever—violating the absence of deadlock requirement on slack elastic programs.

5. Selections and Loops

The definition of partitions can be extended to handle selection statements and loops. When a selection statement is decomposed into two parts, we must ensure that we do not change the value of the guards of the selection statement. The definition of partitions for selection statements and loops is given below. We will use A to refer to the argument of **part** on the LHS for the remainder of this definition.

- $\mathbf{part}([G_1 \wedge H_1 \to S_1 \,[]\, \cdots \,[]\, G_n \wedge H_n \to S_n]) =$
  $\{\langle LSync(A)\ ,\ RSync(\mathbf{skip})\rangle,$
  $\quad\langle LSync(\mathbf{skip})\ ,\ RSync(A)\rangle\} \cup$
  $\{s_1, \ldots, s_n \mid s_i \in \mathbf{part}(S_i) \triangleright$
  $\langle LSync(\mathbf{skip});\ [G_1 \to LSync(\mathbf{skip}); \mathbf{l}(s_1)$
  $\qquad [] \cdots$
  $\qquad [] G_n \to LSync(\mathbf{skip}); \mathbf{l}(s_n)],$
  $RSync(\mathbf{skip});\ [H_1 \to RSync(\mathbf{skip}); \mathbf{r}(s_1)$
  $\qquad [] \cdots$
  $\qquad [] H_n \to RSync(\mathbf{skip}); \mathbf{r}(s_n)]\rangle\}$

  where $G_i \equiv H_i$, for all i. The first part of the expression is the trivial partition, where the entire statement is in one half of the partition. The rest of the definition consists of the case when we partition each statement in each guarded command. The guards can be partitioned as well; however, they can only be partitioned when they are equivalent in the two partitions, and equivalent to the guards used in the original program. Note that G_i and H_i can be interchanged freely. The partition of a selection statement is the union of all such conjunctive decompositions of the guards. The additional synchronization sequences the guard evaluation and the execution of the statement in the guarded command.

- $\mathbf{part}(*[G_1 \wedge H_1 \to S_1 \,[]\, \cdots \,[]\, G_n \wedge H_n \to S_n]) =$
  $\{\langle LSync(A)\ ,\ RSync(\mathbf{skip})\rangle,$
  $\quad\langle LSync(\mathbf{skip})\ ,\ RSync(A)\rangle\} \cup$
  $\{s_1, \ldots, s_n \mid s_i \in \mathbf{part}(S_i) \triangleright$
  $\langle LSync(\mathbf{skip});$
  $*[G_1 \to LSync(\mathbf{skip}); \mathbf{l}(s_1); LSync(\mathbf{skip})$
  $[] \cdots$
  $[] G_n \to LSync(\mathbf{skip}); \mathbf{l}(s_n); LSync(\mathbf{skip})$
  $],$
  $RSync(\mathbf{skip});$
  $*[H_1 \to RSync(\mathbf{skip}); \mathbf{r}(s_1); RSync(\mathbf{skip})$
  $[] \cdots$
  $[] H_n \to RSync(\mathbf{skip}); \mathbf{r}(s_n); RSync(\mathbf{skip})$
  $]\rangle\}$

  where $G_i \equiv H_i$, for all i.

Non-deterministic loops and selection statements cannot be partitioned. This is because the implementation of a non-deterministic selection statement is free to pick any alternative that has a true guard. This implies that the two partitions might pick different alternatives since the choice is demonic (cf. [2]), thereby resulting in a system that would not be equivalent to the original program.

Example: Consider the following process. It receives inputs on channels A and B, and a control input on channel C. If the value received on C is **true**, the input on A is sent on channel X; otherwise, the input on B is sent on channel Y.

$$
\begin{array}{l}
*[\ A?x; B?y; C?c; \\
\quad [\ c = \mathbf{true} \ \longrightarrow\ X!x \\
\quad []\ c = \mathbf{false} \longrightarrow Y!y \\
\quad] \\
]
\end{array}
$$

We assume that the input on C is known early, and we would like to eliminate synchronization among actions A and B. To partition the selection statement, and to decouple the channel C from A and B we introduce variable ca and cb that is equivalent to c:

$$
\begin{array}{l}
*[\ A?x; B?y; C?c; ca := c; cb := c; \\
\quad [\ ca = \mathbf{true} \wedge cb = \mathbf{true} \ \longrightarrow\ X!x \\
\quad []\ ca = \mathbf{false} \wedge cb = \mathbf{false} \longrightarrow Y!y \\
\quad] \\
]
\end{array}
$$

The first step is to replace the assignment statements with communication actions, to obtain:

$$
\begin{array}{l}
*[\ A?x; B?y; C?c; (Ca!c \| Ca?ca); (Cb!c \| Cb?cb); \\
\quad [\ ca = \mathbf{true} \wedge cb = \mathbf{true} \ \longrightarrow\ X!x \\
\quad []\ ca = \mathbf{false} \wedge cb = \mathbf{false} \longrightarrow Y!y \\
\quad] \\
]
\end{array}
$$

Next, we project this process onto $\{C?, c, Ca!, Cb!\}$ and the rest of the ports and channels. The resulting system is:

$$
\begin{array}{l}
*[\ A?x; B?y; Ca?ca; Cb?cb; \\
\quad [\ ca = \mathbf{true} \wedge cb = \mathbf{true} \longrightarrow X!x \\
\quad []\ ca = \mathbf{false} \wedge cb = \mathbf{false} \longrightarrow Y!y \\
\quad] \\
] \\
\| \\
*[\ C?c; Ca!c; Cb!c\ ]
\end{array}
$$

Finally, the first process can be projected onto $\{A?, x, Ca?, ca, X!\}$ and $\{B?, y, Cb?, cb, Y!\}$ to obtain:

$$\begin{array}{l}
*[\ \ A?x;\, Ca?ca; \\
\quad [\ \ ca = \textbf{true} \longrightarrow X!x \\
\quad \square\ \ ca = \textbf{false} \longrightarrow \textbf{skip} \\
\quad] \\
\ ] \\
\| \\
*[\ \ B?y;\, Cb?cb; \\
\quad [\ \ cb = \textbf{true} \longrightarrow \textbf{skip} \\
\quad \square\ \ cb = \textbf{false} \longrightarrow Y!y \\
\quad] \\
\ ] \\
\| \\
*[\ \ C?c;\, Ca!c;\, Cb!c\ \ ]
\end{array}$$

In the final program, we have decoupled channels A and B. Since the processes are all locally slack elastic, we can add slack on channels Ca and Cb to further decouple the control channel C from channels A and B. ∎

Quite often, we are faced with a program where we would like to partition a selection statement, but we cannot write the guards in the form required by our definition of **part**$(\cdot)$. Consider the selection statement shown below:

$$[G_1 \longrightarrow S_1\ \square\ \cdots\ \square\, G_n \longrightarrow S_n]$$

We make the assumption that the guards of the selection statement are *stable*, i.e., the environment cannot change their value from true to false. (This is the case when the guards do not contain negated probes.) We introduce a fresh variable g as follows:

$$\begin{array}{l}
[G_1 \longrightarrow g := 1\ \square\ \cdots\ \square\, G_n \longrightarrow g := n]; \\
[G_1 \longrightarrow S_1\ \square\ \cdots\ \square\, G_n \longrightarrow S_n]
\end{array}$$

Next, we strengthen the guards in the second statement to the following:

$$\begin{array}{l}
[G_1 \longrightarrow g := 1\ \square\ \cdots\ \square\, G_n \longrightarrow g := n]; \\
[G_1 \wedge g = 1 \longrightarrow S_1\ \square\ \cdots\ \square\, G_n \wedge g = n \longrightarrow S_n]
\end{array}$$

We replace the assignment to g with communication actions, using a fresh slack zero channel G.

$$\begin{array}{l}
[G_1 \longrightarrow G!1 \| G?g\ \square\ \cdots\ \square\, G_n \longrightarrow G!n \| G?g]; \\
[G_1 \wedge g = 1 \longrightarrow S_1\ \square\ \cdots\ \square\, G_n \wedge g = n \longrightarrow S_n]
\end{array}$$

Observe that $G?g$ is common to all alternatives in the selection statement. Therefore, we can write:

$$\begin{array}{l}
(G?g \| [G_1 \longrightarrow G!1\ \square\ \cdots\ \square\, G_n \longrightarrow G!n]); \\
[G_1 \wedge g = 1 \longrightarrow S_1\ \square\ \cdots\ \square\, G_n \wedge g = n \longrightarrow S_n]
\end{array}$$

So far, we have not changed the computation performed by the original guarded command. This final form can be partitioned, because the guards of the selection statement are in the required form. We call this sequence of steps *control duplication*, as we have created a copy g of control flow information.

6. A Case Study

We present the decomposition of the *writeback* unit from the asynchronous MIPS processor [8]. The MIPS Instruction Set Architecture (ISA) specifies that any exception caused by an instruction is *precise* [4]. A precise exception mechanism has to guarantee that that the instruction that caused the exception and all instructions following it until the first instruction of the exception handler do not modify any observable state of the processor. The observable state in the MIPS ISA consists of the memory, special purpose registers, and general purpose registers. The writeback unit coordinates this behavior, controlling when a particular instruction is permitted to modify the state of the processor. The sequential CHP program for the writeback unit is:

$$\begin{array}{l}
WB \equiv\ \ valid\uparrow; \\
\quad *[\ \ UN?un,\, UZ?uz,\, VA?va,\, EPC?epc; \\
\qquad [un = 0 \longrightarrow EX0?e \\
\qquad \square\, un = 1 \longrightarrow EX1?e \\
\qquad \square\, un = 2 \longrightarrow e := \textbf{none} \\
\qquad]; \\
\qquad [uz = 1 \longrightarrow REG!(valid \wedge (e = \textbf{none})) \\
\qquad \square\, uz = 0 \wedge un = 1 \longrightarrow \\
\qquad\qquad MEM!(valid \wedge (e = \textbf{none})) \\
\qquad \square\, uz = 0 \wedge un = 2 \longrightarrow \\
\qquad\qquad MULT!(valid \wedge (e = \textbf{none})) \\
\qquad]; \\
\qquad [valid \wedge (e \neq \textbf{none}) \longrightarrow \\
\qquad\qquad valid\downarrow,\, EX,\, CP0pc!epc,\, CP0e!e \\
\qquad \square\, va \longrightarrow valid\uparrow \\
\qquad \square\, \textbf{else} \longrightarrow \textbf{skip} \\
\qquad] \\
\quad]
\end{array}$$

The information received on various input channels is as follows:

- UN: where to read the exception information for the next instruction to be processed.
- UZ: 1 if the instruction writes its results to the register file, 0 otherwise.
- VA: 1 if the instruction is the first instruction of the exception handler, 0 otherwise.
- EPC: the program counter for the instruction.
- $EX0$, $EX1$: the type of exception from different execution units.

The writeback process is in two states: executing valid instructions ($valid$ is true); canceling the results of instructions ($valid$ is false). The variables uz and va are one bit wide; un is two bits wide; epc is a 32-bit quantity.

The writeback process reads the exception information from the appropriate channel. It sends permission to modify the state of the processor (or lack thereof) to the appropriate part of the processor (register file, memory unit, or multiplier/divider unit which has local registers). If an exception is detected, it notifies the $PCUNIT$ (via EX) and sends the program counter and exception type to the CP0 on channels $CP0pc$ and $CP0e$ respectively. Finally, it updates the valid bit. (For a more detailed description, readers are referred to [8] and [4]). Note that the writeback process is locally slack elastic, because it does not probe any channels.

First, we decompose the computation of $valid$ out of the writeback. The computation of valid depends on va, $valid$, and the value of e. Note that va is not used by any other part of the computation. However, the rest of the computation uses $valid$. We introduce a copy of valid, $valid2$, that is used by the rest of the program. To project the last selection statement (containing epc and e), we introduce a copy of e as well.

After introducing the appropriate copies of $valid$ and e, and adding communication actions (as in the examples earlier), we obtain:

$$
\begin{array}{l}
WB0 \equiv \quad V?valid2;\\
\quad *[\ \ UN?un,\ UZ?uz;\\
\qquad [un = 0 \longrightarrow EX0?e\\
\qquad []\,un = 1 \longrightarrow EX1?e\\
\qquad []\,un = 2 \longrightarrow e := \textbf{none}\\
\qquad];\ \ Enone!(e \neq \textbf{none}), E2!e,\\
\qquad [uz = 1 \longrightarrow REG!(valid2 \wedge (e = \textbf{none}))\\
\qquad []\,uz = 0 \wedge un = 1 \longrightarrow\\
\qquad\qquad\quad MEM!(valid2 \wedge (e = \textbf{none}))\\
\qquad []\,uz = 0 \wedge un = 2 \longrightarrow\\
\qquad\qquad\quad MULT!(valid2 \wedge (e = \textbf{none}))\\
\qquad];\ \ V?valid2\\
\quad]
\end{array}
$$

$$
\begin{array}{l}
WB1 \equiv\\
\qquad valid\uparrow;\\
\qquad V!valid;\\
\quad *[\ \ VA?va; EPC?epc; Enone?en; E2?e2;\\
\qquad [valid \wedge en \longrightarrow\\
\qquad\qquad\quad valid\downarrow, EX, CP0pc!epc, CP0e!e2\\
\qquad []\,va \longrightarrow valid\uparrow\\
\qquad []\,\textbf{else} \longrightarrow \textbf{skip}\\
\qquad];\\
\qquad V!valid\\
\quad]
\end{array}
$$

We now focus on the second process. Observe that action $V!valid$ is performed on initialization, and after each iteration of the non-terminating outer loop in $WB1$. Therefore, it can be rewritten as:

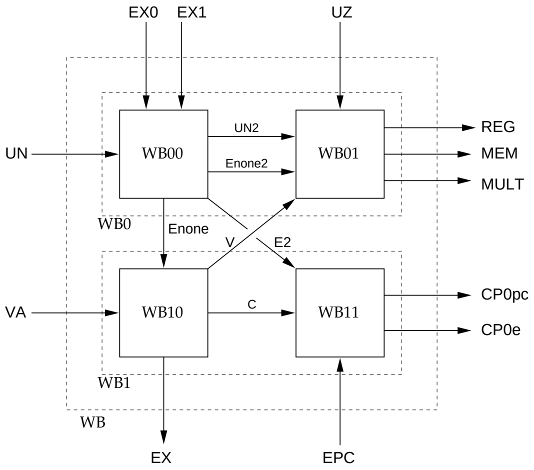

Figure 1. Decomposed version of the writeback.

$$
\begin{array}{l}
WB1 \equiv\\
\qquad valid\uparrow;\\
\quad *[\ \ V!valid; VA?va; EPC?epc; Enone?en; E2?e2;\\
\qquad [valid \wedge en \longrightarrow\\
\qquad\qquad\quad valid\downarrow, EX, CP0pc!epc, CP0e!e2\\
\qquad []\,va \longrightarrow valid\uparrow\\
\qquad []\,\textbf{else} \longrightarrow \textbf{skip}\\
\qquad]\\
\quad]
\end{array}
$$

Since epc is a 32-bit quantity, we decompose it out into a separate process. We apply the *control duplication* steps in order to decompose the selection statement in $WB1$. We name the newly introduced channel C, and the new variable introduced gd.
Applying projection once again, we obtain:

$$
\begin{array}{l}
WB10 \equiv\\
\qquad valid\uparrow;\\
\quad *[\ \ V!valid; VA?va; Enone?en;\\
\qquad [valid \wedge en \longrightarrow valid\downarrow, EX, C!0\\
\qquad []\,va \longrightarrow valid\uparrow, C!1\\
\qquad []\,\textbf{else} \longrightarrow C!2\\
\qquad]\\
\quad]\\
WB11 \equiv\\
\quad *[\ \ EPC?epc; E2?e2; C?gd;\\
\qquad [gd = 0 \longrightarrow CP0pc!epc, CP0e!e2\\
\qquad []\,\textbf{else} \longrightarrow \textbf{skip}\\
\qquad]\\
\quad]
\end{array}
$$

wb1 We now examine process $WB0$. We separate the process into two parts: one that computes the value of e, and the rest that uses the value of e. The part that computes the value of e depends on the value un, received on channel UN. The result of this projection is:

$$
\begin{array}{l}
WB00 \equiv \\
\quad *[\ UN?un; \\
\qquad [un = 0 \longrightarrow EX0?e \\
\qquad \| un = 1 \longrightarrow EX1?e \\
\qquad \| un = 2 \longrightarrow e := \textbf{none} \\
\qquad]; \\
\qquad Enone!(e \neq \textbf{none}), E2!e, \\
\qquad Enone2!(e = \textbf{none}), UN2!un \\
\quad]
\end{array}
$$

$$
\begin{array}{l}
WB01 \equiv \\
\quad *[\ V?valid2; UN?uz; Enone2?en2; UN2?un2; \\
\qquad [uz = 1 \longrightarrow REG!(valid2 \wedge en2) \\
\qquad \| uz = 0 \wedge un2 = 1 \longrightarrow MEM!(valid2 \wedge en2) \\
\qquad \| uz = 0 \wedge un2 = 2 \longrightarrow MULT!(valid2 \wedge en2) \\
\qquad] \\
\quad]
\end{array}
$$

By the projection theorem, the concurrent composition of $WB00$, $WB01$, $WB10$ and $WB11$ implement the original WB process. Figure 1 shows the final process decomposition, with the channels visible to the rest of the system as well as the internal channels introduced for the purposes of projection.

The final decomposition of the writeback is superior to the original program WB because each process in the final decomposition performs fewer actions in sequence, reducing the overall cycle time of the system. In addition, by further adjusting the slack on the different channels in the system (in particular, adding slack to channels $UN2$, $Enone2$, and $E2$) we can ensure that the pipelined implementation of the writeback operates at peak throughput (cf. [1, 10]).

7. Related Work

Process decomposition is a transformation that is purely syntactic, yet preserves the semantics of the computation [6]. However, the transformation does not increase the amount of concurrency in the system—a key property of projection.

8. Conclusions

We presented a synthesis technique for the design of finely pipelined asynchronous systems. The technique was based on the concept of projecting a CHP program onto different variables used in the text of the program. We provided conditions under which the transformation could be applied, which relied on certain processes being locally slack elastic. We showed how the transformations were used to construct a pipelined implementation of the write-back process used in the design of an asynchronous MIPS microprocessor.

References

[1] U. Cummings, A. Lines, and A. Martin. An asynchronous pipelined lattice structure filter. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 126–133, Nov. 1994.

[2] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[3] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[4] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice-Hall, 1992.

[5] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Fourth International Conference on the Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer-Verlag, July 1998.

[6] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[7] A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus. The design of an asynchronous microprocessor. In C. L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[8] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, U. V. Cummings, and T.-K. Lee. The design of an asynchronous MIPS R3000. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[9] J. L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[10] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.

A. Notation

The notation we use is based on Hoare's CSP [3]. What follows is a short and informal description of the notation we use.

Simple statements and expressions.

- Skip: **skip**. This statement does nothing.

- Assignment: $x := E$. This statement means "assign the value of E to x." When E is **true**, we abbreviate $x := E$ to $x\uparrow$, and when E is **false** we abbreviate $x := E$ to $x\downarrow$.

- Communication: $X!e$ means send the value of e over channel X; $Y?x$ means receive a value over channel Y and store it in variable x. When we are not communicating data values over a channel, the directionality of the channel is unimportant. In this case, the statement X denotes a synchronization action on port X.
- Probe: The boolean $\overline{X}$ is true if and only if a communication over channel X can complete without suspending.

Compound statements.

- Selection: $[G1 \rightarrow S1 \;[]\; ... \;[]\; Gn \rightarrow Sn]$, where G_i's are boolean expressions (guards) and S_i's are program parts. The execution of this command corresponds to waiting until one of the guards is true, and then executing one of the statements with a true guard. The notation $[G]$ is short-hand for $[G \rightarrow \textbf{skip}]$, and denotes waiting for the predicate G to become true. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Repetition: $*[G1 \rightarrow S1 \;[]\; ... \;[]\; Gn \rightarrow Sn]$. The execution of this command corresponds to choosing one of the true guards and executing the corresponding statement, repeating this until all guards evaluate to false. The notation $*[S]$ is short-hand for $*[\textbf{true} \rightarrow S]$. If the guards are not mutually exclusive, we use the vertical bar "|" instead of "[]."
- Sequential Composition: $S; T$. The semicolon binds tighter than the parallel composition operator $\|$, but weaker than the comma or bullet.
- Parallel Composition: $S \;\|\; T$ or S, T. The $\|$ operator binds weaker than the bullet or semicolon. The comma binds tighter than the semicolon but weaker than the bullet.
- Simultaneous Composition: $S \bullet T$ (read "S bullet T") means that the actions S and T complete simultaneously. Typically, the two actions are communication actions only, and the implementation of the bullet corresponds to replacing S by $S; S$ and T by $T; T$ and then picking an interleaving of the "doubled" actions, like $S; T; S; T$. The operator binds tighter than the semicolon and parallel composition.

The concurrent execution of a collection of CHP processes is assumed to be *weakly fair*—every continuously enabled action will be given a chance to execute eventually. The choice operator in the selection statement is assumed to be demonic, and therefore the choice is *not* fair. Consider the following four processes:

$$
\begin{array}{l}
\quad *[\; X!0 \;] \;\|\; *[\; Y!1 \;] \\
\| \; *[[\overline{X} \longrightarrow X?x \;[]\; \overline{Y} \longrightarrow Y?x \;];\; Z!x \;] \\
\| \; *[\; W!2 \;]
\end{array}
$$

Since the selection statement is not fair, Z is permitted to output an infinite sequence of zeros. However, both $Z!x$ and $W!2$ will execute eventually, since parallel composition is assumed to be weakly fair.

Set Comprehensions.

We use the notation shown below to describe a set of elements that satisfy some property.

$$\{x \mid R \triangleright T\}$$

x is an unordered list of variable names (*dummies*), R (the *range*) is a predicate, and T (the *term*) is an expression. The notation describes the set containing all terms T where x satisfies R.