Slack Elasticity in Concurrent Computing

Rajit Manohar and Alain J. Martin

California Institute of Technology, Pasadena CA 91125, USA

Abstract. We present conditions under which we can modify the slack of a channel in a distributed computation without changing its behavior. These results can be used to modify the degree of pipelining in an asynchronous system. The generality of the result shows the wide variety of pipelining alternatives presented to the designer of a concurrent system. We give examples of program transformations which can be used in the design of concurrent systems whose correctness depends on the conditions presented.

1 Introduction

In the design of an asynchronous clone of a MIPS R3000 microprocessor, we were faced with the problem of reasoning about a number of new program transformations that were introduced for performance reasons. The majority of the transformations corresponded to the introduction of pipelining in the processor [6]. In this paper, we provide general conditions under which we can pipeline a distributed computation.

We specify a distributed computation using CHP, a variant of CSP [2] (a brief description is contained in the appendix), and restrict our attention to systems that do not share variables among concurrent processes. 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 CHP specification of a process completely characterizes both the computation it performs as well as its synchronization behavior. For instance, we can specify a process that performs addition with the following CHP:

$$*[\ (A?x \| B?y);\ \ C!(x+y)\ ]$$

Unfortunately, for performance reasons, this specification can be very restrictive in practice. If $\mathbf{c}X$ is the number of completed actions on channel X, the specification includes the property that

$$0 \leq \mathbf{c}A - \mathbf{c}C \leq 1$$

In other words, the specification includes the fact that an implementation cannot accept its next set of inputs on channel A without producing an output on channel C. This restriction causes the throughput of an asynchronous delay-insensitive circuit that implements the computation to degrade as $1/\log N$, where N is the number of bits used to represent x. However, it is possible that

this property of the specification is not critical—namely, modifying it to the weaker

$$0 \leq \mathbf{c}A - \mathbf{c}C \leq \log N$$

does not affect the correctness of the computation. In that case, we can prevent the throughput degradation by pipelining the computation—a significant performance improvement.

It is often necessary to adjust the amount of pipelining in an asynchronous computation to optimize its performance based on the timing behavior of components of the system [9]. Ideally this should be a transformation applied after the high-level design is completed, since we may not have the necessary timing information until the physical design of the system has been simulated. Such transformations, in general, involve examining the entire asynchronous system instead of just a single process.

In this paper, we address the issues raised above by examining the following question: when can we change the slack of communication channels that are part of a system without modifying the behavior of the system? This single transformation can be used to show the correctness (or lack thereof) of a number of different program transformation techniques. Changing the slack of a synchronization channel is a non-trivial operation. Consider the following example in which channels A, X, and Y are slack-zero channels.

$$X; A \;\|\; A; Y \;\|\; [\; \overline{X} \longrightarrow X; Y;\ \text{“}good\text{”} \;[]\; \overline{Y} \longrightarrow Y; X;\ \text{“}bad\text{”} \;]$$

The only possible computation is the sequence $X; A; Y;$ “*good*”. However, if we introduce slack on channel A, we now have the possibility $A; Y; X;$ “*bad*”.

When we are permitted to add slack to a channel in the system, we say that the particular channel is *slack elastic*. If every channel in the system is slack elastic, the system is said to be slack elastic.

2 Model

We assume that the computation of interest is described by a collection of CHP programs communicating via first-in first-out channels. The programs do not share any variables; all interaction is via message-passing using single-sender single-receiver channels. Let X be a command causing an “X-action” when executed. We define $\mathbf{c}X$ to be the number of *completed* X-actions since the beginning of a computation.

2.1 Synchronization

(X,Y) form a pair of synchronization primitives if the difference $|\mathbf{c}X - \mathbf{c}Y|$ is bounded [4]. Formally, there exist two integer constants $\mathbf{k}X$ and $\mathbf{k}Y$ such that at least one of the two constants is finite, and:

$$-\mathbf{k}Y \leq \mathbf{c}X - \mathbf{c}Y \leq \mathbf{k}X \qquad \text{(SAFETY REQUIREMENT)}$$

The quantity $K = \mathbf{k}X + \mathbf{k}Y$ is called the *synchronization slack* [4].

The probe of a synchronization primitive can be used to determine if the action can complete [5]. Formally,

$$\overline{X} \Rightarrow (\mathbf{c}X - \mathbf{c}Y < \mathbf{k}X) \quad \wedge \quad (\mathbf{c}X - \mathbf{c}Y < \mathbf{k}X) \Rightarrow \Diamond\overline{X}$$
$$\overline{Y} \Rightarrow (-\mathbf{k}Y < \mathbf{c}X - \mathbf{c}Y) \quad \wedge \quad (-\mathbf{k}Y < \mathbf{c}X - \mathbf{c}Y) \Rightarrow \Diamond\overline{Y}$$

where $\Diamond E$ means that expression E becomes true eventually. Probes can only occur in the guards of selection statements.

The value $\mathbf{q}X$ is defined as the number of X-actions currently *suspended*. The progress requirement on synchronization primitives states that the set of suspended actions is minimal, i.e., the completion of any non-empty subset of suspended actions would violate the safety requirement [4]. Formally, if (X, Y) form a pair of synchronization primitives,

$$\mathbf{q}X = 0 \vee \mathbf{q}Y = 0 \qquad \text{(PROGRESS REQUIREMENT)}$$

CHP communication channels that carry data can be described using this framework. A CHP channel C has two *ports* associated with it: a sender $C!$, and a receiver $C?$. $(C!, C?)$ form a pair of synchronization primitives. We define $\mathbf{s}C!$ to be the sequence of data values that have been sent on the sender port, and $\mathbf{s}C?$ the sequence of received values. Let $|s|$ be the length of sequence s. Then, $|\mathbf{s}C!| = \mathbf{c}C!$ and $|\mathbf{s}C?| = \mathbf{c}C?$.

2.2 Computations and Behaviors

We restrict our attention to systems that satisfy the four properties listed below; their need will become evident in the sections that follow.

- the system is closed, i.e., we have specified the CHP processes of interest and their environment;
- the system is deadlock-free;
- negated probes of the sender port of channels are not used in the computation;
- if a sender port is probed, the probe will be true infinitely often.

An execution *trace* is a particular interleaving of atomic actions that can occur during execution of the system. The system is completely characterized by the set of possible traces that can occur [8]. We only consider the *complete* traces of the system [3]. The execution of processes is assumed to be weakly fair, and the selection statement is assumed to be unfair. (The appendix contains a more detailed description of the model.)

Given a concurrent system, we are not interested in the possible interleavings of actions that occur in a trace. Rather, we are interested in the sequence of data values that are sent on certain channels of the system, given the sequence of values being sent on other channels. For instance, in the example above, we might only be interested in the fact that the data values sent on channel C correspond to the sum of the values received on channels A and B. To this end, we define a *behavior* of a system in terms of the possible traces that can occur.

A behavior in our model is primarily characterized by the sequence of values that are sent and received on the channels of the system. Since processes in the

system can only interact using communication channels, behaviors capture the data values that are exchanged by interacting processes. Therefore behaviors can be used to describe the input/output characteristics of processes in the system. In addition, we would like to specify a computation without specifying the synchronization behavior as far as possible. In our model, the only ordering between values that have been sent on various channels that can be inferred from the behavior itself is the ordering preserved by the FIFO nature of the individual channels.

Since the sequences of values sent and received on channels can be infinite, behaviors capture the notion of weakly fair execution. The notion of weak fairness in traces corresponds to enabled actions in a process being executed eventually; the notion of weak fairness in behaviors corresponds to the next value (if any) that can be sent/received on a channel being sent/received eventually.

The other component of a behavior is the sequence of non-deterministic choices made by processes in the system, since these choices can affect the data values being sent on channels. The only construct in CHP that introduces such choices is the selection statement.

We assume that all the channels in the system are initialized empty, i.e., for all channels c, $\mathbf{k}c? = 0$. The assignment of initial values to variables and the initialization of channels is assumed to be part of the CHP program for each process. Therefore, the actual initial values of variables do not affect the behavior, because every variable is assigned a value initially (or the value the variable has initially is not used).

Given the sequence of choices made by a process and the sequence of values that have been received by the process, we can completely determine the local state of a process. Therefore, our model does not include the local state of the process as part of a behavior.

Definition 1 (decision point). *Given a trace, a decision point for a process p is a point between two actions in the trace where p has selected a guard of a selection statement for execution and several guards of the selection are true.*

A decision point is characterized by a tuple $(n, sel, gset, alt)$, where n is the occurrence index of the selection statement in the execution of p, sel denotes the selection statement, $gset$ is the set of guards of the selection statement that are true, and alt is the alternative chosen by p.

Decision points of the system correspond to places where a non-deterministic choice is made. We assume we have no control over the mechanism used to implement this choice; therefore, the choice made by the computation is assumed to be unfair.

Definition 2 (behavior). *Given a trace, the corresponding behavior $\mathcal{B}$ of a system is a function that maps each channel c in the system to pairs of sequences of values $(\mathbf{s}c?, \mathbf{s}c!)$ that occurred in the trace, and processes to their set of decision points in the trace.*

Given a channel c and process p, we denote $(\mathbf{s}c?, \mathbf{s}c!)$ by $\mathcal{B}.c$, and the set of decision points corresponding to p by $\mathcal{B}.p$. The behavior corresponding to a

trace is unique. However, multiple traces can map onto the same behavior, since different interleavings of actions that do not interact will be reduced to the same behavior if they do not affect the sequence of values sent on the channels in the system.

Definition 3 (system). *A system is a closed, deadlock-free collection of CHP processes and is defined by the set of behaviors that can occur during execution.*

Note that a system will be the empty set just when it does not contain any processes.

Example 1. The system

$$
\begin{array}{l}
\quad *[\ X!0\ ]\ \|\ *[\ Y!1\ ]\ \|\ *[\ Z?w\ ] \\
\|\ *[[\overline{X} \longrightarrow X?x;\ Z!x;\ \ [\overline{Y} \longrightarrow Y?x; Z!x\ \mid\ \neg\overline{Y} \longrightarrow \textbf{skip}] \\
\quad\ \mid \overline{Y} \longrightarrow Y?y;\ Z!y;\ \ [\overline{X} \longrightarrow X?y; Z!y\ \mid\ \neg\overline{X} \longrightarrow \textbf{skip}] \\
\quad]]
\end{array}
$$

has an execution that corresponds to the sequence $(X!0\|X?x);\ (Z!0\|Z?w);\ (Y!1\|Y?x);\ (Z!1\|Z?w)\ldots$ where the first guard $\overline{X} \rightarrow \ldots$ is chosen for execution with $\overline{Y}$ being true in the outer selection statement, and $\overline{Y} \rightarrow \ldots$ is chosen in the inner selection statement. The behavior corresponding to this trace maps Y to the pair of infinite sequences $([1,1,\ldots],[1,1,\ldots])$, X to $([0,0,\ldots],[0,0,\ldots])$, Z to $([0,1,0,1,\ldots],[0,1,0,1,\ldots])$, and the process with the selection statements to the set $\{(0, \mathit{selout}, \{\overline{X},\overline{Y}\},\overline{X}),\ (1, \mathit{selout}, \{\overline{X},\overline{Y}\},\overline{X}),\ldots\}$, where *selout* is the outer selection statement that selects between $\overline{X}$ and $\overline{Y}$, and the labels $\overline{X}$ and $\overline{Y}$ refer to the alternatives in the selection statement.

2.3 Specifications and Observability

The specification of a closed CHP program is a set of behaviors. Usually a specification does not completely specify the sequence of values sent and received on all channels of the system. Accordingly, we classify the channels of the system into *internal* and *external* channels, depending on whether or not the data values sent on those channels are part of the specification. All properties of interest must be specified only using the quantities $\mathbf{s}E!$ and $\mathbf{s}E?$, where E is an external channel.

Example 2. It is possible that we may not be able to observe certain properties of a computation, since behaviors do not contain as much information as the sequence of actions in the computation. For example, consider the following two processes:

$$
\begin{array}{l}
\quad *[\ NCS_1;\ \ CS_1\ ] \\
\|\ *[\ NCS_2;\ \ CS_2\ ]
\end{array}
$$

We cannot directly observe the property that two processes access their critical sections CS_i in an exclusive manner since we can only observe the sequence of values on channels. However, we can make the mutual exclusion property visible by the introduction of a third process and an external channel C as follows:

$$
\begin{array}{l}
\quad *[\ NCS_1;\ \ A!1; A!1; CS_1\ ] \\
\| *[\ NCS_2;\ \ B!2; B!2; CS_2\ ] \\
\| *[[\overline{A} \longrightarrow A?x\ [\!]\ \overline{B} \longrightarrow B?x];\ \ C!x\ ]
\end{array}
$$

By observing the sequence of values on channel C, we can determine if mutual exclusion is maintained. For instance, if sequence $1, 2, 1, 2, \ldots$ is possible, we have violated the mutual exclusion requirement.

Definition 4. *Given two sets of decision points D_1 and D_2 for a process p, we say that $D_1 \sqsubseteq D_2$ iff for every decision point $(n_1, sel_1, gset_1, alt_1) \in D_1$, there exists $(n_2, sel_2, gset_2, alt_2) \in D_2$ such that $n_1 = n_2$, $sel_1 = sel_2$, $gset_1 \subseteq gset_2$, and $alt_1 = alt_2$.*

The relation "$\sqsubseteq$" on sets of decision points orders them in terms of the number of non-deterministic choices that were possible.

Definition 5 (implementation). *We say that a system implements a specification if for each behavior $\mathcal{B}_{sys}$ of the system, there exists a behavior $\mathcal{B}_{spec}$ in the specification such that the sequence of values on all external channels in $\mathcal{B}_{spec}$ is the same as in $\mathcal{B}_{sys}$, and $(\forall p :: \mathcal{B}_{sys}.p \sqsubseteq \mathcal{B}_{spec}.p)$.*

This implementation relation is different from the traditional implementation relations used in trace theory and other models of concurrent programming because it does not include the synchronization behavior of the computation.

Example 3. Consider the following two systems:

$$
\begin{array}{l}
S_0 \equiv\ *[\ X!0\ ]\ \|\ *[\ Y!0\ ]\ \|\ *[\ X?x\ ]\ \|\ *[\ Y?y\ ] \\
S_1 \equiv\ *[\ X!0;\ \ Y!0\ ]\ \|\ \textbf{skip}\ \|\ *[\ X?x\ ]\ \|\ *[\ Y?y\ ]
\end{array}
$$

The computations specified by S_0 and S_1 are indistinguishable under our model because the sequence of values sent and received on channels X and Y remain unchanged, and both systems have no decision points. Standard concurrency models will differentiate them because the communication on X and Y cannot be executed in parallel, and because of the additional bound $0 \leq \mathbf{c}X - \mathbf{c}Y \leq 1$ in system S_1.

We now present the theorems that enable a large number of transformations, including the introduction and elimination of pipelining, data-flow style process decomposition, and pipelined completion detection.

3 Main Theorems

Throughout this section we will use $\mathcal{S}$ to denote the set of possible behaviors of the system of interest, p to denote a process in the system, and c to denote a channel in the system.

Lemma 1 (monotonicity). *Let $\mathcal{S}^+$ be the system obtained from $\mathcal{S}$ by increasing the slack on a particular channel. Then $\mathcal{S} \subseteq \mathcal{S}^+$*

Proof: Consider any behavior of $\mathcal{S}$. This behavior corresponds to some execution of system $\mathcal{S}$. It suffices to show that this execution is possible in $\mathcal{S}^+$. Let c be the channel whose slack was increased from $\mathbf{k}c!$ to $\mathbf{k}c! + n$. By definition, computations from $\mathcal{S}$ satisfy $\mathbf{c}c! - \mathbf{c}c? \leq \mathbf{k}c!$. These computations still exist in $\mathcal{S}^+$ because we can postpone any attempted send action on c so that this condition is satisfied, since $\mathcal{S}$ is deadlock-free. We now show that the communication actions that were attempted in $\mathcal{S}$ can also occur in $\mathcal{S}^+$.

The only construct in CHP which can affect control flow behavior is the selection statement. Increasing slack does not change the probe of the receiver end of the channel (by definition). The probe of a sender is monotonic with slack (by definition). Since we disallow negated probes of sender ports, this implies that all guards of selection statements are monotonic with slack. Also, a true probe on a sender port can be postponed (since probes only become true eventually) in $\mathcal{S}^+$ until the point when it becomes true in $\mathcal{S}$. Therefore, the guards true in $\mathcal{S}$ will eventually become true in $\mathcal{S}^+$, and so any behavior from $\mathcal{S}$ could occur in $\mathcal{S}^+$. □

Lemma 1 shows that the set of behaviors is monotonic with the slack on the channels. We now show that the only way in which increasing the slack on a channel can affect the computation is by increasing non-determinism. Note that both restrictions on computations that were mentioned in the previous section are needed for this proof.

Theorem 1 (decreasing slack). *Decreasing the slack of a channel does not affect the correctness of computations if and only if it does not introduce deadlock.*

Proof: Let $\mathcal{S}^-$ be the system obtained from $\mathcal{S}$ by decreasing the slack of a channel. If $\mathcal{S}^-$ is deadlock-free, $\mathcal{S}^- \subseteq \mathcal{S}$ by lemma 1. By definition 5, $\mathcal{S}^-$ implements $\mathcal{S}$. □

Definition 6 (extension). *A behavior $\mathcal{B}'$ is said to be the extension of behavior $\mathcal{B}$ iff:*

$$(\forall c :: \mathcal{B}.c = \mathcal{B}'.c) \wedge$$
$$(\exists p_0 :: (\forall p : p \neq p_0 : \mathcal{B}.p = \mathcal{B}'.p) \wedge \mathcal{B}.p_0 \neq \mathcal{B}'.p_0 \wedge \mathcal{B}.p_0 \sqsubseteq \mathcal{B}'.p_0)$$

Intuitively, the extension of a behavior corresponds to the same data behavior but with at least one additional choice which did not exist in the original behavior.

Theorem 2 (increasing slack). *Let $\mathcal{S}^+$ be the system obtained from $\mathcal{S}$ by increasing the slack of a channel. Then either $\mathcal{S} = \mathcal{S}^+$, or there exists a behavior $\mathcal{B}^+ \in (\mathcal{S}^+ - \mathcal{S})$ that is the extension of a behavior in $\mathcal{S}$.*

Proof: By lemma 1, $\mathcal{S} \subseteq \mathcal{S}^+$. Therefore either $\mathcal{S} = \mathcal{S}^+$, or there exists $\mathcal{B}_0 \in \mathcal{S}^+ - \mathcal{S}$. Assume such a $\mathcal{B}_0$ exists. Now $\mathcal{B}_0$ differs from every behavior in $\mathcal{S}$ in either the sequence of values sent on some channel or in the set of decision points

for some process in $\mathcal{S}$. This implies that the local state of some process from $\mathcal{S}^+$ differs from the the local state that could occur in $\mathcal{S}$. Consider the first point in execution when this occurs. The only non-deterministic construct in CHP is the selection statement, and therefore the only way a new local state could occur is because of a new true guard in a selection statement. By the same argument as in lemma 1, the guards true in $\mathcal{S}$ will eventually become true in $\mathcal{S}^+$. Therefore, we can pick an alternative of the selection statement that is possible in $\mathcal{S}$, and continue execution as in the original system $\mathcal{S}$. This new behavior is the required extension. □

The strength of Theorem 2 lies in the fact that if we can show that we cannot possibly introduce new decision points, this implies that adding slack does not change the behavior of a computation.

We now present some corollaries of the results of the previous section that can be used to reason about a large class of CHP programs.

4 Subsidiary Results

The monotonicity lemma coupled with Theorem 2 permits us to make the following statement that is very useful in practice.

Corollary 1 (sandwich theorem). *If a system satisfies its specification when the slack on channel c is k and when the slack on channel c is l ($> k$), it satisfies its specification when the slack on c is s, for all s satisfying $k \leq s \leq l$.*

Proof: The set of behaviors (and therefore the implementation relation) is monotonic with slack. Therefore, if the system is correct with c having slack k and slack l, the set of decision points is included on the set of those at slack l for all slack s satisfying $k \leq s \leq l$, concluding the proof. □

When computations are entirely deterministic, we can introduce slack on any channel without affecting correctness.

Corollary 2 (deterministic computations). *If the guards in selection statements are syntactically mutually exclusive and there are no probed channels, the system has only one behavior.*

Proof: Since the computation is deterministic, the sequence of values sent on channels is always the same and there are no decision points. □

A selection statement with probed channels in its guards is said to exhibit *maximal non-determinism* if all the guards can be true whenever the selection statement is executed.

Corollary 3 (maximal non-determinism). *If all selection statements with probes have maximal non-determinism, the system is slack elastic.*

Proof: The set of decision points of the system cannot be increased, so by Theorem 2 we can increase the slack on any channel without changing the behavior of the system. □

Corollary 3 is extremely useful in practice. The design of the MIPS R3000 processor undertaken by our group satisfies its requirements.

Consider the problem of measuring the slack of a channel c. To be able to measure the slack of c, we must be provided with a collection of processes to which c is connected, and a single channel which produces one output on channel *result*: *true*, if the slack of c is equal to a specified value, say k, or $false$ otherwise. We claim that this task is impossible under the assumptions of the model.

Corollary 4 (impossibility of measuring slack). *It is not possible to measure the slack of a communication channel.*

Proof: Assume that a collection of deadlock-free processes can be used to answer the question "is the slack of channel c equal to k?" Consider the closed system $\mathcal{S}$ where we observe channel *result*, and where c has slack k. The only possible output on *result* is *true*, by our assumption. Let $\mathcal{S}^+$ be the system, where we add slack 1 to channel c. By Theorem 1, $\mathcal{S}$ implements $\mathcal{S}^+$. Therefore, *result* can produce the value *true* in $\mathcal{S}^+$—a contradiction. □

More generally, if a system can be used to compute any relationship among the slacks of a set of channels, then the relation must be trivial—i.e., the system always outputs *true* or always outputs $false$.

5 Applications

When designing concurrent systems, we can increase the slack on a particular channel under the conditions outlined above. We now present some important transformations that can be shown to be semantics-preserving using the results derived above.

5.1 Pipelining

Pipelining is a technique whereby the computation of a function is distributed over a number of stages so as to reduce the cycle time of the system—increasing the throughput—at the cost of increasing the latency of the computation. A simple two-stage linear pipeline can be described by the following program:

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

We introduce pipelining when we transform program:

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

into the program shown above. It should be clear that we can apply this transformation if and only if we are permitted to increase the slack on channels L or R. Under those conditions, we can formally pipeline a computation as follows:

$$
\begin{aligned}
& *[L?x; R!g(f(x))] \\
= \quad & \{ \text{ add slack 1 to channel } R, \text{ introducing internal channel } I \ \} \\
& *[L?x; I!g(f(x))] \ \| \ *[I?y; R!y] \\
= \quad & \{ \text{ distribute computation—} I \text{ is internal } \} \\
& *[L?x; I!f(x)] \ \| \ *[I?t; R!g(t)]
\end{aligned}
$$

5.2 Eliminating Synchronization Among Actions

When designing a delay-insensitive system, we face a problem when attempting to design datapaths where the quantities being manipulated are constituted of a large number of bits. The problem is illustrated by examining the circuit implementation of the following program:

$$*[\ L?x;\ \ R!x\ ]$$

Before we send value x on channel R, we must be sure that all the bits used to represent x have been received on channel L. The circuit that waits for all the bits to have been received has a throughput that degrades as $1/\log N$, where N is the number of bits. As a result, as we increase the number of bits in x, the system throughput will decrease.

Instead, we examine an alternative implementation strategy. We implement channel L using an array of $\Theta(N)$ channels, where the individual channels use a fixed number of bits. As a result, we transform the program shown above into:

$$*[\ (\|i :: L[i]?x[i]);\ \ (\|i :: R[i]!x[i])\ ]$$

We have moved the performance problem from the implementation of the communication action on a channel to the implementation of the semicolon that separates the L and R actions. However, we observe that there is no data-dependency between channels $L[i]$ and $R[j]$ when $i \neq j$. We will attempt to remove the synchronization between the parts of the program that are not data-dependent.

We introduce a dummy process that enforces the sequencing specified by the program above. The original program is equivalent to:

$$
\begin{aligned}
& (\|i :: *[\ S[i] \bullet L[i]?x[i];\ \ S[i] \bullet R[i]!x[i]\ ]) \\
\| \ & *[\ (\|i :: S[i])\ ]
\end{aligned}
$$

since the slack zero $S[i]$-actions ensure that the actions on channels L and R are properly sequenced. (The bullet operator "$\bullet$" ensures that actions on $S[i]$ and $L[i]$ (or $R[i]$) are tightly synchronized.)

Now, we increase the slack on channels $S[i]$. If we let the slack on channels $S[i]$ go to infinity, the program shown above is equivalent to:

$$(\|i :: *[\ L[i]?x[i];\ \ R[i]!x[i]\ ])$$

Therefore, we can transform the original program into this one if and only if we can add slack on channels $S[i]$. Observe that we now have $\Theta(N)$ *independent* processes, and increasing N will not affect the throughput of the system. This transformation can be generalized to a technique which permits the distribution of a control value in a loosely synchronized manner [7].

5.3 General Function Decomposition

In general, if we have a computation graph which is supposed to implement a function that has a simple sequential specification, we can show its correctness by introducing "ghost channels" which sequence all the actions in the computation graph. A single process that sequences all the actions in the computation is introduced, so that the resulting system mimics the behavior of the sequential program. Adding slack to the ghost channels introduced for sequencing permits the processes in the computation graph to proceed in parallel; when we add infinite slack to the sequencing channels, we have a computation that behaves exactly like the original computation without the sequencer process, and the ghost channels can be deleted without modifying the behavior of the computation. Therefore, showing the correctness of the original computation can be reduced to showing whether adding slack on the ghost channels modifies the behavior of the system.

Example 4. Suppose we would like to demonstrate that the following CHP program implements a first-in first-out buffer:

$$*[\ L?x;\ \ U!x;\ \ L?x;\ \ D!x\ ]\ \|\ *[\ U?y;\ \ R!y;\ \ D?y;\ \ R!y\ ]$$

We begin by closing the system with the introduction of two processes which send data on channel L and receive data from channel R. Next, we introduce a sequencer process which sequences the actions in the computation. The resulting system is shown below.

$$\begin{array}{l} \quad i := 0; *[\ L!i;\ \ i := i+1\ ]\ \|\ *[\ R?w\ ] \\ \|\ *[\ L?x \bullet S_1;\ \ U!x \bullet S_2;\ \ L?x \bullet S_4;\ \ D!x \bullet S_5\ ] \\ \|\ *[\ U?y;\ \ R!y \bullet S_3;\ \ D?y;\ \ R!y \bullet S_6\ ] \\ \|\ *[\ S_1;\ \ S_2;\ \ S_3;\ \ S_4;\ \ S_5;\ \ S_6\ ] \end{array}$$

The sequencer process restricts the computation so that only one interleaving is possible, namely the sequence

$$(L!0\|L?x); (U!x\|U?y); (R!y\|R?w); (L!1\|L?x); (D!x\|D?y); (R!y\|R?w); (L!2\|L?x);\ \ldots$$

which clearly implements a first-in first-out buffer, since the sequence of values sent on R is the same as the sequence of values received on L. We can increase the slack on channels S_i without modifying its behavior because the computation is deterministic. In the limit of infinite slack on the channels S_i for all i, the

sequencer process does not enforce any synchronization between the actions, and we can eliminate the sequencer process entirely leaving us with the original computation. Therefore, the original computation implements a first-in first-out buffer.

5.4 A Recipe for Slack Elastic Programs

Corollary 3 can be used as a guideline for the design of programs that are guaranteed to be slack elastic. Ensuring slack elasticity of the design is important in order to be able to postpone decisions related to the amount of pipelining to be used in an implementation. In the design of an asynchronous MIPS processor, we found it necessary to adjust the slack on communication channels after most of the physical layout was complete because we did not have accurate estimates of the timing behavior of the processes we used until analog simulations were performed.

There are two selection statements in CHP. Selection statements that are described using the thick bar "[]" indicate that the guards are mutually exclusive. If such selection statements do not use any probes in their guards, they cannot be the cause of the introduction of new decision points. Selection statements that use the thin bar "|" indicate that their guards might not be mutually exclusive. If such selection statements are maximally non-deterministic—i.e., if the computation meets its specification irrespective of the alternative chosen when the selection is encountered, then they will not be the cause of erroneous computations. If we follow these two guidelines, we will be guaranteed that the computation is slack elastic. Every process in the high-level description of the asynchronous MIPS processor we designed satisfied these criteria.

6 Conclusion

We have presented a new technique for reasoning about the correctness of a concurrent system based on the concept of synchronization slack, and presented conditions under which the slack of a channel in a distributed computation can be modified without affecting its behavior.

We showed how a number of program transformations can be analyzed by considering the effect of changing the slack of a communication channel, demonstrating that slack elasticity is an important property for a computation to have.

We presented sufficient conditions under which a distributed computation is slack elastic. The conditions were strong enough to be satisfied by the complete high-level design of an asynchronous processor. Slack elasticity was an important tool that enabled us to reason about a number of complex transformations in the design of the processor.

References

1. van der Goot, M.: The Semantics of VLSI Synthesis. Ph.D. thesis, California Institute of Technology (1996)

2. Hoare, C.A.R.: Communicating Sequential Processes. *Communications of the ACM*, **21**(8) (1978) 666–677
3. van Horn, K.S.: *An Approach to Concurrent Semantics Using Complete Traces.* M.S. thesis, California Institute of Technology (1986)
4. Martin, A.J.: An Axiomatic definition of synchronization primitives. *Acta Informatica*, **16** (1981) 219–235
5. Martin, A.J.: The Probe: An addition to communication primitives. *Information Processing Letters*, **20** (1985) 125–130
6. Martin, A.J., Lines A., Manohar R., Nyström, M., Penzes, P., Southworth, R., Cummings, U.V., and Lee, T.K.: The design of an asynchronous MIPS R3000. *Proceedings of the 17th Conference on Advanced Research in VLSI* (1997)
7. Manohar, R.: The Impact of Asynchrony on Computer Architecture. Ph.D. thesis, California Institute of Technology (1998)
8. van de Snepscheut, J.L.A.: Trace theory and VLSI design. Lecture Notes in Computer Science 200, Springer-Verlag (1985)
9. Williams, T.E.: Self-timed Rings and their Application to Division. Ph.D. thesis, Computer Systems Laboratory, Stanford University (1991)

A Notation

The notation we use is based on Hoare's CSP [2]. What follows is a short and informal description of the notation we use. A formal semantics can be found in [1].

Simple statements and expressions.

- Skip: **skip**. This statement does nothing.
- Assignment: $x := E$. This statement means "assign the value of E to x."
- 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 port X can complete without suspending.

Compound statements.

- Selection: $[G_1 \rightarrow S_1 \,[]\, \ldots \,[]\, G_n \rightarrow S_n]$, 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: $*[G_1 \rightarrow S_1 \; [] \; ... \; [] \; G_n \rightarrow S_n]$. 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 $*[\mathbf{true} \rightarrow S]$. 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: $S \parallel T$ 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. The bullet synchronizes the two actions by enforcing $\mathbf{c}S = \mathbf{c}T$. Action $S \bullet T$ is implemented by decomposing actions S and T into smaller actions and interleaving them. 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 selection statement is assumed to be demonic, and it therefore *not* fair. Consider the following four processes:

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

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.