Reducing the Communication of Message-Passing Systems Synthesized from Synchronous Programs

Daniel Baudisch  
University of Kaiserslautern, Germany  
baudisch@cs.uni-kl.de

Yu Bai  
University of Kaiserslautern, Germany  
bai@cs.uni-kl.de

Klaus Schneider  
University of Kaiserslautern, Germany  
klaus.schneider@cs.uni-kl.de

Abstract—This paper presents a method to translate a given synchronous system to a multithreaded system where process nodes communicate via channels with each other. It is well-known that the reduction of communication has been identified to be a crucial key for efficient utilization of multiprocessor systems. For this reason, we first use synchronous elastic design methods to generate a distributed/multithreaded system from a synchronous system, and then, reduce communication overhead between the obtained process nodes. Our benchmarks show that we can save up to 67.5% of communication costs using our method and can achieve an average speed-up of up to 1.09.

I. INTRODUCTION

The synchronous model of computation (MoC) has many advantages, in particular, for the specification and verification of reactive systems. The logical time and implicit synchronization allow programmers to focus on the functional correctness of their algorithm instead of dealing with architecture-specific timing/synchronization problems. Moreover, the synchronous MoC provides implicit parallelism and synchronization, which is a promising feature for the steadily growing community of parallel software developers. In contrast, programming parallel software in imperative languages, e.g., C or Java, often forces programmers to take care of explicit communication and synchronization of parallel processes. Hence, besides hardware as a target for translating synchronous programs, software becomes a target of interest, too. The development of synthesis tools for synchronous languages challenges researchers to map these programs to parallel software, e.g., for shared memory or message-passing architectures.

Automatic distribution of synchronous programs has already been considered in [10, 15, 21]. In [21] DSystemJ, they present a formal MoC to create dynamic distributed systems. The primary goal of these approaches is the distribution of synchronous programs, but does not include an optimization of the communication, which is in the focus of this paper.

The necessity for desynchronization of synchronous programs originally comes from hardware design. Handling of clock skews and clock distribution in large circuits became a major problem in recent years. Solutions have been presented by latency insensitive designs in [7–9] and as synchronous elastic designs [19]. These ideas have been applied in [2] to create multithreaded code from synchronous program descriptions. Therefore, a synchronous program is partitioned into several processes, which communicate with each other through FIFO buffers (or in other words channels) as suggested in [19]. This results in a (Kahn) data-flow process network (DPN) [17], which is a generic representation of a decoupled distributed system description where process nodes run independent of each other. DPNs are easily mapped to message-passing software: Each node of a DPN is thereby translated to a separate thread. As already mentioned, communication between threads is done using any FIFO buffer implementation like the concurrent bounded queue from the Intel® TBB library. However, the synchronous MoC is still reflected in the communication scheme: when a node fires, it reads one token from each input buffer and writes exactly one token to each output buffer. This raises two points of criticism:

1) Computed values are sent to other nodes independent of whether the sent value is actually required by the target node, which leads to a potential unnecessary communication overhead.

2) A node always waits for all input values, no matter whether all of them are actually required for the next calculation.

The main idea to tackle the above criticisms in this paper is to transport a data value only when it is really needed for the computation performed in the target node. This is achieved by defining read and write conditions for variables that are transported between nodes. A variable is transported only when its value changes (fulfilling the write condition) and is needed for the computation (fulfilling the read condition). The read and write conditions are evaluated both at compile time and runtime for a maximum precision and are further optimized to balance between communication and computation. The analysis is based on a quite general intermediate format (guarded actions) and is therefore independent of particular programming languages used for implementation.

The rest of this paper is organized as follows: Section II discusses and compares related methods with ours. Section III introduces synchronous guarded actions, which are a generic format to represent synchronous programs. The partitioning of these systems results in synchronous DPNs, which will be explained in Section III-A. The algorithm for removing communication in synchronous DPNs is explained in Section IV and a refinement for further optimization is sketched in Section V. Section VI presents our benchmark results, and Section VII will conclude the paper and its results.

II. RELATED WORK

Different ways to deal with the two criticisms raised in the previous section have been proposed in the past. In particular, the second problem has been addressed in early evaluation concepts [29] where an evaluation is started as soon as
sufficiently many data values are available. A practical example for early evaluation is token counterflow and related variants [11, 12, 28]. One issue in early evaluation is to assign actual tokens to the correct calculation. Counterflow tokens are used to eliminate tokens that have been identified to be useless for calculations. However, this technique does not consider the reduction of the communication overhead in a system.

The idea of demand-driven evaluation as proposed in [22, 23] is to compute values only if necessary. In particular, data should only be generated if other calculations depend on it (similar to lazy evaluation in functional programming languages). This approach clearly targets the reduction of computation instead of reduction of communication.

Communication reduction is also considered by endo-isochronous systems. In addition to general desynchronized systems, a variable is allowed to be absent in a macro step, i.e., it has no value at that point of time. Endochrony characterizes a node in a desynchronized program by its ability to derive the presence and absence of all variables from the communicated ones. Two endochronous nodes are called isochronous if the nodes agree on the presence of variables at all points of time, i.e., each variable is always either present or absent in both nodes (which means that the two nodes assign the variable the same clock). [24–27] provides theoretical work to analyze whether a system fulfills endo-isochony.

A method to desynchronize synchronous programs by modes is presented in [6] which requires a global mode manager. The global mode manager has to distribute the modes at least to a part of the nodes. Since this method has not been evaluated, it is not clear, if the global mode manager may become a bottleneck. Our approach manages communication reduction purely on existing communication without adding a centralized manager. Despite of the rich theory in this area, no implementation is yet known to us.

III. SYNCHRONOUS GUARDED ACTIONS

Synchronous programs as written in Esterel [5], Quartz [27] or Lustre [16] rely on the synchronous model of computation (MoC) [4, 16]. Reactions of a system are computed in logical steps that are called macro steps. In each macro step, the reaction is computed by a set of actions that are called micro steps. In each macro step, all micro steps are evaluated with the same variable environment and are therefore virtually executed in zero time.

In this paper, we usually refer to the variable environment of a specific macro step, but also have to refer to values of variables of the previous and next macro step. Therefore, we define the operators $\times$ and $\bar{\times}$:

**Definition 1 (Next and Previous Operator):** We define $\times$ as the next operator and $\bar{\times}$ as the previous operator: $\times x$ in macro step $i$ refers to the value of $x$ in macro step $i + 1$ and $\bar{\times} x$ in macro step $i$ refers to the value of $x$ in macro step $i - 1$. $\times x$ in macro step 0 is defined as the default value of $x$, e.g., false if $x$ is Boolean and 0 if $x$ is an integer.

Synchronous guarded actions are a general representation of synchronous programs. A single guarded action is formally given as $\langle \gamma \Rightarrow A \rangle$, where $\gamma$ is a boolean expression that triggers the atomic action $A$. Typically, the action $A$ is an assignment or a function call. Even though our approach is not limited to assignments, we will subsequently focus on this type of action for the sake of simplicity. We thereby consider two kinds of assignments: immediate assignments of type $x = \tau$ and delayed assignments of type $x \times = \tau$. Both evaluate the right-hand side of the assignment in the current macro step. An immediate assignment stores the result in the left-hand side in the same macro step, while the delayed assignment does this at the beginning of the next macro step. The functions given in Definition 2 determine the variables that are read and written by a guarded action. They are relevant for the analysis of actually written and read values in a data-flow process network, which is considered in Section III-A.

**Definition 2 (Read and Written Variables):** Let the free variables occurring in the expression $\tau$ be denoted as $\operatorname{Vars}(\tau)$. The variables read and written by a synchronous guarded action are defined as follows:

$$\begin{align*}
\operatorname{rdVars}(\gamma \Rightarrow x = \tau) & := \operatorname{Vars}(\gamma) \cup \operatorname{Vars}(\tau) \\
\operatorname{wrVars}(\gamma \Rightarrow x = \tau) & := x
\end{align*}$$

Synchronous programs require the value of a variable to be uniquely determined in each macro step. In particular, if there is no action that determines the value of a variable $x$ in a macro step, a default value is assigned, which ensures a deterministic behavior. In the following, we refer to this reaction as the default action, because it is not explicitly coded. The behavior of a default action depends on the storage type of a variable, which reflects the ability of the system to memorize the value of a variable or to use a default value instead: A memorized variable keeps its value from the previous macro step by default, while an event variable is reset to a default value (like 0 or false). The default action plays an important role in this paper, because it is the key to reduce communication costs.

A. Synchronous Data-Flow

Data-flow process networks (DPNs) have been proposed as a graph-based model of computation for parallel and distributed computing in [13, 17, 18]. A DPN consists of a fixed set of nodes that are connected by unbounded FIFO buffers, thereby decoupling the execution of the nodes to let them run in parallel. Each node represents a function that maps the inputs coming from its input FIFO buffers to output values that are put into its output FIFO buffers. The function is activated by the availability of data, i.e., DPNs are purely data-driven and do not require a global control like a clock.

In general DPNs, the number of tokens that are read and written by a node may vary per reaction. However, this flexibility comes with an additional effort in the analysis of DPNs: Checking for liveness, absence of deadlocks [31], and boundedness of buffers is undecidable, but necessary especially for safety-critical systems. A precise analysis has to take the behavior of nodes into account, because it controls the number of tokens that are read and written per node.

The analysis is much easier for synchronous data-flow process networks (SDFG) [14, 20]. SDFGs represent a subset...
of DPNs, characterized by the number of tokens that are read from and written to buffers that is the same for each reaction of a node. In our synthesis process, we assume that we have a SDFG \( D = (N, B) \) consisting of a set of nodes \( N \) and buffers \( B \subseteq N \times N \). Each node \( N \in N \) is connected to other nodes via buffers \( B \in B \). The behavior of each node \( N \in N \) is described by a set of guarded actions \( A(N) \). Analogous to Definition 2, we define \( \text{wrVars}(N) := \{ \text{wrVars}(A) \mid A \in A(N) \} \) as the set of variables written by node \( N \) and \( \text{rdVars}(N) := \{ \text{rdVars}(A) \mid A \in A(N) \} \) as the variables that are read by node \( N \).

We can then determine the nodes writing or reading to a variable \( x \) as \( N_{\text{wr}}(x) := \{ N \mid N \in N \land x \in \text{wrVars}(N) \} \) and \( N_{\text{rd}}(x) := \{ N \mid N \in N \land x \in \text{rdVars}(N) \} \), respectively. By definition of DPNs [17], a buffer is always written by exactly one node, i.e., the writing node of a variable is uniquely determined. Formally, \( \text{wrVars}(N_1) \cap \text{wrVars}(N_2) = \{ \} \) for all \( N_1, N_2 \in N, N_1 \neq N_2 \).

In the following, we use the term \textit{local variables} to describe internal states of DPN nodes. Internal states can be implemented by adding buffers that are read and written by the same node starting with an initial token.

IV. REDUCING COMMUNICATION

A. Main Idea

Data values that are communicated through FIFO buffers are not always needed by the consumer nodes. This observation is the key for the reduction of the communication costs that are described in the following. The idea is thereby to send a value from one node to another only if it is needed for the calculations of the target node. The actual necessity of a value for the program’s outputs (which is actually of interest) is impossible or at least hard to ascertain, because these values may also influence the behavior in the future, which can neither be detected at compile time nor at runtime without extensive effort. Nevertheless, we are able to ascertain whether a variable influences the calculations that directly depend on that variable.

In the following, we use the term \textit{transport}, formally \( \zeta(x) \), as the description of the communication of a single value of variable \( x \) between two nodes, the writing node (\textit{sender}), formally \( \zeta_{\text{wr}}(x) \), and a reader node (\textit{receiver}), formally \( \zeta_{\text{rd}}(x) \). The respective antagonists of a transport will be called \textit{partner node}, i.e., the partner node of the sender is the receiver and vice versa. To this end, let \( T \) be the set of all transports in a given DPN.

Transports can be saved by considering changes of variable values. In particular, the default action, which defines the default value of a variable in a macro step, is a quite trivial action that either assigns a constant or just keeps the value from the previous macro step. Hence, executing the default action on the local copy of the consumer node, instead of invoking a transport, will have the desired effect. Hence, we require each node to have a local copy for each variable that is written by another node. This local copy will either contain the current value or the symbol \( \perp \) to signal that the local copy is outdated. In the following, \textit{Bubble} will be the term for a value of a transport that is an unnecessary value or the result of the default action.

In the first part of our optimization, we assume that our optimization is based on a \textit{per transport} analysis, i.e., for each transport \( \zeta(x) \in T \).

B. True Writes and True Reads

We assume that the nodes of the DPN contain guarded actions, i.e., conditional assignments. Hence, we can determine when a variable is modified and when it is actually read by a node. If a value of a variable is changed by its writer (read by its reading node), it is referred to as \textit{true write (true read)}.

\textbf{Definition 3}: The condition for a true write to \( x \) is the disjunction of all guards of actions writing to variable \( x \):

\[ \gamma_{\text{wr}}(x) := \bigvee \{ \gamma \mid \exists A. A \in A(N_{\text{wr}}(x)) \land x \in \text{wrVars}(A) \land A = \langle \gamma \Rightarrow A \rangle \} \]

If \( x \) occurs in a guard of a guarded action of \( A(N) \), then the variable \( x \) is read by \( N \), and otherwise a true read of \( x \) is the disjunction of all guards of actions reading variable \( x \) (in the action):

\[ \gamma_{\text{rd}}(x, N) := \begin{cases} \text{true} : & \exists A. A \in A(N) \land x \in \text{rdVars}(A) \land A = \langle \gamma \Rightarrow A \rangle \land \end{cases} \]

C. Transport Conditions for Immediately Written Event Variables

The condition whether a value for variable \( x \) has to be transported depends on the type of \( x \) and the kind of assignment (immediate or delayed). We start with an explanation when a value for an immediately written variable \( x \) has to be transported:

1) \( \gamma_{\text{rd}}(x, N) \land \gamma_{\text{wr}}(x) \), i.e., \( x \) is read by node \( N \) and written by some action. Hence, the written value must be transported from the sender to the receiver (node \( N \)).

2) \( \neg \gamma_{\text{rd}}(x, N) \land \gamma_{\text{wr}}(x) \), i.e., node \( N \) does not read \( x \) and it is currently written. Hence, the value is not sent to \( N \), since it is not needed by \( N \). The local copy of \( x \) in the receiver \( N \) has then value \( \perp \).

3) \( \gamma_{\text{rd}}(x, N) \land \neg \gamma_{\text{wr}}(x) \), i.e., node \( N \) reads \( x \), and \( x \) is currently not written. Hence, the sender of \( x \) applies the default action on \( x \), and \( N \) applies the default action on its local copy of \( x \), so that there is no need to communicate that value to \( N \).

4) \( \neg \gamma_{\text{rd}}(x, N) \land \neg \gamma_{\text{wr}}(x) \), i.e., node \( N \) does not read \( x \), and \( x \) is also not written by an action. Hence, \( x \) is not involved into any calculations. Hence, \( N \) applies the default action on its local copy of \( x \) (which is what the sender of \( x \) also does without communicating that value).

Hence, only in the first case, we have to communicate the value of \( x \) from the sender of \( x \) to the readers.

D. Transport Conditions for Immediately Written Memorized Variables

In case that \( x \) is an immediately written memorized variable, the handling of communication is more sophisticated. First of all, the writer of \( x \) must keep track whether the local copy of reader \( N \) is up-to-date:
The next task is to handle delayed assignments, where the timing has to be considered for the overall write guard in the written variables, we can use the case distinctions above to hold. Definition 4 allows us to precisely define the transport behavior for immediately written memorized variables:

1) \( \gamma_{rd}(x, N) \land \gamma_{wr}(x) \), i.e., \( x \) is read by node \( N \) and written by some action. Hence, the written value must be transported from the sender to the receiver (node \( N \)). In addition, \( x^N \) must be set, since the receiver has an up-to-date value.

2) \( \neg \gamma_{rd}(x, N) \land \gamma_{wr}(x) \), i.e., node \( N \) does not read \( x \) and it is currently written. Hence, the value is not sent, the local copy of \( x \) of receiver \( N \) is then \( \perp \), and \( x^N \) must be unset to keep track of the invalid copy in the receiver.

3) \( \gamma_{rd}(x, N) \land \neg \gamma_{wr}(x) \), i.e., node \( N \) reads \( x \), and \( x \) is currently not written. The behavior of the nodes depends on the flag \( x^N \), i.e., whether the local copy is up-to-date. If \( x^N \) holds, the reader must execute the default action to keep it local copy up-to-date, otherwise the sender must send the current value of \( x \) to the receiver. In both cases, the value of \( x \) will be updated in \( N \), and hence, the flag \( x^N \) must be set. Finally, the condition whether to transport a value is \( \gamma_{rd}(x, N) \land \neg \gamma_{wr}(x) \land \neg x^N \).

4) \( \neg \gamma_{rd}(x, N) \land \neg \gamma_{wr}(x) \), i.e., node \( N \) does not read \( x \), and \( x \) is also not written by an action. Hence, if \( x^N \) holds, the reader must execute the default action to keep its local copy up-to-date. Otherwise \( x^N \) is unset) the local copy remains invalid. This behavior guarantees that transports are avoided as long as the variable is not changed by the sender. To this end, the local copy of \( x \) and the flag \( x^N \) will simply keep their values.

E. Transport Conditions for Delayed Written Variables

The next task is to handle delayed assignments, where the reads will take place only in the macro step following the write. Therefore, we have to consider the values of \( \gamma_{rd}(x, N) \) and \( \gamma_{wr}(x) \) of different macro steps, i.e., the sender has to evaluate \( X \gamma_{rd}(x, N) \). Hence, the values that are necessary to evaluate the condition would have to be read ahead by the receiver. This somehow corresponds to ‘reading into the future’. Reading values that eventually will not be available, yet, will technically end up with a deadlock of the DPN. Hence, the sender must assume the worst case (in the sense of communication overhead): the receiver will read the value of \( x \). To this end, if we set \( \gamma_{rd}(x, N) \) to true for all delayed written variables, we can use the case distinctions above to check whether a value has to be sent or not. The different timing has to be considered for the overall write guard in the receiver, too. In particular, to determine whether to read \( x \), a receiver must evaluate \( \gamma_{wr}(x) \) in the preceding iteration. Since all variables to evaluate this expression may already have been changed, we simply add a carrier variable \( \gamma_{wr}(x) \) and the guarded action \( \text{true} \Rightarrow \gamma_{wr}(x) \) to the behavior of the receiver.

Because the overall guard for reading a delayed variable at the receiver must be assumed to be true, the transport of these variables provide the least potential for communication reduction. The condition for transporting a value for a macro step depends only on the overall write guard. As a result, all value changes of delayed written variables are committed to the receiver, i.e., the receiver’s copy of the variable will be always up-to-date. If the variable of the sender does not change, the reaction to absence is applied to the variable in the sender and to the copy of each receiver. Hence, there is no need to keep track of whether the local copy of a receiver is up-to-date as done for immediately written memorized variables (see \( x^N \) in case 4 of the behavior definition for immediately written memorized variables).

F. Summarizing Transport Conditions

To summarize the above discussions, we can now precisely determine the conditions, when the writer of a variable \( x \) has to send (push) the value to a reader \( N \) through the corresponding buffer, and when reader \( N \) of that variable has to read (pop) the value from the corresponding buffer:

- **If** \( x \) is an immediately written event variable:
  \( \gamma_{push}(x, N) = \gamma_{pop}(x, N) = \gamma_{rd}(x, N) \land \gamma_{wr}(x) \)

- **If** \( x \) is an immediately written memorized variable:
  \( \gamma_{push}(x, N) = \gamma_{rd}(x, N) \land (\gamma_{wr}(x) \lor \gamma_{wr}(x) \land \neg x^N) \)

- **If** \( x \) is a delayed written event variable:
  \( \gamma_{push}(x, N) = \gamma_{wr}(x) \land \gamma_{pop}(x, N) = \neg \gamma_{wr}(x) \)

- **If** \( x \) is a delayed written memorized variable:
  \( \gamma_{push}(x, N) = \gamma_{wr}(x) \land \gamma_{pop}(x, N) = \neg \gamma_{wr}(x) \land \neg x^N \)

To handle immediately written event variables, it is sufficient to modify the conditions for executing a push and pop. Whereas, memorized and/or delayed written variables need to change the behavior of the sender and receiver, i.e., to add additional guarded actions. The modifications in concrete terms depend on the specific case as shown in the pseudo-algorithm in Figure 1.

G. Quantification

In the following, we assume that a variable is known in a node, if the node reads or writes that variable. Conversely, if a variable is neither written nor read by a node, it is stated as unknown in that node. This is based on the fact that nodes in a DPN will only receive data that is necessary to compute their own behavior. In addition, each node is supposed to be capable of reading variables that are written in the same macro step by itself.

Until now, we assumed that all variables that appear in the expressions \( \gamma_{wr}(x) \) and \( \gamma_{rd}(x, N) \) are known in both the sender
and receiver. In general, a node will not have knowledge of all variables of the overall system, i.e., some variables might be unknown in a node. As mentioned, a node only knows variables that are either necessary for the computation of its behavior or that are produced by the node itself. Hence, the sender and the receiver may not be able to evaluate \( \gamma_{rd}(x, N) \) and \( \gamma_{wr}(x) \), respectively, without adding communication. In conclusion, the presented approach at that point is only applicable in case that all variables in both expressions are known by both nodes. To extend this approach, this section introduces simplified valid evaluable guards.

Let \( \mathcal{V} \) be the set of all variables in the DPN including inputs from and outputs to the environment. Furthermore, let \( \text{VAR}(\tau) \subseteq \mathcal{V} \) denote all variables of the formula \( \tau \). In addition, \( \text{VAR}(N) \) denotes all variables that are known by node \( N \).

Obviously, we must preserve sending of all non-bubble values. This leads to the following constraints of a simplified guard to ensure that values can be evaluated: 1) A simplified guard \( \gamma^3 \) that simplifies \( \gamma \) is valid iff \( \gamma \rightarrow \gamma^3 \). 2) A simplified guard \( \gamma^3 \) that simplifies \( \gamma \) is evaluable by node \( N \) iff \( \text{VAR}(\gamma^3) \subseteq \text{VAR}(N) \). The simplification is transport-specific, i.e., the simplified overall write guard gets individually adapted to each reading node \( N \). Hence, \( N \) becomes a parameter in \( \gamma^3_{wr}(x, N) \).

The requirement for validity leads to a pessimistic overestimation of guards, i.e., if \( \gamma \) cannot be evaluated due to missing variables, each node must assume a read/write by its partner node. For a transport of \( x \) between its sender and its receiver \( N \), this is formally expressed by the following conditions: \( \gamma_{rd}(x, N) \rightarrow \gamma^3_{rd}(x, N) \) and \( \gamma_{wr}(x) \rightarrow \gamma^3_{wr}(x, N) \). The evalubility of a simplified guard \( \gamma^3 \) means that a node knows all variables that appear in \( \gamma^3 \). An existential quantification over the unknown variables in \( \gamma_{rd}(x, N) \) and \( \gamma_{wr}(x) \) will result in the desired simplified guards.

Let \( \zeta(x) \) be the transport of variable \( x \) from sender \( \zeta_{wr}(x) \) to a receiver \( \zeta_{rd}(x) \). In addition, let \( \gamma_{wr}(x) \) be the guard for a write of \( x \) in \( \zeta_{wr}(x) \), and let \( \gamma_{rd}(x, \zeta_{rd}(x)) \) be the guard for a read of \( x \) in \( \zeta_{rd}(x) \). Then, \( \mathcal{V}_1 = \text{VAR}(\zeta_{wr}(x)) \setminus \text{VAR}(\zeta_{rd}(x)) \) is the set of variables that are known in the sender but not in the receiver, and \( \mathcal{V}_2 = \text{VAR}(\zeta_{rd}(x)) \setminus \text{VAR}(\zeta_{wr}(x)) \) denotes the set of variables that are known in the receiver but not in the sender. We can then define the simplified guards as follows:

\[
\begin{align*}
\gamma^3_{rd}(x, \zeta_{rd}(x)) &= \exists \nu \in \mathcal{V}_1 \nu \cdot \gamma_{rd}(x, \zeta_{rd}(x)) \\
\gamma^3_{wr}(x, \zeta_{wr}(x)) &= \exists \nu \in \mathcal{V}_2 \nu \cdot \gamma_{wr}(x)
\end{align*}
\]

Then, one can see why \( \gamma^3_{rd}(x, \zeta_{rd}(x)) \) = true must always hold for delayed written variables: to obtain a correct temporal access to the variables in the sender \( \zeta_{wr}(x) \), all variables \( \nu \) in \( \gamma^3_{rd}(x, \zeta_{rd}(x)) \) must be replaced by \( \mathcal{X}_\nu \). All variables for a macro step in the future are unknown. Applying the existential quantifier to \( \gamma_{rd}(x, \zeta_{rd}(x)) \) for all unknown variables will result then in true, because all variables occurring in that expression are unknown.

As a result of the definition of \( \gamma^3_{rd}(x, \zeta_{rd}(x)) \) and \( \gamma^3_{wr}(x, \zeta_{wr}(x)) \), the condition whether to invoke a transport is determined by

\[
\begin{align*}
\gamma_{push}(x, \zeta_{rd}(x)) &= \exists \nu \in \mathcal{V}_1 \cup \mathcal{V}_2 \nu \cdot \gamma_{push}(x, \zeta_{rd}(x)) \\
\gamma_{pop}(x, \zeta_{rd}(x)) &= \exists \nu \in \mathcal{V}_1 \cup \mathcal{V}_2 \nu \cdot \gamma_{pop}(x, \zeta_{rd}(x))
\end{align*}
\]

V. Refinement

The necessity of quantification due to unknown variables as described in the previous section may result in a highly overestimated push/pop condition. The output of our synthesis tool has shown that in some nodes the same variable had to be quantified quite often. Therefore, the question arose whether it is possible to furthermore reduce the communication by sending additional variables to avoid its quantification of guards. In turn, this should give a less overestimated guard and reduce the communication caused by transport of other variables. This refinement was also implemented and is part of the benchmarks discussed in Section VI.

During the first benchmarks, we recognized that in some applications only a few transports have been modified. The reason can be found in the overall write and read guards, which are usually a disjunction of Boolean expressions. An existential quantifier applied to a disjunction may lead to true as soon as a sub-expression of the disjunction is evaluated to true. This case may occur, e.g., if a sub-expression is an unknown variable. Depending on the probability of a guard to be true and the
number of additional variables that are required to evaluate the non-quantified guard, the overall communication effort may be decreased.

![Figure 2](image)

Figure 2. Simple sketch to demonstrate that additional communication may reduce overall communication. Left: without additional communication, right: with additional communication

Figure 2 illustrates such an optimization using the sketched part of a DPN. After each node has fired once, 4 transports have to be done in the complete system on the left-hand side. One value is transported through the input channel, two values (\(x\) and \(y\)) through the middle channel and a fourth value through the output channel. The number of transports in the right-hand side depends on the probability of \(\gamma\) being true (\(P(\gamma = true)\)). In particular, the lower \(P(\gamma = true)\) is, the more communication can be saved. The average number of transports is then determined as follows: one value is transported through the input and output channel, i.e., three values in total because the second node now also receives the input value \(\gamma\) of the system. Between the given two nodes 2 \(\times P(\gamma = true)\) values are transported in average. In total, we have \(3 + 2 \times P(\gamma = true)\) tokens to send, which is less than 4 if \(\gamma\) holds in less than 50% of the cases. To this end, in this example, sending \(\gamma\) to the lower node as done on the right-hand side of Figure 2 reduces the amount of communication in the system if \(\gamma\) is false in most of the time.

To determine whether it is worth to send additional data depends on the amount of additional communication by these variables on the one side and communication reduction of sending other variables on the other side. In general, the net reduction of communication is determined as follows: Consider all transports between a node \(N_1\) and \(N_2\). Let \(V_{\text{unknown}} = V_1 \cup V_2\) be the set of variables that are unknown in \(N_1\) or \(N_2\). Consider \(V_{\text{add}} \subseteq V_{\text{unknown}}\) as the set of variables that have to be additionally sent to the reader or writer. Note that each variable in that set must be unknown by exactly one node, because it comes from an expression from its partner node, i.e., it must be known there. In conclusion, \(|V_{\text{add}}|\) is the number of transports that have to be added to the involved nodes \(N_1\) and \(N_2\) to make \(V_{\text{unknown}}\) known in both nodes.

The additionally sent variables affect all transports between node \(N_1\) and \(N_2\). Let \(T_{N_1,N_2}\) be the set of transports between the sender \(N_1\) and the receiver \(N_2\), i.e., \(T_{N_1,N_2} = \{\zeta \mid \zeta \in T \land \zeta_{\text{wr}} = N_1 \land \zeta_{\text{rd}} = N_2\}\). Let \(P(\zeta)\) be the probability for the case where the given transport \(\zeta\) is invoked in an iteration in case that no additional variables are known. Furthermore, let \(P_{\text{add}}(\zeta)\) be the probability for the case where the given transport \(\zeta\) is invoked in an iteration, if the variable set \(V_{\text{add}}\) is known in both nodes. The overall number of transports that have to be invoked is then given as: \(\Theta(N_1, N_2) = \sum_{\zeta \in T_{N_1,N_2}} P(\zeta)\), if no additional variables are known, and \(\Theta_{\text{add}}(N_1, N_2) = |V_{\text{add}}| \times \sum_{\zeta \in T_{N_1,N_2}} P_{\text{add}}(\zeta)\), if the variable set \(V_{\text{add}}\) is known in both nodes. The difference \(\Delta = \Theta(N_1, N_2) - \Theta_{\text{add}}(N_1, N_2)\) defines the number of transports that are not invoked per iteration, which already includes the transports for the additional variables. If \(\Delta > 0\), the insertion of additional transports for the variable set \(V_{\text{add}}\) can be considered as reasonable under the assumption that costs for transported values are the same for all variables. However, the costs for transports can be different in reality, e.g., since spatial distribution of nodes in a network results in different communication ways with different costs. Moreover, the calculation of probabilities requires precise knowledge of the probabilistic value distribution. This is a separate research topic and the interested reader may refer to Markov processes [30] for stochastic evaluation [1].

VI. RESULTS AND SUMMARY

The presented approach has been implemented in our Averest framework. All benchmarks presented in this paper have been written in the synchronous programming language Quartz and compiled to DPNs as described in [2] and [3], which already reduces the communication due to array structures. The resulting DPN is synthesized to C code. Because of its high abstraction level, the optimization allows us to use any back-end with message passing for the final synthesis output. In particular, we target shared-memory systems using PThread synthesis with FIFO buffer communication and distributed memory systems using MPI. The results of our benchmarks are depicted in Tables I and II. Table I shows the execution time for the unoptimized version, which is given in seconds. Speed-ups are given for different optimization methods: the first method is to synthesize the programs without any optimization that is presented in this paper. This serves as the reference to compare the optimization and its performance changes. All other methods use the presented approach to reduce the communication overhead at different levels. Thereby, each level includes the optimization from its previous optimization level. The speed-up and the amount of remaining communication that is gained by an optimization level is printed in Table I and Table II, respectively.

The first level in Table I applies the reduction as described in Section IV-F, i.e., only if all variables occurring in \(\gamma_{\text{wr}}(x)\) and \(\gamma_{\text{rd}}(x,N)\) (without existential quantification!) are known. The existential quantification can be a computationally expensive process depending on the expressions. This level is considered to check the valuability of the quantification process. The second level applies existential quantification to the read and write guards as described in Section IV-G. The last level uses the refinement from Section V, which adds transports to get a more precise guard evaluation, which in turn should reduce overall communication.

For all variants, we use the same partitioning, i.e., each created DPN has 16 nodes. We used two different systems to evaluate our approach. The first system is a i5-750 SMP system with four cores, and all the benchmarks are executed as a pthreaded application using the FIFO buffer implementation of the Intel®TBB library. The second system is a cluster of 4
Raspberry Pis (Rev. B), and all the benchmarks are executed as a MPI application.

Table II shows that the amount of data that has to be sent between nodes in a DPN can be reduced with our approach. The values in column “w/o opt.” are the absolute amounts of scalar values that have been sent in the unoptimized version. Other values remark the amount of communication reduction given as the percentage of the number of invoked transports compared to the unoptimized version. The reduction is between 1% to 67.5% depending on the application. To provide a better understanding of the benchmark results, we have to anticipate the conclusion: the guarded actions are mainly triggered by boolean variables that belong to the control flow of a synchronous program. The basic meaning of control flow in synchronous programs is to determine which part of the program is active in a macro step.

The parallel matrix multiplication has been implemented to provide maximal parallelism by calculating a complete matrix multiplication per iteration of the program. In each step, the program reads two matrices, calculates their product and sends it to the environment. As a consequence, each action in this program will be involved into the calculation in each step, making communication reduction nearly impossible. The sequential matrix multiplication calculates only one element of the result in each step. However, this involves less actions in the program, which are still executed in each step. Hence, the scope of optimization in this program is very limited, too.

Similar to the matrix multiplications, the design of the square root, landscape, DFT-IDFT, pitchshift and delay benchmark focused on maximal parallelism. Due to their complexity, they require more control logic compared to the matrix multiplication. As a result, they provide more possibilities for communication reduction.

Finally, the 3D transform provides the best benchmark result. This program reads parameters for a camera and calculates a transformation matrix. This matrix is applied to a sequence of vectors. The calculation of the transformation matrix is triggered by an input flag, which is also sent to other DPNs that do the actual transformation. Since this transformation matrix is calculated once for a sequence of vectors, it has to be broadcasted quite rarely. Hence, this benchmark provides the largest amount of saved transports.

The execution times (see Table I) of our benchmarks depend 1) on the saved amount of data transfers, which may be 2) reduced by the additional effort of evaluating guards that trigger data transfer. A third determining factor for the benchmark times are the remaining dependencies in a benchmark. Finally, the communication media strongly determines communication costs. Therefore, our communication reduction results in better speedups for systems where communication is more expensive, i.e., in our benchmarks using explicit message passing (MPI).

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>i5-750 (PThreads, Intel TBB Queue)</th>
<th>Raspberry Pi Cluster (MPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time/sec</td>
<td>Speedup</td>
<td>Time/sec</td>
</tr>
<tr>
<td>(w/o opt)</td>
<td>(w/o opt)</td>
<td>(w/o opt)</td>
</tr>
<tr>
<td>w/o quant. (level 1)</td>
<td>quant. (level 2)</td>
<td>quant. (level 1)</td>
</tr>
<tr>
<td>ref. (level 3)</td>
<td>ref. (level 3)</td>
<td>ref. (level 3)</td>
</tr>
<tr>
<td>3D transform</td>
<td>99.12</td>
<td>1.17</td>
</tr>
<tr>
<td>square root</td>
<td>240.43</td>
<td>1.04</td>
</tr>
<tr>
<td>LU decomp. (16 × 16)</td>
<td>385.19</td>
<td>1.31</td>
</tr>
<tr>
<td>matrix mul (16 × 16)</td>
<td>106.64</td>
<td>0.94</td>
</tr>
<tr>
<td>seq. matrix mul (16 × 16)</td>
<td>4.94</td>
<td>1.10</td>
</tr>
<tr>
<td>landscape</td>
<td>115.93</td>
<td>1.02</td>
</tr>
<tr>
<td>DFT-IDFT</td>
<td>847.94</td>
<td>0.94</td>
</tr>
<tr>
<td>pitchshift</td>
<td>38.68</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Table II. AMOUNT OF COMMUNICATION REDUCTION

<table>
<thead>
<tr>
<th>Benchmark Name</th>
<th>Amount of Communication Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/o opt</td>
<td>w/o quant. (level 1)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>3D transform</td>
<td>2.4e+9</td>
</tr>
<tr>
<td>square root</td>
<td>4.9e+8</td>
</tr>
<tr>
<td>LU decomp.(16^2)</td>
<td>9.3e+8</td>
</tr>
<tr>
<td>par. MatMult(16^2)</td>
<td>4.6e+8</td>
</tr>
<tr>
<td>seq. MatMult(16^2)</td>
<td>4.3e+7</td>
</tr>
<tr>
<td>landscape</td>
<td>2.1e+9</td>
</tr>
<tr>
<td>DFT-IDFT</td>
<td>1.0e+10</td>
</tr>
<tr>
<td>pitchshift</td>
<td>9.4e+8</td>
</tr>
</tbody>
</table>

The saved amount of data transfers strongly depends on the benchmark. Some characteristics can be derived from our results to decide whether it is reasonable to apply communication reduction. Potential candidates to attain speed-ups are benchmarks that contain variables that are rarely changed. The less often a variable is changed the less often its value must be sent to nodes that require that value. Moreover, less changes imply less computations. One side of the coin is that less values must be read for computations, which may reduce communication again. The other side of the coin is that less values must be read for computations, which may reduce communication again. The lower number of computations lead to a lower ratio of computation to communication. In many systems, the communication is much more expensive compared to executing an instruction. Hence, to efficiently exploit these systems, the ratio of communication to computation must be kept high. Pure data transforming algorithms, e.g., the parallel matrix multiplication, provide the worst results. In each iteration each node has to do computations based on values from all input buffers. Hence, the DPN does not allow communication reduction at all and our approach fails. Finally, the effectiveness of our approach also depends on the probability of transport guards. Determining these probabilities is an expensive process. As mentioned in Section V, this is a separate research topic [1, 30].

VII. CONCLUSION

In this paper, we presented a method to reduce the amount of data transfers in SDFGs that have been synthesized from synchronous programs. Based on the information whether a variable is changed by a writing node or read by a reading...
node, we can determine whether the value of this variable has to be sent between the sender (writing node) or read by the receiver (reading node). While the achieved reduction of data transfer endorses our approach, the benchmarks show that some refinements remain to be done.

In addition to our goal, which was synthesis of multi-threaded software from synchronous programs, this technique may be also of interest in the area of MPSoCs. A couple of processor elements (PEs) on a single die have to share a single bus. Hence, communication on that chip between the PEs becomes an important crucial factor for the overall performance, which may be improved with the presented approach.

REFERENCES


