Concurrency-Enhancing Transformations for Asynchronous Behavioral Specifications: A Data-Driven Approach

John Hansen and Montek Singh
University of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
{jbhansen,montek}@cs.unc.edu

Abstract

Several asynchronous system design tools are based on syntax-driven translation of behavioral specifications (e.g., Balsa, Haste). While they provide rapid design times, the performance of the resulting implementations is typically limited, in part because specifications written by designers often have limited concurrency due to unpipelined operation and unnecessary sequencing.

To overcome these challenges, this paper proposes a “source-to-source” transformation (i.e., code rewriting) of the original specification into a new one using a variety of concurrency-enhancing optimizations: (i) automatic parallelization, (ii) automatic pipelining, (iii) arithmetic optimization, and (iv) reordering of channel communication.

Our approach has been integrated into an existing design flow, and applied to a suite of examples. Experimental results demonstrate that our approach correctly and efficiently rewrites the original specifications into highly concurrent ones. If code length is used as an indicator of designer effort, our approach reduces the required effort by a factor of 3.3x on average (up to 8.8x). Alternatively, the impact of our approach can be quantified by the throughput improvement achieved by optimizing the original specification: up to 59x speedup using our basic approach, and a further 5.2x using arithmetic optimization.

1. Introduction

Many state-of-the-art tools for the design of asynchronous systems are limited in their ability to generate high-speed implementations with ease. The best-known tools (e.g., Haste/Tangram [6], and Balsa [1]) use syntax-directed translation to compile behavioral specifications directly to circuits, with few high-level optimizations. Straightforward specifications written by designers, however, often have limited concurrency due to unnecessary sequencing and unpipelined operation. As a result, either the designer must contend with rather slow implementations, or bear the burden of writing highly optimized specifications.

Burdening the designer with optimizing a specification has several drawbacks. First, writing highly concurrent code entails much effort and is error-prone. Second, such code often lacks readability and maintainability, and is therefore hard to modify and reuse. Finally, such a manual approach hinders automatic design-space exploration. In particular, in an ideal design flow, performance analysis tools are typically used to identify bottlenecks in the system, and then local modifications are applied to remove the bottleneck; this procedure is repeated until desired performance is achieved. Therefore, code rewriting should ideally be automated.

This paper introduces an alternative to manual optimization: an automated “source-to-source” compiler that transforms one behavioral specification into another behavioral specification with significantly higher concurrency. The proposed approach introduces a suite of transformations: (i) parallelization for increasing statement-level concurrency, (ii) pipelining for increasing concurrency within a statement group, (iii) arithmetic optimization for increasing concurrency at the substatement level, and (iv) reordering of channel communication. As a result, designers can write straightforward behavioral code, focusing mainly on its functional correctness rather than on its concurrency. The code is automatically transformed by our compiler to be highly concurrent, and then passed through the original design flow.

The two techniques of arithmetic optimization and communication reordering are fundamentally novel contributions of our approach. In particular, while basic parallelization and pipelining may help optimize a specification at the granularity of individual statements, there are often performance bottlenecks due to individual statements with long-latency arithmetic operations (e.g., 64-bit adds or multiplications). Further, a single statement may have a complex expression in-
volving multiple arithmetic operators. Our approach pushes concurrency enhancement down to a sub-statement level by introducing all of the following: expression refactoring to introduce parallelism, expression pipelining, and pipelining of individual (‘atomic’) operators. As a result, bottlenecks due to complex arithmetic are alleviated.

Our approach to reordering of channel communication actions addresses a challenging problem which, to the best of our knowledge, has not been addressed before. In particular, for a given module, changing the order of two communication actions is fundamentally different from reordering two computational actions. In the latter case, dependency analysis can easy help determine which reorderings or parallel groupings of those actions preserve the original semantics. However, channel communication inherently involves subtle synchronization issues, and naïvely reordering two communication actions may introduce a deadlock into the system. A conservative approach is to always maintain original order of channel actions; although safe, such an approach is suboptimal. Our strategy, instead, is to pursue a more optimal approach that includes a careful analysis to determine the space of legal code transformations. As a result, our approach provides greater opportunity for concurrency enhancement.

Previous approaches to improving the throughput of implementations produced by the Haste and Balsa tools have mostly focused at the circuit and intermediate (handshake) levels, including more optimized circuit-level designs of handshake components (e.g., more concurrent sequencers [12]), and peephole optimization and resynthesis at the intermediate level [4]. While some of these approaches have yielded significant speedup (1.54–2.06x), they are unable to take advantage of the significantly greater optimization opportunities at a higher level. As our results show, optimizing at the source level can provide an order of magnitude greater speedup. Moreover, the intermediate and circuit-level approaches are orthogonal to ours, and our approach does not exclude these optimizations from being applied within the design flow.

The domain of specifications targeted by our approach are slack elastic systems [9]. A slack elastic system preserves correct operation even if extra pipeline buffer stages (i.e., extra slack) are introduced on any communication channel. It was shown that a system is slack elastic if it is deadlock-free and it satisfies certain properties regarding channel probing and non-determinism [9]. Since our approach introduces pipelining into a specification, the assumption of slack elasticity is a requirement.

The proposed approach has been implemented in an automated tool, and evaluated on a suite of design examples. The resulting concurrency-enhanced specifications were run through the commercial Haste tools from Philips/Handshake Solutions [6], and synthesized to gate-level netlists and simulated. Experimental results demonstrate that our approach correctly and efficiently rewrites the original specifications into highly concurrent ones. If code length is used as an indicator of designer effort, our approach reduces the required effort by a factor of 3.3x on average (up to 8.8x). Alternatively, the impact of our approach can be quantified by the throughput improvement achieved by optimizing the original specification: up to 59x speedup using our basic approach, and a further 5.2x using arithmetic pipelining.

The remainder of the paper is organized as follows. Section 2 provides background on the Haste flow and on asynchronous pipelining, and reviews related previous work. Then, Section 3 presents our basic concurrency-enhancing transformations. Section 4 discusses some of the advanced topics, including arithmetic optimization, handling of conditionals and loops, and reordering of channel communication actions. Section 5 presents results, and finally Section 6 gives conclusions and future work directions.

2. Background and Previous Work

This section first reviews the Haste design flow, which is a commonly used syntax-driven translation approach for the design and simulation of asynchronous systems, and discusses its limitations. Next, asynchronous pipelines are briefly reviewed, along with a discussion of the distinctions between control-driven, data-driven, and data-flow design paradigms. Finally, prior related work is presented.

2.1. The Haste Design Flow

The examples discussed in this paper have been synthesized and simulated using the Haste design flow (formerly “Tangram”), a product of Philips/Handshake Solutions [6]. Haste is one of a few mature asynchronous design flows currently available; the toolset focuses on rapid design of low-power, low- to medium-speed custom asynchronous hardware.

The Haste toolset is a silicon compiler. It accepts specifications written in a high-level hardware description language, and compiles them, via syntax-driven translation, into a gate-level circuit. The high-level language is a close variant of the CSP behavioral modeling language [7].

The main Haste language constructs that are used in the presentation of this paper are:

- channel reads (IN?x)
- channel writes (OUT!x+y)
- assignments (a:=b+c)
- sequential composition (b:=a+x ; c:=b+y)
- parallel composition (a:=b+x || c:=d+y)
Figure 2 shows the Haste specification of a simple program, a single stage FIFO. The program has an input channel IN, through which it receives data items from the environment, and an output channel OUT, through which it transmits results to the environment. Each channel consists of a pair of request-acknowledge wires along with the data wires. In the specification, \( x \) is a storage variable. The main construct in the body of the specification is a \( \text{forever do} \) loop that performs the following actions: (i) read a value from channel IN and store it into variable \( x \); then (ii) write the value stored in \( x \) to the output channel OUT; and (iii) perform this sequence of actions repeatedly, forever.

Given a specification, the Haste compiler performs parsing, then syntactically maps each construct onto a predefined library component to generate a hardware implementation, as shown in Figure 2. In particular, there is a predefined component that implements the \( \text{forever do} \) construct: it repeatedly initiates handshakes with its target. Similarly, there is a predefined component that implements sequencing, denoted by \( ; \). The sequencer, upon receiving a handshake from its parent, performs a handshake with its left child followed by a handshake with its right child. The variable \( x \) maps to a storage element. Finally, the read and write operations, (e.g., read from channel IN and write to \( x \)) map to predefined components called \textit{transferrers}, denoted in the Figure by \( \text{——} \).

In summary, the compilation approach is quite simple but very powerful: fairly complex algorithms can be easily mapped to hardware. Gate-level implementations for complex designs, such as complete microcontroller, can be generated from a few hundred lines of high-level code.

\textbf{Performance Limitations.} As the number of statements increase in the code snippet in Figure 2, the size of the control cycle increases, resulting in a higher latency block. Several handshakes in the control tree may be required before an action can occur. As a result, the performance of the system suffers. We describe this situation as “control-dominated.”

\subsection*{2.2. Asynchronous Pipelining}

To overcome the performance limitation of large control cycles, a designer can introduce pipelining to reduce the control overhead. Figure 3 illustrates how control overhead is reduced in this situation. Pipelining replaces a large control tree with a forest of smaller trees governing the actions in the system. These actions are now initiated by channel communications directly, as opposed to sequenced by a complex controller. Thus, the single long control cycle in the original tree has been replaced by several relatively smaller control cycles local to individual computation blocks, thereby resulting in significantly better throughput.

In channel actions between stages, all of the variables that will be accessed in the remainder of the pipeline must be communicated. We refer to this set of variables as the “context” of a stage.

In a \textit{data-driven architecture}, the entire context is passed from one stage to the next, irrespective of whether an individual value is needed in the next stage as long as it is needed in some subsequent stage. Thus, once a result is produced, it may go through a number of intermediate stages before reaching the consumer. While this approach may seem somewhat expensive in terms of area and energy consumption, it is quite commonly used in practice due to its simplicity, including in pipelined microprocessors, FIR filters, etc.

In a \textit{data-flow architecture}, by contrast, concurrency may be further increased by allowing data to propagate directly to stages in which it is used [3]. As a result, the pipeline is typically forked off into many branches, which often reconverge. However, such an approach introduces additional challenges: the performance can suffer if branches are not properly balanced \( \text{i.e., not “slack-matched” [2]} \), potentially resulting in throughput that may be \textit{worse} than that of the slowest stage because of stalls caused by mismatched branching. In order to avoid slack mismatch issues, this paper therefore adopts the data-driven paradigm instead of full data-flow.

\subsection*{2.3. Previous Work}

Budiu et al. [3] introduced the approach of spatial computation, which compiles ANSI C specifications directly into hardware. Their approach includes a number of optimizations that aim to enhance concurrency. However, their work fundamentally belongs to a different domain—ANSI C software specifications—which is less general than the behavioral specifications targeted in this work. In particular, C specifications, unlike Haste, do not allow explicit communication via channels between processes to be modeled, whereas such communication is key to modeling complex asynchronous systems. In addition, fork-join style of concurrency cannot be explicitly specified by a designer in C; such concurrency again is central to many asynchronous system specifications. Finally, their approach does not consider
Figure 4: Original

Figure 5: Parallel

Figure 6: Pipelined

Figure 7: Parallelized and Pipelined

pipelining of atomic units, such as adders and multipliers, which is a key contribution of our approach.

Teifel et al. [13] and Wong et al. [14] have introduced approaches that translate specifications written in CHP [10] (a variant of CSP [7]) into pipelined implementations. While these approaches allow channel communication, their communication models can be restrictive, e.g., requiring that channel actions be unconditional or occur at most once in the body of a process. In contrast, this paper allows a more general framework for communication optimization.

Two recent approaches target conversion of behavioral specifications between CDFG and Haste/Balsa representations [11, 8]. Their goal is to leverage mature synchronous tools that are capable of performing resource scheduling, allocation and binding (under physical constraints), thereby getting around the limitations of the Haste and Balsa tools which lack such capability. These approaches also include some peephole optimizations, but do not aim to enhance system-level concurrency. In contrast, our approach specifically targets concurrency enhancement through pipelining, parallelization, and arithmetic and communication optimizations, with the goal of high system performance.

Many other approaches focus on peephole optimizations at the circuit and handshake level [12], [4] to improve throughput. However, solely using lower-level optimizations fails to take advantage of concurrency that can be gained at a higher level. Our approach does not preclude these optimizations, and in most cases these can be performed in an orthogonal fashion.

3. Basic Approach

In this section we describe how our compiler optimizes code through parallelization and pipelining. We first discuss how performance optimizations change the hardware structure of the system and give an overview of the optimizations that are performed at a source level. We then discuss how parallelization and pipelining are performed within our source-to-source compiler.

3.1. Method Overview

3.1.1. Hardware Level

Figure 4 shows an example of synthesized code and its representation in hardware. Each small block in the Figure represents a basic datapath operation. Similar to the case in Figure 3, control delays dominate, and the throughput obtained is rather low. The only channel communications that occur are with the environment. In essence, the original code is synthesized into a single, unpipelined, high latency block. The throughput of the system is solely determined by the latency of this unpipelined block.

Consider now the case where some operations in the original code are parallelized, as shown in Figure 5. The resulting circuit is still control driven, and again channel communication is only performed with the environment. The control tree is the same size, however some parallel blocks replace sequential blocks in the tree. As a result, the latency of the system is reduced, but the throughput is still determined by the latency of the whole system. The throughput of the system is solely determined by the latency of this unpipelined block.

The result of pipelining the original implementation is shown in Figure 6. Each operation now has its own individual latch to store data and channels to connect it with other stages. Note that the control cycle at each stage is considerably shortened. This data-driven pipeline has multiple,
low-latency stages, yielding an increase in system throughput. In this case, the throughput is limited by the cycle time of the slowest stage, rather than the latency of the whole system. Therefore, the throughput is increased, though possibly at the cost of some latency overhead.

By performing both optimizations, parallelizing then pipelining, the circuit of Figure 7 is produced. This circuit benefits from the reduced latency of parallelization, as well as the increased throughput of pipelining. Conversion to this design is the goal of our transformations.

3.1.2. Source Level

We now give an outline of how our algorithm is applied at the source level. Starting with a piece of straight-line, sequenced code, Figure 4, we transform it into the highly concurrent code of Figure 7.

The first step in the algorithm is to group the statements in a block of code that can be performed in parallel. In the code fragment in Figure 4, the assignments to variables c and d can be performed in parallel, and e and f can be performed in parallel, producing the circuit shown in Figure 5. This step performs simple instruction-level parallelization, reducing latency. However, we aim to increase performance by pipelining as well.

To pipeline, a channel is placed between every parallel grouping. This channel communicates the context for this dataset. We show a code fragment for the pipeline stages for the assignments to b, c, and d in Figure 7 (note that the procedure headers have been removed for clarity).

3.2. Class of Specifications Handled

3.2.1. Handling Specifications with Cycles

Even though the example of Figures 4–7 shows a code snippet that is acyclic, our optimization approach itself is fully capable of handling specifications with cycles. In particular, our approach is hierarchical: at each level of code hierarchy, a compound statement (e.g., if-then-else, while, etc.) as well as any statement group with cyclic dependencies is treated as an atomic statement for the purpose of performing parallelization and pipelining. The compound statement or cyclic group is then separately optimized when traversing the next lower level of hierarchy.

An example of how a cycle in a specification is handled is shown in Figure 8. In the code fragment of Figure 8b, the variable c is reused across iterations of the loop, creating a backwards dependency from statement 6 to statement 3. As a result, a cycle is introduced in the dependency graph (Figure 8a). Alternatively, cycles can also be caused by loop constructs such as for and while.

Our optimization approach operates hierarchically, and starts at the top level where it treats the cycle temporarily as a single node (Figure 8c), and parallelizes the body of the outermost block. Subsequently, the code within the cycle is parallelized. The resulting code is shown in Figure 8d.

While parallelization is performed at all levels of hierarchy, the pipelining transformation is typically not performed inside a cycle. In particular, typical cycles resulting from loops allow only a single token to be present in them at a time, and therefore their performance does not benefit from pipelining. Therefore, in our approach, pipelining is performed at the top level of the hierarchy, and further down the hierarchy into acyclic code blocks, until a cycle is encountered.

It is important to note that even if pipelining is not performed for a loop block, the loop’s performance may still be improved by parallelization. In particular, as seen in Figure 8d, parallelization can shorten the latency of the cycle, which translates to both shorter latency and shorter cycle time at the next higher level of the hierarchy. This topic is dealt with in detail in Section 4, which discusses other methods for loop optimization as well.

3.2.2. Set of Language Constructs

We handle the full set of Haste constructs allowable; however, the transformed specification will be equivalent to the original one only if the original specification satisfies the conditions for slack elasticity [9]. In particular, loops, conditionals, case statements, function calls, sequential, and parallel constructs are all supported. Similar to Figure 8, more complex constructs are collapsed into a single node and each of these are further hierarchically parallelized. Similarly, pipelining is hierarchically performed until blocks with cyclic dependencies are encountered. Details on the handling of some of these complex constructs (specifically, conditionals and loops) are presented in Section 4.

3.3. Parallelizing Transformation

At the core of the parallelization transformation is dependence analysis. This subsection briefly describes how this analysis is performed, then shows how the results allow the compiler to modify the program to increase concurrency.
The process of generating channel communications between stages using IN and OUT sets.

To begin the pipelining transformation, the compiler first breaks every group of statements delimited by a sequencer (\( ; \)) into its own pipeline stage. In source code, each stage will be represented by a statement block in which the initial statement is a channel read and the final statement is a channel write. The channel read accepts the context from a prior stage; the channel write transmits the updated context to a subsequent stage.

To complete the transformation, the correct context for each stage must be determined. First, the compiler visits each stage, building a list of the variables accessed (\( \text{VAR}_x \)) by the group of statements in that stage. Next, the compiler generates the IN set for each stage, which consists of all the variables in use prior to or within the stage. IN sets are determined using the following productions, where \( x \) indicates the stage number:

\[
\text{IN}_x = \text{IN}_{x-1} \cup \text{VAR}_x, \quad \text{IN}_1 = \emptyset
\]

The compiler then determines the OUT set for the stage: the set of all variables accessed in subsequent stages. A similar production is used (\( n \) indicates the final stage in the pipeline):

\[
\text{OUT}_x = \text{OUT}_{x+1} \cup \text{VAR}_{x+1}, \quad \text{OUT}_n = \emptyset
\]

Two important observations are made by comparing the IN and OUT sets for each stage. First, if a variable is contained in a stage’s IN set but not contained in its OUT set, that variable will be accessed in this stage, but will not be accessed in any future stages. Therefore, the variable does not need to propagate beyond this stage.

Second, a variable that exists in the OUT set of a stage but not in its IN set is being used for the first time in the next stage. If the variable is read in the next stage, the read can be replaced with the variable’s initialization. In this case, the current stage sends the initial value of the variable, or zero if the variable is declared without an initialization. If the variable is only written in the next stage, the current stage does not need to communicate a value for the variable, since it will merely be overwritten.

Using the IN and OUT sets for each stage, the context for each stage is determined. For a stage \( x \), the set of variables in the stage’s context is the following:

\[
\text{context}_x = \text{OUT}_{x-1} \cap \text{IN}_x
\]

The variables that must be communicated on its output channel are:

\[
\text{context}_{x+1} = \text{OUT}_x \cap \text{IN}_{x+1}
\]

Once the contexts have been computed for each stage, channel reads are inserted for each stage after the first. Likewise, a channel write is inserted for all stages except the last. In operation, each stage will read in the values of each variable needed in this stage or a future stage. The stage will

---

**Figure 9: Precedence Graph with Parallel Groupings**

Figure 9 shows a sample precedence graph. After the graph is generated, a topological sort of the graph is performed. (As described in Section 3.2.1, any cycles encountered as treated as atomic statements for the purpose of this sorting.) Each statement that has no input edges (dependencies) is placed into the first grouping of parallel statements. These statements are then removed from graph, along with any edges they produce. Next, all statements that have no input edges are placed into the second grouping, then their edges are removed. The process repeats iteratively until all the statements are placed into a grouping.

The compiler then generates a new subtree in which parallel groupings are children of a parallel (\( || \)) construct. The parallel groupings are in turn combined using sequencers (\( ; \)).

Our approach employs variable renaming to achieve greater concurrency enhancement. Thus, if a second assignment to a variable occurs within a block of code, the target location is renamed, along with any future accesses. As a result, write-after-read and write-after-write dependencies are removed from the graph.

Theoretically, even further concurrency can be achieved by using a partial ordering of the statements, resulting in a full data-flow specification at the cost of greater forking and joining. However, with the resulting greater branching, challenging slack matching issues may arise, and without an effective pipeline-balancing approach, the resulting specifications can exhibit reduced throughput. This issue was the motivation behind adopting the simpler data-driven approach in this paper, instead of a fully data-flow approach.

### 3.4. Pipelining Transformation

Pipelining is an orthogonal process to parallelization; it can be performed on code that is sequential or has already been parallelized. We assume these specifications read from a set of input channels, perform a computation, and write to a set of output channels. This subsection discusses how pipelining is achieved for such a specification, in particular,
then perform operations on these variables using the concurrent statement grouping associated with the stage. If a variable is modified, the output channel will transmit an expression containing the updated value. If unmodified, the output channel will merely transmit the original value of the variable. The channel read, variable modification, and channel write are then nested within a forever do loop, creating a pipeline stage. This process is followed for each stage to create a complete data-driven pipeline.

4. Advanced Techniques

This section describes several advanced approaches for improving the performance of a specification. We first discuss several methods for optimizing arithmetic operations, then describe how conditionals and loops are handled. Finally, we present an approach for increasing concurrency in the presence of channel communication.

To the best of our knowledge, the two techniques of arithmetic optimization and communication reordering are fundamentally novel contributions. The former pushes concurrency enhancement to the substatement level, whereas the latter technique enlarges the space of solutions by carefully allowing communication between distinct modules to be reordered without the introduction of deadlocks.

4.1. Arithmetic Optimization

While our basic approach can potentially obtain substantial speedup by optimizing code at the statement level, further improvement is possible by optimizing at the substatement (i.e., expression and operator) level. We now describe three methods for optimizing arithmetic computation: expression tree balancing, which can reduce the latency of a series of arithmetic operations; and expression pipelining and operator pipelining, which can improve the throughput of a system.

4.1.1. Balancing Expression Trees

Many languages, including Haste, rely on both operator precedence and a left-right expression ordering to determine how sub-expressions are evaluated. Therefore, expression trees produced by the parser can be unbalanced, even linear in some cases. Refactoring sections of the tree by taking advantage of operator associativity can lead to more balanced expression sub-trees, and introduce additional concurrency into the specification. This optimization reduces the overall depth of the tree, improving both latency and throughput for a statement.

For example, the expression $a+b+c+d$ initially requires three sequential addition stages. Tree balancing converts the expression to $(a+b) + (c+d)$, which requires only two sequential addition stages since evaluating expressions $a+b$ and $c+d$ can be performed in parallel. As this optimization may change the meaning of a program in exception cases (e.g., overflow), it is provided as an option to the user. In effect, this optimization can be regarded as parallelization pushed to the granularity of arithmetic expression evaluation.

4.1.2. Expression Pipelining

While balancing expression trees can provide some benefit to throughput if the latency of a statement is reduced, a statement with a large expression tree can still be a major bottleneck in the specification. By pipelining a computationally complex expression tree, significant gains in throughput can be attained.

To perform expression pipelining, we divide the original statement’s expression tree into several smaller assignments. Each atomic sub-expression becomes its own individual assignment with a single arithmetic operation. For example, consider a statement with a complex expression tree: $a := ((b+c) * d) + e$. Through optimization, a series of simple statements are produced: $t_1 := b + c$; $t_2 := t_1 + d$; $a := t_2 + e$. In essence, expression pipelining replaces one high-latency pipeline stage with multiple low-latency stages, improving throughput.

4.1.3. Operator Pipelining

Further gains in throughput are achieved by decomposing and pipelining individual arithmetic operators. Haste implementations, however, pose a special challenge to correctly pipelining an arithmetic function. In particular, naïvely replacing an arithmetic unit such as a combinational adder with a pipelined adder circuit does not yield any throughput improvement. This is because the controller associated with the stage that contains the combinational adder allows only one token in that stage at a time. Therefore, in order to pipeline the adder, that stage’s controller itself must be modified; this modification is more easily performed at the source level.

In order to effectively pipeline arithmetic, control must be distributed to individual stages. This is achieved in source code by breaking down the arithmetic operation into several smaller assignments. Figure 10 illustrates how pipelining using finer granularity operators is performed for a 32-bit addition. First, each operand is broken down into four 8-bit operands. These operands are then fed into four 8-bit adders in parallel to produce 9-bit results (including the carry out). The partial results are then combined sequentially to produce the final value.

By implementing operator pipelining at the source level, not only is throughput increased, but latency can be improved as well. In particular, the source-level decomposition of individual operators into several smaller operations affords new opportunities for parallelization (i.e., exploiting parallelism among stages of distinct pipelined operators).
Figure 11: Replacing Conditionals with Conditional Assignments

Performing this task by hand is a time-consuming and error-prone operation, but is well-suited to our source-to-source compiler.

4.2. Conditional Optimization

Not all code the user wishes to synthesize is linear in nature, as conditionals (if-then-else) are often present. There are many options to handle these breaks in linearity.

**Conditional Assignment.** If both branches of a conditional consist solely of variable assignments, i.e., no channel communications or loops exist in either branch, conditional assignment of variables is the preferred method. To perform a conditional assignment, the assignments in either branch are removed and replaced with a tertiary assignment outside of the conditional. The form is as follows:

\[
\text{var} := \begin{cases} \text{exp} & \text{if bool} \\ \text{exp} & \text{else} \end{cases}
\]

Consider the code in Figure 11. In the else branch, the variable \( x \) is assigned \( x+1 \). In the then branch, no assignment is made. The assignment can be removed from the loop and replaced with a conditional assignment:

\[
x := \begin{cases} \text{if } a > b \text{ then } x \text{ else } x+1 & \end{cases}
\]

If assignments are made in both branches, such as for variable \( y \), the same idea applies:

\[
y := \begin{cases} \text{if } a > b \text{ then } y-1 \text{ else } y+1 & \end{cases}
\]

If the Boolean condition itself is a function of variables modified in either branch, the Boolean must be computed and stored prior to performing the conditional assignments in order to preserve the semantics of the conditional. Finally, if several writes to the same variable occur in both branches, variable renaming is employed.

**Early Decision.** A second option for handling conditionals is early decision. Early decision (Figure 12) is used when either branch contains a channel communication or internal loop. It is necessary that the pipeline be split into two branches to handle this situation: one containing the ‘then’ branch, the other containing the ‘else’ branch. Two additional stages are introduced: one that forks the branches prior to execution, and one that merges them after execution.

Figure 12: Early / Late Decision in Conditionals

In early decision, the value of the conditional’s Boolean is computed prior to entering either branch, just as it would in a normal system. After the computation, the fork stage decides the path to which the context should be sent. The context is then operated on by the proper branch, and then accepted by the merge stage to be sent out.

If the two paths are poorly matched in terms of slack and forward latency, early decision may result in out-of-order execution of consecutive datasets. In some cases, such as computer graphics and networking, out-of-order execution is allowable. If, however, correct order is required by the user, a third Boolean path (with buffering) is introduced between the fork and join stages to indicate which branch the join stage should read from to preserve execution order.

**Late Decision.** A final alternative, Late decision (Figure 12), can be applied in the case where either branch contains an internal loop, but cannot be applied when channel communication is performed by the branches. In late decision, both branches are executed concurrently, and the correct result is later chosen based on the Boolean outcome. This is a form of speculation. Because both paths are taken regardless of the value of the Boolean, channel communication inside the conditional is disallowed; otherwise unnecessary channel actions could potentially occur, thereby compromising the system’s correctness.

In late decision, the pipeline must be split into three branches, two for then and else and one for the Boolean value. Again, a fork and a join stage must be included in the pipeline. At the join stage, all three branches have completed computation. The join stage selects the context from the correct branch using the Boolean value and forwards it, discarding the context from the incorrect branch.

Late decision suffers from poor energy consumption and can also limit throughput if the branches are not slack-matched. However, the latency of the conditional can be reduced if the Boolean takes a long amount of time to compute. Early decision, in comparison, has the advantage of high throughput even if the paths are not slack matched.

4.3. Optimization of Loops

Loops are a significant roadblock for designers aiming for high throughput specifications. In typical implementations, new data items cannot enter the loop until the current item exits, a loop effectively acts as a single, high-latency stage. Pipelining the internals of a loop provides no benefit if the loop contains a single token, and in fact decrease performance of a system due to latency overheads.

The designer does have three potential options for increasing performance: (i) statement parallelization, (ii) ex-
pression tree balancing, and (iii) loop unrolling. All three optimizations have the potential for reducing latency, and thus improving the throughput of the loop.

We have previously discussed how statement parallelization and expression tree balancing are performed in the compiler. The third option, loop-unrolling, is performed in the same manner in this domain as in software, and provides the potential for statement interleaving across loop iterations. Both full and partial unrolling can be performed. However, since loop unrolling essentially replicates hardware, this optimization comes at a cost of area. We have performed full unrolling for one of the benchmarks in Section 5.

While traditional design methods typically allow only a single token inside an algorithmic loop, recent work by Gill et al. [5] introduced a novel approach to implementing loops that can operate on multiple tokens concurrently. This technique, called loop pipelining, correctly handles the flow of control and all data dependency challenges created by allowing multiple tokens inside a loop, thereby significantly increasing throughput. This optimization can be performed at the source level, and we plan to integrate it into our source-to-source compiler in the near future.

4.4. Communication Optimization

The presence of channel communication introduces new challenges for code rewriting. In particular, simply relying on dependence analysis is not sufficient to determine the space of legal reorderings and groupings of statements. The reason is that communication involves not only flow of data but also control synchronization. In fact, sometimes communication actions omit data altogether, and are simply used to synchronize two modules. Naïvely reordering communication actions can introduce a deadlock into the system.

A safe but suboptimal approach is to prevent re-ordering of channel actions altogether. Instead, we introduce a more flexible approach for handling communication which includes a careful analysis of computation and communication actions within and across modules, to determine the space of legal code rewritings. As a result, our approach allows more opportunities for concurrency enhancement.

In this subsection, we first illustrate how channel communication makes code rewriting challenging, and then describe our proposed approach.

Avoiding Deadlock. In our first example, we illustrate the effects of re-ordering a pair of channel communications. Consider module 1 that consists of three channel communications: A?a; B!a; C?c. Here we see a data dependence only between the first two communications. Suppose the counterpart channel communications for A, B, and C are in three distinct modules with no other channel actions. The channel action on C is ready to be performed earlier in the module, and is only prohibited by the sequencing of this module. In this example, re-ordering can increase the concurrency and reduce the latency of the module. One approach would be to parallelize the channel actions: (A?a||C?c); B!a.

However, suppose that the counterpart channel actions for A, B, and C are all contained in a single module, in which they are all sequenced: (A!a; B?b; C!b). Here a data dependence exists between the channel communications on B and C. By re-ordering the channel actions in the original module, a deadlock has been introduced.

Solution Overview. To be able to safely optimize, the designer must first determine the flow of data across channels. This goal can be accomplished by building a directed graph, as shown in Figure 13a.

For each channel in the specification, a node is introduced in the directed graph. A dashed edge is drawn between two nodes if the two channels have sequenced actions in any module (IN1?a; IN2?b), while full edges are drawn between nodes that have a data dependence. Data dependences can either occur directly: F?f; OUT!f, or due to statements sequenced between the two communications: A?a; c:=a+b; C!c.

By removing all of the sequencing arcs in the original specification, the directed graph in Figure 13b is produced. The compiler can use the optimized directed graph to generate a new behavioral specification by creating parallel groupings, similar to Figure 9. More generally, the compiler may re-order and parallelize communications as long as two criteria are met: (i) by sequencing channel actions in a module, new edges must be inserted in the graph, and (ii) the resulting graph may not contain a cycle, as this indicates a deadlock has been introduced.

While several orderings are possible from the outlined rules, our approach is to parallelize all channel communications in a grouping, and ensure that all groupings are performed in sequence. This precludes the opportunity for introducing deadlock in a system.

Performance. In general, communication optimization is focused on increasing the concurrency of a specification. By performing pipelining balancing, the opportunity exists for improvement in throughput and latency. One scenario where throughput can be improved is when stalling occurs in the original specification due to unnecessary sequencing of channel actions. A rough indication of latency can be determined by counting the number of nodes in the longest path in the directed graph; in the case of Figure 13, the longest path is reduced from 9 nodes to 4 nodes.
<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Parallel</th>
<th>Pipelined</th>
<th>Parallel+Pipelined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycle Time (ns)</td>
<td>Latency (ns)</td>
<td>Cycle Time (ns)</td>
<td>Latency (ns)</td>
</tr>
<tr>
<td></td>
<td>190.6</td>
<td>381.2</td>
<td>99.85</td>
<td>199.7</td>
</tr>
<tr>
<td>Ay1cde</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>50.09</td>
<td>15.02</td>
<td>45.06</td>
<td>13.28</td>
</tr>
<tr>
<td></td>
<td>790.3</td>
<td>790.3</td>
<td>790.3</td>
<td>13.37</td>
</tr>
<tr>
<td></td>
<td>343.1</td>
<td>343.1</td>
<td>343.2</td>
<td>14.74</td>
</tr>
<tr>
<td></td>
<td>1218</td>
<td>1218</td>
<td>576.8</td>
<td>1246</td>
</tr>
<tr>
<td></td>
<td>132.4</td>
<td>132.4</td>
<td>118.6</td>
<td>137.3</td>
</tr>
<tr>
<td></td>
<td>1997</td>
<td>748.0</td>
<td>748.0</td>
<td>187.0</td>
</tr>
<tr>
<td>Cyclic</td>
<td>341.6</td>
<td>341.6</td>
<td>341.6</td>
<td>342.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ODE</td>
<td>1497</td>
<td>1497</td>
<td>748.0</td>
<td>187.0</td>
</tr>
<tr>
<td>ROOT</td>
<td>132.4</td>
<td>132.4</td>
<td>118.6</td>
<td>137.3</td>
</tr>
<tr>
<td>QUAD</td>
<td>1997</td>
<td>748.0</td>
<td>748.0</td>
<td>187.0</td>
</tr>
<tr>
<td>LTEA</td>
<td>341.6</td>
<td>341.6</td>
<td>341.6</td>
<td>342.4</td>
</tr>
</tbody>
</table>

Table 1: Performance of Original and Transformed Specifications

<table>
<thead>
<tr>
<th>Optimization Technique / Granularity of Operator Pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>FIR</td>
</tr>
<tr>
<td>ALU</td>
</tr>
<tr>
<td>ADD</td>
</tr>
</tbody>
</table>

Table 2: Performance Improvement through Operator Pipelining

5. Results

Our optimizing compiler was written in Java and executed on a 2.16GHz Core Duo processor system with 2GB of RAM. Execution time for our compiler was 0.2 to 2 sec in all cases, with the exception of one example (ADD) which took 2 min. These runtimes are quite short for a Java implementation, and even in the slowest case an order of magnitude shorter than the subsequent compilation step by the Haste compiler.

Experimental Setup. Each example was designed and simulated using the Haste/TiDE design flow (formerly “Tangram”) from Philips/Handshake Solutions [6], described earlier in Section 2. All designs were synthesized to the gate level using a generic tech library supplied with the tools.

We chose eight different benchmarks to illustrate the effects of our approach: (i) FIR: a simple FIR filter that performs a weighted average of the previous eight inputs, (ii) ALU: an ALU based on the communication example of Section 4 that receives two inputs and produces their sum and difference, (iii) ADD: a 64-way adder tree derived from a real-world Boeing project, (iv) UTEA: an unrolled version of the Tiny Encryption Algorithm, (v) ODE: a differential equation solver, (vi) ROOT: a specification that performs the square root of an input, (vii) QUAD: a specification that determines if the quadratic roots of two input polynomials are interleaved, and (viii) LTEA: a second version of the Tiny Encryption Algorithm, without unrolling.

The first four specifications (FIR, ALU, ADD, UTEA) consist of straight line code, while the second four specifications (ODE, ROOT, QUAD, LTEA) include one or more loops. For each example, our approach was used to create several transformed specifications: parallelized, pipelined, and both parallelized and pipelined. Furthermore, arithmetic pipelining at 32, 16, 8, and 4 bit granularities was performed using three specifications in addition parallelization and pipelining.

The ROOT, QUAD, and LTEA specifications contain very tight loops with little room for internal parallelization. ODE’s loop, however, contains several sequenced operations that have the potential to be parallelized. QUAD contains two ROOT loops in sequence with no data-dependencies between the loops, as well as several unpipelined 64-bit multiplications in each version.

Our optimization approach was fully automated by the tool and applied without manual intervention to every example with the exception of ALU, for which tool integration issues required one communication-related optimization to be performed manually. All other optimizations for ALU, and all of the optimizations for all other examples were fully automated.

Performance. Table 1 shows the cycle time and latency for each specification with the arithmetic pipelining option disabled. Throughput generally increases (cycle time generally decreases) from left to right in the table, most notably for the four straight-line code examples. The four examples with loops also show throughput improvement, although in most cases only due to parallelization which tightens the loop body. As expected, pipelining alone had no benefit on the examples with loops except for QUAD, which consists of not just one large loop, but two sequential ROOT loops along with several other complex sequential operations. As a result, QUAD was significantly sped up (8x) by pipelining. Overall the greatest throughput improvements were for the straight-line examples: from 2.2x for ALU, to 14x for FIR, 23x for LTEA, and 59x for the adder tree (ADD).

Latency is generally reduced (or remains unchanged) after performing parallelization, but is usually increased after per-
forming pipelining. As a result, the latency of parallelized and pipelined specifications is increased in some cases and decreased in others: 1.2x longer for UTEA, but 8.4x shorter for ADD.

The table shows a few intriguing anomalies. The first is the reduced latency of FIR and ADD after performing parallelization and pipelining, when compared to the parallelized version. This reduction is due to expression tree balancing, which was only enabled when both parallelization and pipelining were performed. A second anomaly is the reduced latency of FIR after performing pipelining. FIR’s original specification required flip-flop variables due to auto-assignment. The pipelining optimization was able to replace these flip-flops with lower latency latches, thus reducing the overall latency of the specification.

A second set of results shows the effect of arithmetic pipelining on the first three straight-line code examples. As shown in Table 2, an additional 5.2x improvement was achieved by pipelining arithmetic in addition to the previous optimizations, as long as the pipeline is not limited by other higher granularity stages such as those containing loops.

**Area and Design Effort.** Table 3 gives numbers for the total area of synthesized circuits, as well as the number of lines for each specification as an indication of design effort. By comparing the number of lines in the original specification to the number of lines in the parallelized and pipelined specification, we can get a sense of how much effort was saved by using our tool versus performing these optimizations by hand. This amount averages around 3.3x, and is 8.8x in the best case (UTEA). In terms of area, the original and parallelized specifications have similar numbers, while the pipelined version generally has significantly more area. The parallelized and pipelined version falls somewhere between the original and pipelined versions in most cases.

<table>
<thead>
<tr>
<th></th>
<th>Original Area ($\mu m^2$)</th>
<th>Lines (n)</th>
<th>Parallel Area ($\mu m^2$)</th>
<th>Lines (n)</th>
<th>Pipelined Area ($\mu m^2$)</th>
<th>Lines (n)</th>
<th>Parallel+Pipelined Area ($\mu m^2$)</th>
<th>Lines (n)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Acyclic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR</td>
<td>12365</td>
<td>45</td>
<td>11225</td>
<td>47</td>
<td>25204</td>
<td>209</td>
<td>11869</td>
<td>88</td>
</tr>
<tr>
<td>ALU</td>
<td>2441</td>
<td>46</td>
<td>2482</td>
<td>56</td>
<td>5913</td>
<td>227</td>
<td>3495</td>
<td>83</td>
</tr>
<tr>
<td>ADD</td>
<td>44610</td>
<td>76</td>
<td>44610</td>
<td>76</td>
<td>658913</td>
<td>1661</td>
<td>53248</td>
<td>217</td>
</tr>
<tr>
<td>UTEA</td>
<td>32731</td>
<td>61</td>
<td>32736</td>
<td>61</td>
<td>31225</td>
<td>551</td>
<td>30754</td>
<td>540</td>
</tr>
<tr>
<td><strong>Cyclic</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ODE</td>
<td>1750</td>
<td>28</td>
<td>1600</td>
<td>34</td>
<td>2292</td>
<td>91</td>
<td>2010</td>
<td>84</td>
</tr>
<tr>
<td>ROOT</td>
<td>928</td>
<td>27</td>
<td>924</td>
<td>33</td>
<td>1245</td>
<td>94</td>
<td>1122</td>
<td>76</td>
</tr>
<tr>
<td>QUAD</td>
<td>6213</td>
<td>54</td>
<td>5880</td>
<td>70</td>
<td>9412</td>
<td>294</td>
<td>6969</td>
<td>170</td>
</tr>
<tr>
<td>LTEA</td>
<td>5268</td>
<td>36</td>
<td>5268</td>
<td>36</td>
<td>6475</td>
<td>86</td>
<td>6475</td>
<td>86</td>
</tr>
</tbody>
</table>

|                |                           |           |                           |           |                           |           |                                    |           |
| **Table 3: Area and Code Length** |

6. Conclusion

This paper proposed a source-to-source compiler to increase the performance of a specification while maintaining ease of design. Our automated approach yielded improved throughput for a full suite of specifications, up to 59x in one case (293x with arithmetic pipelining). By performing these optimizations using an automated tool rather than by hand, design effort was reduced by up to 8.8x.

In our ongoing work, we plan to leverage the loop pipelining approach of [5], in order to provide better optimizations for while/for loops. We aim to implement a wider variety of conditionals: including variants on early and late evaluation. Finally, we plan to target full dataflow implementation as an alternative to the data-driven approach. With proper slack matching, we believe a dataflow architecture may be able to enhance performance even further.

**References**


