CONCURRENCY-ENHANCING TRANSFORMATIONS FOR ASYNCHRONOUS BEHAVIORAL SPECIFICATIONS

John B. Hansen

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill
2007

Approved by:

Montek Singh

Anselmo Lastra

John Poulton

ABSTRACT

JOHN B. HANSEN: Concurrency-Enhancing Transformations for Asynchronous Behavioral Specifications
(Under the direction of Montek Singh)

State-of-the-art synthesis tools for the design of asynchronous systems rely on syntax-driven translation of behavioral specifications. While these tools provide the benefit of rapid design, they are severely limited in the performance of their resulting implementations (e.g., 10-100 MHz). This research proposes a synthesis approach that builds upon the existing state-of-the-art tools, preserving rapid design times and allowing for an order of magnitude increase in performance.

In particular, this thesis proposes a powerful approach to enhance the *concurrency* of the original behavioral specifications. The proposed approach is a "source-to-source" transformation of the original behavioral specification into a new behavioral specification using two specific optimizations: automatic parallelization and automatic pipelining.

The approach has been implemented in an automated design tool and applied to a suite of examples for validation. All examples were synthesized to the gate level after optimization and compared with the original, non-optimized versions. Results indicate improvement in throughput by a factor of up to 23X and a reduction in latency by up to 72%.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

There is a resurgence of interest in asynchronous or "clockless" design due to its potential to mitigate some of the imminent challenges to synchronous design, including difficulties in high-speed clock distribution, management of power consumption, and increasing demands of design modularity and reusability [1, 3, 5]. This thesis focuses on the automated design of high-speed asynchronous systems from high-level behavioral specifications.

Existing state-of-the-art asynchronous synthesis tools are limited in their ability to automatically generate high-speed implementations. The best-known industrial-strength tools (*e.g.,* Haste from Philips / Handshake Solutions [7]) use syntax-directed translation to compile high-level behavioral specifications directly to low or medium-speed implementations. In many cases, control overhead limits performance; this is due to the large control trees generated by many syntax-directed translation tools. Little is done by these tools to remedy performance loss; at most, they perform peephole optimizations at the *circuit level*.

These existing tools provide little or no support for higher-level concurrency-oriented optimizations such as instruction parallelization of the specification. As a result, while design times are shortened, the performance of automatically synthesized implementations has thus far generally been limited to 10-100 MHz. For asynchronous design to be a viable alternative to clocked design, the critical challenge of providing design tools for high-performance implementations must be addressed.

A further drawback of popular existing tools such as [7], which use syntax-directed translation, is that design-space exploration is quite difficult. Optimizations to the implementation must typically be performed at the source level by modifying the behavioral specification. Therefore, the burden of optimizing a design falls squarely on the designer, who must rewrite the behavioral description. Such source-level optimizations by hand are time-consuming, and can greatly hinder productivity by reducing readability, modularity, and reusability, and can increase the possibility of design errors.

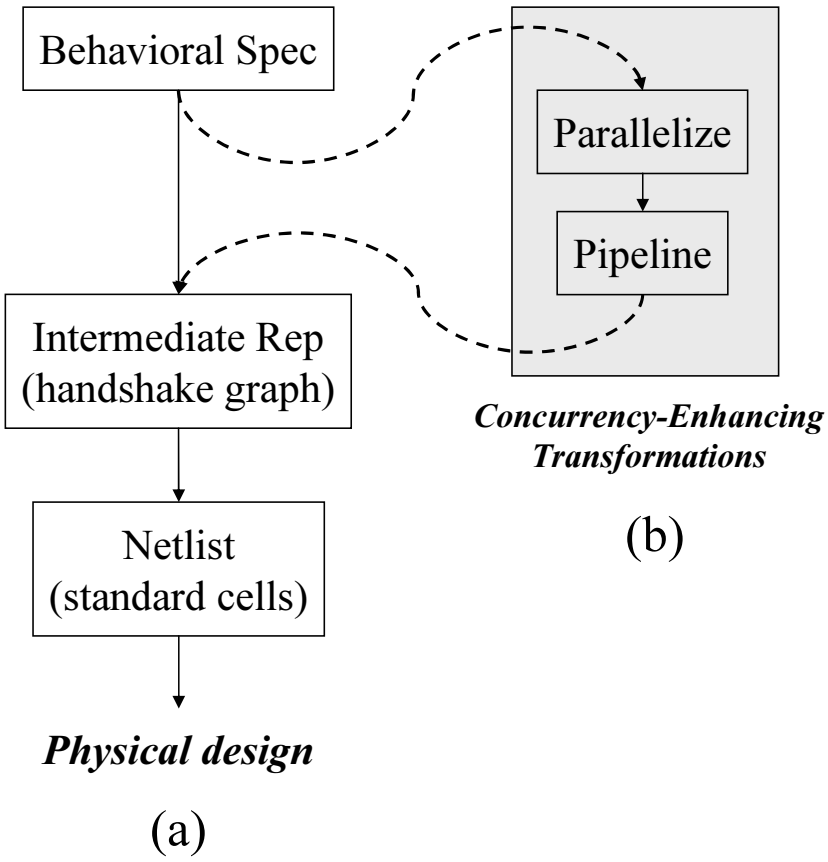


Figure 1.1: Design Flow: (a) existing flow, and (b) proposed source-to-source transformations

This thesis presents 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 enhances performance in three ways: (i) increasing instruction-level concurrency through *parallelization,* (ii) increasing instruction group level concurrency through *pipelining,* and (iii) reducing control overhead by making the specification *data-driven.*

The resulting implementations are high-throughput, low-latency, data-driven pipelined systems in which individual datapath operations, or small groups of such operations, become concurrent processes. Synchronization between the processes (*e.g.,* due to data dependencies) is achieved via point-to-point communication, instead of by the rather complex control trees that are generated by syntax-driven translation tools.

A key contribution of this work is to handle communication (via point-to-point channels), in addition to computation. In particular, the concurrency-enhancing transformations of parallelization and pipelining account for, and correctly address, conflicts

and dependencies via channel communication. In particular, the transformations are constrained in order to guarantee safety, *i.e,* no data or control hazards, or deadlocks, are introduced as a result of the transformations. In addition, the approach guarantees that the order of communication with the environment is preserved.

The proposed approach has been implemented in an automated tool, and evaluated on a suite of stream-processing examples specifications. The resulting concurrency-enhanced specifications were run through the commercial Haste tools from Philips / Handshake Solutions [7], synthesized to gate-level netlists, and simulated. The simulation results show the resulting implementations to have throughputs that are higher by a factor of up to 23x when compared to the original behavioral specifications.

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

CHAPTER 2

Background and Previous Work

This chapter first introduces the Haste design flow, a commonly-used syntax-driven translation approach for the design and simulation of asynchronous systems, and discusses its limitations. 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

2.1.1 Representation and Compilation

The examples discussed in this thesis have been synthesized and simulated using the Haste design flow (formerly "Tangram"), a product of Philips / Handshake Solutions [7]. Haste is one of a few mature asynchronous design flows currently available; the toolset focuses on rapid design of custom asynchronous hardware. It targets low- to medium-speed low-power implementations running in the 10-100 MHz range (in 0.13μm technology).

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 [8], which allows complex behaviors to be easily specified in a few lines of code.

The Haste language is robust; if offers many constructs to control the flow and operation of a program. The proposed compiler accepts specifications that use any of the constructs in the full language; however only a subset of them are described in this thesis. The main Haste language constructs that are used in the presentation of this thesis are:

- channel reads (`IN?x`)

```
&fifo=proc(IN?chan byte &
           OUT!chan byte).
begin
  & x: var byte
  | forever do
      IN?x;
      OUT!x
    od
end
```

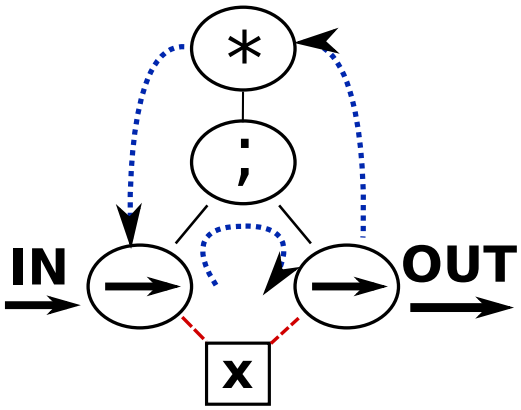


Figure 2.1: Haste Example: Source and Handshake Graph

- 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`)
- conditional expressions (`if` *boolean* `then` *expression*$_1$ `else` *expression*$_2$)
- conditional statements (`if` *boolean* `then` *statement*$_1$ `else` *statement*$_2$)

Figure 2.1 shows the Haste specification of a simple program. 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 `forever do` loop that performs the following actions: (i) read a stimulus 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 2.1. For example, there is a predefined component that implements the `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 compiler maps the variable `x` to a storage element. Finally, the read and write operations, (*e.g.*, read from

channel IN and write to `x`) map to redefined components called *transferrers*, denoted in the Figure by "⟶".

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. Specifications of large systems are naturally decomposed into subsystems or smaller components (*i.e.,* individual procedures).

2.1.2 Performance Limitations

As the number of statements increase in the code snippet in Figure 2.1, the size of the control cycle increases (Figure 2.2), 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. This situation is referred to as "control-dominated."

2.2 Asynchronous Pipelines

To overcome the performance limitation of large control cycles, a designer can pipeline to reduce the control overhead of a block. Rather than a single large control tree, consider 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. Figure 2.2 illustrates how control overhead is reduced in this situation. Small control cycles occur at the computation blocks rather than larger cycles in the initial tree. This transformation is performed using asynchronous pipelines.

Asynchronous pipelines consist of several pipeline stages that communicate via request-acknowledge handshaking signals (Figure 2.3). A stage initiates computation only when it receives new data and a request from its left neighbor. Once data has been accepted (latched), the left neighbor is acknowledged. The stage may then perform operations on the data and forward the results along with a new request to its right neighbor. This behavior is unlike a synchronous pipeline (Figure 2.4), in which signals are received from a global clock to latch data.

Unlike a control-driven approach in which control pushes and pulls data between nodes, the source-to-source compiler produces a data-driven pipeline. In data-driven pipelines, data initiates computation through channel actions. Several datasets can concurrently push through the pipeline, each at a different stage of computation.

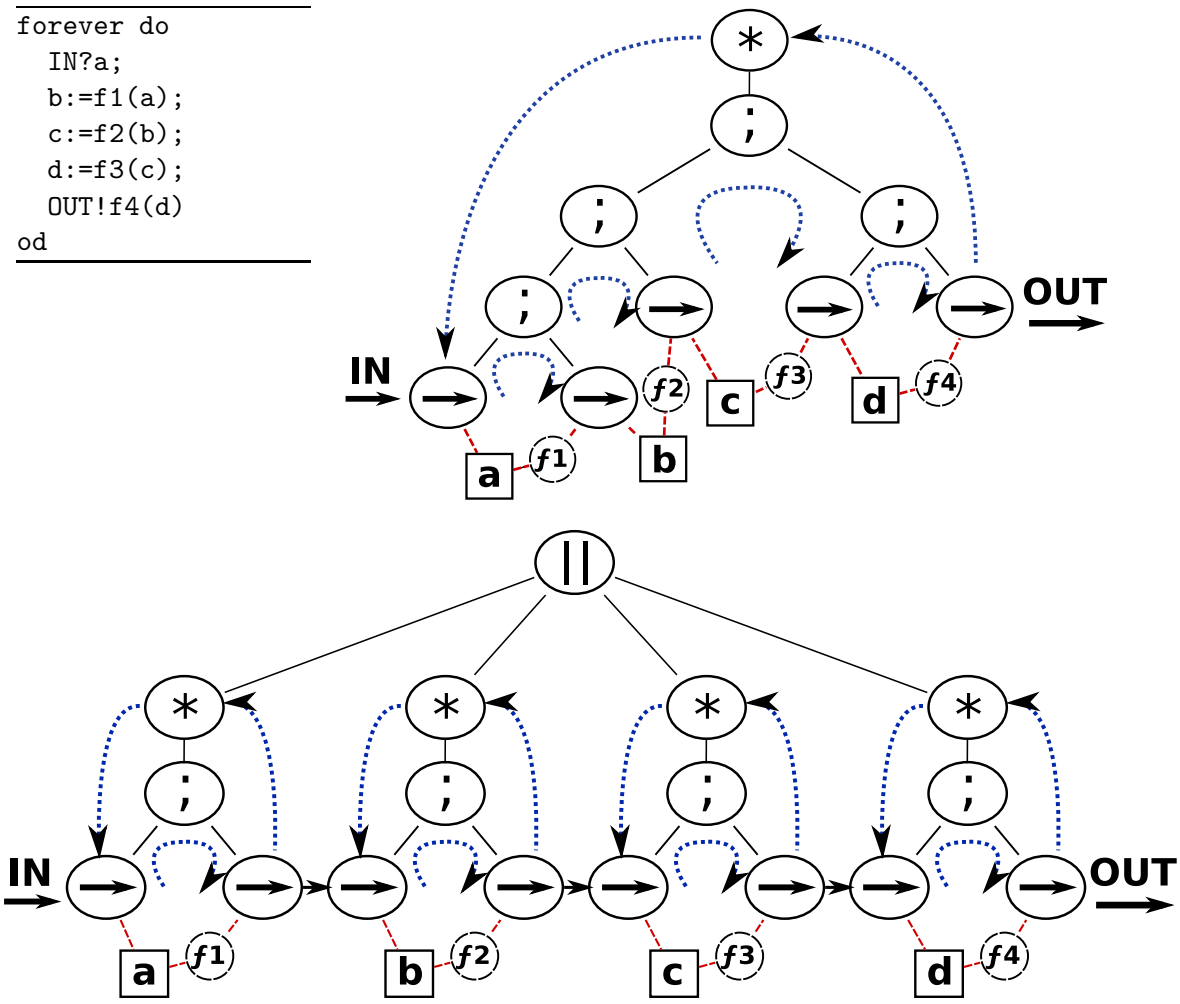


Figure 2.2: Control Dominated vs. Data-Driven Handshake Graphs

In this data-driven implementation, the *context* of the program is communicated between stages. The context consists of all the variables that will be accessed or modified in the remainder of the pipeline. With some exceptions, the data-driven pipelines are linear in nature. The throughput will therefore be limited by the slowest stage in this implementation.

A data-flow architecture, used by Budiu [4] and others, can further increase concurrency by allowing data to propagate only to stages in which it is used. The pipeline is forked off into many different branches in a data-flow architecture. However, performance can suffer if branches are not properly balanced (*i.e.,* not "slack-matched" [2]), resulting in a throughput that may be *worse* than that of the slowest stage.

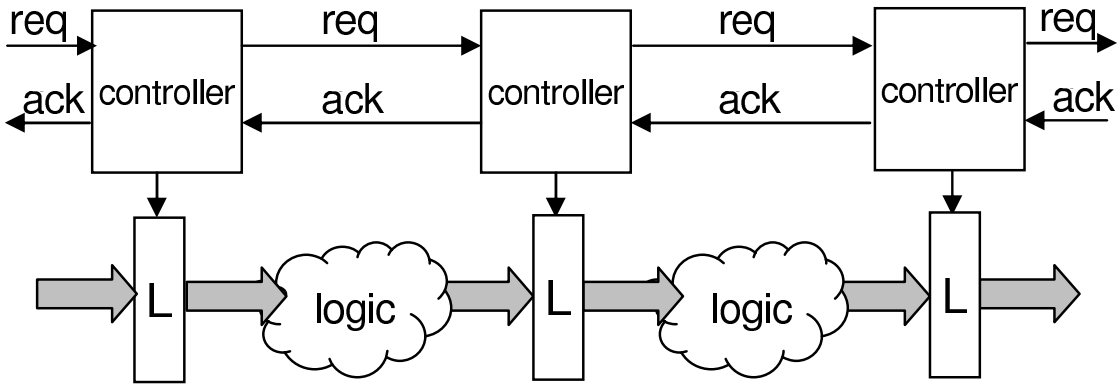


Figure 2.3: Simple Asynchronous Pipeline

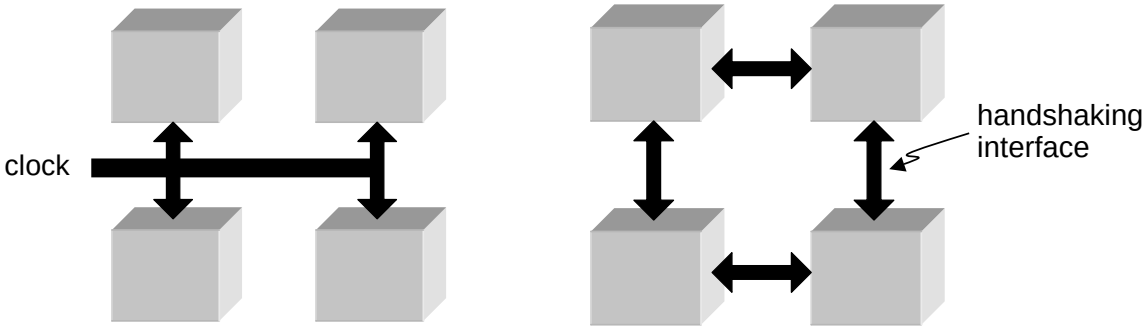


Figure 2.4: Synchronous vs. Asynchronous Module Communication

2.3 Related Work

Budiu et al. [4] 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 approach 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.

Teifel et al. [10] and Wong et al. [11] have introduced approaches that translate specifications written in CHP [9] (a variant of CSP [8]) into pipelined implementations. While these approaches allow channel communication, their communication models can be restrictive: for example, channel actions must be unconditional in [11]. In contrast, this thesis allows a more general communication model, where channel actions are allowed to be conditional as well.

CHAPTER 3

Basic Optimizations

This chapter describes how the proposed compiler optimizes code through parallelization and pipelining. Section 3.1 discusses how performance optimizations change the hardware structure of the system and gives an overview of the optimizations that are performed at a source level. Sections 3.2 and 3.3 then discuss how parallelization and pipelining are performed within the source-to-source compiler.

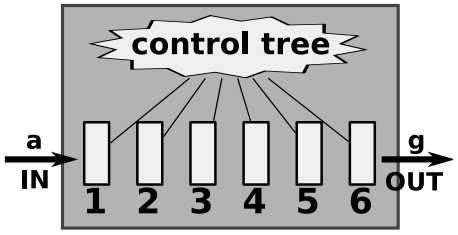
3.1 Method Overview

3.1.1 Hardware-Level Overview

Figure 3.1 shows an example of what synthesized code may look like in hardware. Each small block in the figure represents a basic datapath operation. Similar to the case in Figure 2.2, 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.

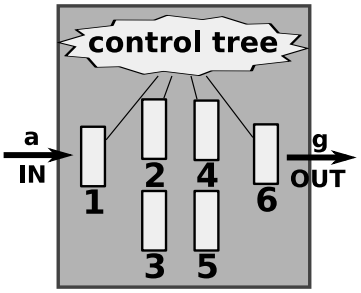
Consider the case where operations in the original code could have been parallelized, but these optimizations were not performed by the designer. By *parallelizing* these operations, the circuit in Figure 3.2 is produced. The resulting circuit is still control driven, and again channel communication 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 full system is reduced, but the throughput is still determined by the latency of the whole system. The system acts as a single stage, with lower latency and higher throughput than that of the previous implementation.

The result of *pipelining* the original implementation is shown in Figure 3.3. Each operation 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 minimized. This data-driven pipeline has multiple, low latency stages, yielding an increase in system throughput. In




```
proc(IN?chan byte & OUT!chan byte).
forever do
    IN?a;
1:  b:=a*2;
2:  c:=b+5;
3:  d:=a+b;
4:  e:=c+d:
5:  f:=d*3;
6:  g:=f+6;
    OUT!g
od
```

Figure 3.1: Original Implementation: Hardware and Source




```
proc(IN?chan byte & OUT!chan byte).
forever do
  IN?a;
  b:=a*2;
  (c:=b+5 ||
   d:=a+b);
  (e:=c+d ||
    f:=d*3);
  g:=f+6;
  OUT!g
od
```

Figure 3.2: Parallel Implementation: Hardware and Source

this case the throughput is limited by the cycle time of the slowest stage, rather than the cycle time of the whole system. Therefore, the throughput is increased, but the latency is likely equal to or greater than that of the original implementation.

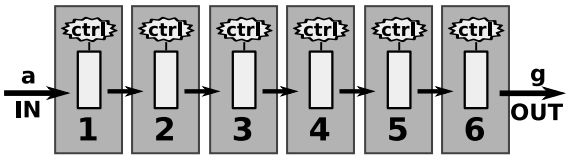
By performing both optimizations, *parallelizing then pipelining,* the circuit shown in Figure 3.4 is produced. This circuit benefits from the reduced latency of parallelization, as well as the increased throughput of pipelining. Conversion to this design is the goal of the compiler's source transformations.

3.1.2 Source-Level Overview

Figure 3.5 is a high-level description of the compiler algorithm. Starting with a piece of straight-line, sequenced code, Figure 3.1, the compiler transforms it into the highly concurrent code of Figure 3.4.

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 3.1, the assignments to variables `c` and `d` can be performed in parallel, and `e` and `f` can be performed in parallel.

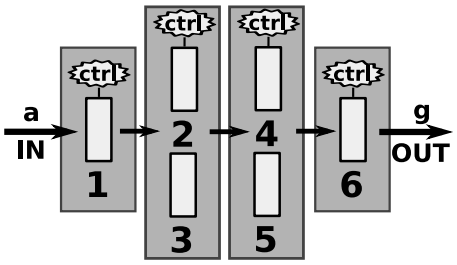
After grouping the parallel blocks, the compiler can place a sequencer between stages




```
stage1(IN?chan byte & OUT!chan byte).
forever do
  IN?a;
  OUT!<<a,a*2>>
od

stage2(IN?chan byte & OUT!chan byte).
forever do
  IN?<<a,b>>;
  OUT!<<a,b,b+5>>
od
...
```

Figure 3.3: Pipelined Implementation: Hardware and Source




```
stage1(IN?chan byte & OUT!chan byte).
  forever do
    IN?a;
    OUT!<<a,a*2>>
  od

stage2(IN?chan byte & OUT!chan byte).
  forever do
    IN?<<a,b>>;
    OUT!<<b+5,a+b>>
  od
...
```

Figure 3.4: Parallelized and Pipelined Implementation: Hardware and Source

(Figure 3.2). This action performs simple instruction-level parallelization, reducing latency. However, performance can be further increased by pipelining.

To pipeline, the compiler places a channel channel communication between every parallel grouping. This channel communicates the context for this dataset. A sample of the pipeline stages for the assignments to `b`, `c`, and `d` are shown in Figure 3.4.

3.2 Parallelizing Transformation

At the core of the parallelization process is dependence analysis. This section briefly describes how dependence analysis is performed using the AST, then shows how the results allow the compiler to modify the program's AST to increase concurrency.

The first step of the process is to traverse the program's AST, maintaining a list of variables within the scope of the current node. At each access to a variable, the compiler creates a link between it and the declaration of the variable. When a statement block is encountered, the compiler generates an array containing all the variables read and written in the statement, as well as a flag that indicates whether the access is a read

```
Source to Source Optimize(Program P){
     Parallelize(P)
     Pipeline(P)
}

Parallelize (Program P){
  for all Statement Blocks in P:
    create a dependence graph of each statement
    perform a topological sort on the dependence graph
    combine grouped statements into a parallel process
    combine statement groupings using a sequencer
}

Pipeline(Program P){
  for all Statement Groups (SG) in P:
    generate IN and OUT sets for the group
    create a module containing the statements in SG
    create channel connectors for the module
}
```

Figure 3.5: Compiler: Optimization Pseudocode

or write. The process is recursive, entering both nested blocks and loops.

Next, the compiler performs dependence analysis on the statements within a block. The compiler first generates a directed graph of the statements. In the graph, edges denote data or control dependencies. Figure 3.6 shows a sample precedence graph. After the graph is generated, a topological sort of the graph is performed. 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. Each parallel grouping then becomes a child of the sequencer (`;`) construct.

Greater concurrency can be achieved if the compiler employs variable renaming. If a second assignment to a variable occurs within a block of code, the location to which the assignment writes is renamed, along with any future accesses. As a result, write-after-read and write-after-write dependencies are removed from the graph.

```
expl=proc(IN?chan byte &
          OUT!chan byte).
begin
  & a,b,c,d,
    e,f,g: var byte
  | forever do
      IN?a;
1:    b:=a*2;
2:    c:=b+5;
3:    d:=a+b;
4:    e:=c+d:
5:    f:=d*3;
6:    g:=f+6;
      OUT!g
    od
end
```

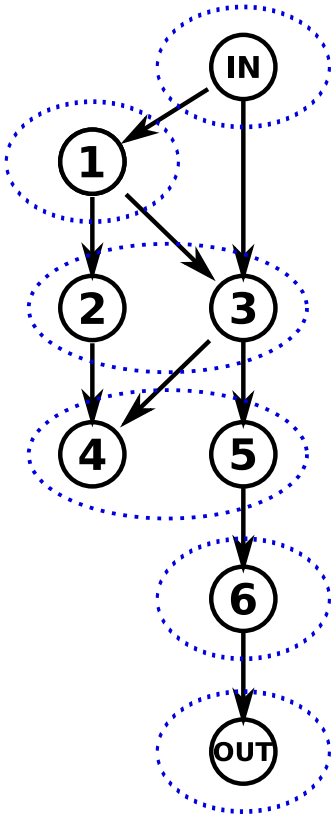


Figure 3.6: Precedence Graph with Parallel Groupings

3.3 Pipelining Transformation

Pipelining is an orthogonal process to parallelization; it can be performed on code that is sequential or has already been parallelized. This thesis only considers feed-forward specifications for the pipelining optimization, *i.e.* specifications that lack feedback between datasets. The approach assumes these specifications read from a set of input channels, perform a computation, and write to a set of output channels. This section discusses how pipelining is achieved for such a specification, in particular, the process of generating channel communications between stages using IN and OUT sets.

To begin the pipelining process, the compiler first breaks every group of statements delimited by a sequencer (;) into its own stage. In source code, each stage will be represented by a procedure definition 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 (VAR_x) by the stage's statement groupings. 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:

$$IN_1 = VAR_x$$

$$IN_x = IN_{x-1} \cup VAR_x \tag{3.1}$$

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):

$$OUT_n = \emptyset$$

$$OUT_x = OUT_{x+1} \cup VAR_{x+1} \tag{3.2}$$

Two important observations can be 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 can be determined. For a stage x, the set of variables in the stage's context is the following:

$$context_x = OUT_{x-1} \cap IN_x \tag{3.3}$$

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

$$context_{x+1} = OUT_x \cap IN_{x+1} \tag{3.4}$$

In the AST for a specification, channel reads for the context 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 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.

CHAPTER 4

Advanced Optimizations

In this chapter, several advanced optimizations are discussed. Section 4.1 describes optimization to conditional constructs. Section 4.2 addresses communication optimization and correctness. Finally, Section 4.3 describes optimizations that can be performed on loops.

4.1 Handling Conditional Communication

Not all code the user wishes to synthesize is linear in nature; conditionals (if-then-else) will frequently exist in the original code. There are many options to handle these breaks in linearity.

4.1.1 Conditional Assignment

If both branches 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:

$$\texttt{var:=if bool then exp}_{then}\ \texttt{else exp}_{else}$$

Consider the code in Figure 4.1. 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:=if a>b then x else x+1
```

If assignments are made in both branches, such as for variable `y`, the same procedure applies:

```
proc(IN?chan byte
     & OUT!chan byte).
begin
  & a,b,x,y
  | forever do
      IN?a;
      IN?b;
      if a>b
         y:=y-1
      else
         x:=x+1;
         y:=y+1;
      fi;
      OUT!x+y
    od
end
```

```
proc(IN?chan byte
     & OUT!chan byte).
begin
  & a,b,x,y
  | forever do
    IN?a;
    IN?b;
    x:=if a>b then x
          else x+1 ||
    y:= if a>b then y-1
          else y+1;
    OUT!x+y
  od
end
```

Figure 4.1: Replacing Conditionals with Conditional Assignments

```
y:= if a>b then y-1 else y+1
```

If the boolean condition is a function of variables modified in either branch, a temporary copy of the boolean must be used to preserve correct operation. Variable renaming may be applied if several writes to the same variable occur in both branches.

4.1.2 Early Decision

Early decision (Figure 4.2) 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.

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. If out-of-order execution is allowable, then no further modifications are needed. However, in many cases a third boolean branch must be introduced between the fork and join stages to indicate which branch the join stage should read from to preserve execution order.

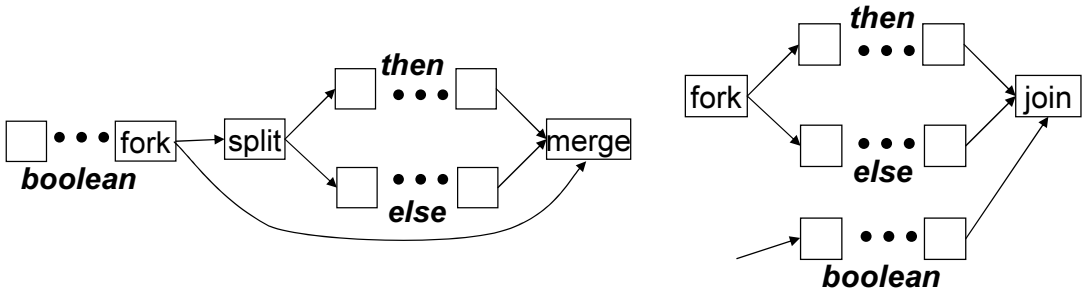


Figure 4.2: Early and Late Decision in Conditionals

4.1.3 Late Decision

Late decision (Figure 4.2) 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, 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.

In late decision, the value of a conditional's boolean is computed in parallel to execution of both branches. For this reason, channel communication is disallowed – channel communications would be initiated regardless of the value of the boolean. 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 significant amount of time to compute. Early decision, in comparison, has the advantage of high throughput even if the paths are not slack matched.

4.2 Preserving Communication Correctness

Maintaining correctness in a specification is a priority over performance enhancement. The existence of channel communication is a difficulty in optimization, because re-ordering sequential communications is generally unsafe in the broadest case. This section enumerates the situations in which channel communications cause difficulty in optimization, and the cases in which performance can be optimized. For these examples, the compiler is presented with a basic module and a black-box with which the module communicates. The compiler is unaware of the statements within the black box.

4.2.1 Correctness Challenges

The first example illustrates the effects of re-ordering a pair of channel communications. For this example, consider a known module that consists of two sequential channel reads: `C?a; D?b`. Its black box counterpart is a module that performs channel writes in the same order `C!x; D!y`. By swapping the channel reads in the known module, a deadlock is introduced: the known module attempts to read from channel `D` while the black-box module attempts to write to `C`. Neither channel communication will terminate.

Instead, the compiler can parallelize the channel reads in the known module. Deadlock is avoided in this scenario. However, no gains in throughput are achieved since the module must wait for both communications to terminate, and these communications are sequenced in the black-box.

The benefit of parallelizing communications occurs when the communications are parallelized either in the black box, or the communications are performed with disjoint modules. If these modules have no channels between them (disjoint), then the two channel actions cannot be externally sequenced. In such a case, it is generally safe to both parallelize and re-order the communications.

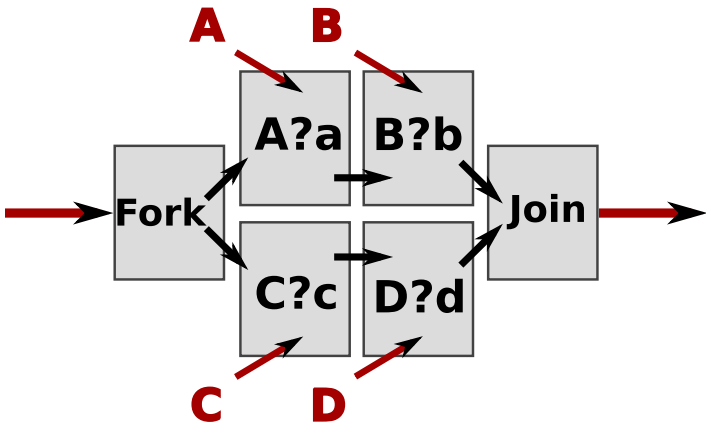
One exception occurs when the initial communication blocks indefinitely. If the known module parallelizes its two channel actions, the second channel communication occurs, even though it should not have occurred in the original specification. Determining whether a communication will block indefinitely is not computable, so the user must help indicate whether parallelization is possible.

Another assumption that must be made for parallelization to occur is that the black-box does not probe its input channels to determine operation. If so, parallelizing two statements can create non-determinism in output.

4.2.2 Solution Overview

The safest solution is to maintain the original ordering of channel operation. Should the compiler have knowledge of the black-box code, or indications from the user, the compiler can perform parallelization on two channels if the following restrictions hold: (i) no data-dependencies between the two channels, (ii) no probes occur causing a selection between the channels, and (iii) both communications will terminate. Furthermore, if the modules to which the channels are connected do not communicate, these channel actions can have their order swapped (this is a loose restriction).

One final example is illustrated in Figure 4.3. In the known module, two parallel



Known	*Black-box*
`(A?a ; B?b )||`	`A!w; B!x;`
`(C?c ; D?d )`	`C!y; D!z`

Figure 4.3: Parallel Sequences of Channel Communications

communication sequences occur. Pipelining the sequence of communications is performed by forking the two parallel streams into separate pipeline branches and merging any changes at their join stage. Had these operations been completely parallelized, the cycle time of the stage would be high due to external sequencing. By performing pipelining on these channel actions, high throughput can be maintained, assuming the black-box is a sequenced pipeline.

4.3 Handling Iterative Computation

The loop pipelining work of [6] discusses how throughput can be conserved in `for` and `while` loops. Loop pipelining can allow several distinct datasets to enter a loop concurrently, increasing throughput in comparison to a single dataset. The optimization is performed using self-timed pipeline rings with a few specialized stages to perform control.

The loop pipelining optimization can be performed at the source level, and can handle arbitrary numbers of nested loops. The examples performed in [6] report a speedup of over 4x when combining loop pipelining with loop unrolling.

In `for` loops when the iteration count is known at compile-time, the compiler can fully unroll the loop to maintain maximum throughput. This optimization comes at a cost of area, and may be a poor choice if the iteration count is high or the internal code is bulky.

CHAPTER 5

Results

This chapter explains how the proposed compiler was tested using the Haste design flow and describes the examples used as benchmarks. Three results tables are included, containing cycle times, latencies, and throughputs of the specifications when simulated at the gate level.

5.1 Experimental Setup

Each example was designed and simulated using the Haste/TiDE toolflow (formerly "Tangram") from Philips/Handshake Solutions [7], described earlier in Chapter 2. Eight different benchmarks were chosen to illustrate the effects of the proposed approach:

(i) SIMPLE: the example of 3.6, consists of straightline code with mixed operators and dependences

(ii) ADD: performs a non-parallelizable sequence of additions

(iii) MULT: performs a highly parallelizable sequence of multiplications

(iv) COND1: performs arithmetic operations using a balanced conditional block

(v) COND2: performs arithmetic operations using a non-balanced conditional block

(vi) TEA: encrypts a binary string given an input key using the Tiny Encryption Algorithm

(vii) ROOT: performs a square root using an iterative loop

(viii) TRIANGLE: computes the area of several triangles using two ROOT loops

The compiler transformed the original specification into three new specifications: parallelized, pipelined, and both parallized and pipelined.

Specification	Throughput (Normalized)			
	Original	Parallel	Pipelined	Parallel+Pipelined
SIMPLE	1.0	1.1	1.3	1.3
ADD	1.0	1.0	7.8	8.0
MULT	1.0	3.4	4.6	4.6
COND1	1.0	1.0	2.9	2.9
COND2	1.0	1.0	1.5	1.5
TEA	1.0	1.0	23.2	23.2
ROOT	1.0	1.0	0.97	1.1
TRIANGLE	1.0	1.0	6.6	6.6

Table 5.1: Simulation Results: Throughput

Specification	Latency(ns)			
	Original	Parallel	Pipelined	Parallel+Pipelined
SIMPLE	71.7	64.9	122.8	118.2
ADD	36.8	36.8	43.0	43.0
MULT	2235.5	654.3	4907.0	1481.3
COND1	23.5	23.5	28.8	29.0
COND2	12.2	12.2	28.8	29.0
TEA	341.3	341.3	349.8	349.1
ROOT	138.7	123.3	571.1	260.0
TRIANGLE	1226.2	510.8	1446.3	709.2

Table 5.2: Simulation Results: Latency

5.2 Results and Discussion

The results in Table 5.1-5.3 demonstrate the benefit of the proposed transformation approach. When automatic parallelization of the specification is performed, the performance improves in those examples where individual statements can be parallelized (e.g., MULT and TRIANGLE). If pipelining transformation were performed, there is a quite substantial improvement in the overall throughput for all examples except for ROOT, which has a data-dependent loop that prevents benefits of pipelining from being realized. Finally, the last column shows results of applying both parallelization and pipelining, resulting in even higher performance gains: from factor of 1.1x to 23x higher throughput.

Furthermore, latency reduction of up to factor of 72% was observed (TRIANGLE); this example has two loop constructs which are parallelized, thereby obtaining significant latency reduction.

	Cycle Time(ns)			
Specification	Original	Parallel	Pipelined	Parallel+Pipelined
SIMPLE	71.7	64.9	53.9	53.8
ADD	36.8	36.8	4.7	4.6
MULT	2235.5	654.3	490.8	490.8
COND1	23.5	23.5	8.0	8.0
COND2	12.2	12.2	8.0	8.0
TEA	341.3	341.3	14.7	14.7
ROOT	138.7	123.3	143.3	130.2
TRIANGLE	1226.2	510.8	186.9	186.9

Table 5.3: Simulation Results: Cycle Times

CHAPTER 6

Conclusions and Future Work

This thesis proposed the use of a source-to-source compiler to increase the performance of a specification while maintaining ease of design . The results show improved throughput for all pipelined specifications (up to 23x) and latency improvement for parallelized specifications (a 2.4x decrease).

In ongoing work, I will evaluate the compiler's performance by using it on more complex systems, such as multimedia streaming processors. I aim to implement a wider variety of conditionals: including variants on early and late evaluation. I also plan to provide more robust communication support by allowing more communications to be parallelized and re-ordered.

In addition, I plan to leverage the loop pipelining approach of [6] in order to provide better optimizations for both `while` and for `loops`. Finally, I plan to perform a full data-flow implementation as an alternative to the data-driven approach. With a high-quality slack-matching heuristic, a dataflow architecture may be able to enhance performance even further.

BIBLIOGRAPHY

[1] Int. Technology Roadmap for Semiconductors. Overall Roadmap Technology Characteristics. `http://public.itrs.net`.

[2] Peter A. Beerel, Nam-Hoon Kim, Andrew Lines, and Mike Davies. Slack matching asynchronous designs. In *ASYNC '06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, page 184, Washington, DC, USA, 2006. IEEE Computer Society.

[3] C. H. (Kees) van Berkel, Mark B. Josephs, and Steven M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, February 1999.

[4] Mihai Budiu. *Spatial Computation*. PhD thesis, Carnegie Mellon University, Computer Science Department, December 2003. Technical report CMU-CS-03-217.

[5] Barbara Chappell. The fine art of IC design. *IEEE Spectrum*, 36(7):30–34, July 1999.

[6] Gennette Gill, John Hansen, and Montek Singh. Loop pipelining for high-throughput stream computation using self-timed rings. In *Proc. Int. Conf. Computer-Aided Design (ICCAD)*, pages 289–296, 2006.

[7] The Haste/TiDE Design Flow. `http://www.handshakesolutions.com`.

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[9] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Pénzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[10] John Teifel and Rajit Manohar. Static tokens: Using dataflow to automate concurrent pipeline synthesis. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 17–27. IEEE Computer Society Press, April 2004.

[11] Catherine G. Wong and Alain J. Martin. Data-driven process decomposition for the synthesis of asynchronous circuits. In *IEEE Int. Conf. on Electronics, Circuits and Systems*, 2001.