CASH: A C to Layout Compiler

1 Introduction

In this paper we present CASH, a Compiler for Application Specific Hardware. CASH performs high-level synthesis, taking as input ANSI C programs, generating structural Verilog as output. The goal of this paper is to describe the research contributions incorporated in CASH, which advance the techniques of high-level synthesis in several directions.

From the outset we set the goal for CASH to support ANSI C programs in their original form, enabling software programmers to create hardware devices. Another goal for CASH is to be able to exploit the parallelism available in the C programs. The granularity of parallelism extracted by CASH is instruction-level parallelism (ILP). Techniques for extracting ILP have been a subject of intense research and implementation in the last 40 years in the compiler domain. ILP is effectively leveraged by the modern superscalar and VLIW microprocessors. CASH incorporates many techniques from traditional software compilers and uses them to increase the parallelism of the synthesized circuits. Some of these techniques have been used before in the context of high-level synthesis, but CASH combines them together in novel ways.

The following are important research contributions present in CASH:

1. CASH uses a target-independent intermediate representation, called Pegasus, to represent and optimize the program. Pegasus is a unique representation because it manages to blend the conflicting goals of compilation and synthesis, by being simultaneously a compiler representation and an RTL-level representation. No other compiler that we are aware of manages to successfully satisfy these two goals.

   (a) Pegasus is an RTL-level representation, allowing us to leverage 30 years of experience in circuit synthesis. The RTL level of Pegasus makes translation of Pegasus to circuits modular, a simple term-rewriting operation. Pegasus should be contrasted with the Control-Dataflow Graph representations employed in other high-level synthesis tool-chains, which are not RTL-level representations.

   (b) Pegasus incorporates many features of modern software compiler representations, embedding SSA, predication and speculation. We briefly describe the importance of these below.

2. CASH incorporates many modern compiler technologies for handling all C constructs in order to extract ILP from sequential C code. These techniques include Static Single Assignment (SSA), predication, and speculation. CASH leverages these techniques to increase the amount of ILP effectively and to simplify the synthesis process:

   (a) Pegasus is an SSA intermediate form. SSA is a program structure in which every variable is assigned by a single instruction in the program. A variable which is written only once is synthesized by a storage structure in hardware with a single write-enable signal which requires no arbitration. It can thus be stored in a simple latch, or a FIFO if used within a loop. Such an implementation allows the circuits to achieve a very high throughput at a very small area, and simplifies the control circuits.

(b) Speculation refers to the execution of some statements even if their result may be unneeded. Speculation allows CASH to transform conditionals (`if` statements) into multiplexors. The main advantage of speculation is to enable some computations to be performed earlier, thus increasing the parallelism available in the program: instead of executing sequentially `a` followed by `b`, speculation enables the concurrent execution of `a` and `b`. Figure 9 shows an instance of the application of speculation transformation in CASH. After speculation the evaluation of the branch condition `if` and expressions in the `then` and `else` blocks are evaluated all in parallel. Cash uses speculation to group together many small basic blocks into larger hyperblocks.

As we describe in Section 2.5, the control signals used for the multiplexors that select the speculative results are one-hot encoded. This seemingly minor point turns out to be extremely important for enabling many efficient optimizations, and for allowing fast evaluation of the multiplexors at execution (see Section 3.2.4).

(c) Predication attaches to some operations predicates, allowing their effect to be ignored when the predicate is false. CASH attaches such predicates to speculated operations which may have undesirable side-effects otherwise. For example, a memory write should never be executed speculatively, so a predicate is used to cancel it when it was incorrectly speculated. This enables CASH to perform speculation across large code blocks, without being concerned about side-effects.

3. CASH handles procedure calls, recursion, `for` and `while` loops, unstructured control-flow (i.e., built with arbitrary `goto` statements) and dynamic software pipelining.

   With the abundance of transistors available, we have chosen in CASH a new trade-off for the layout of the synthesized circuits: a complete spatial layout. The program datapath has each static program machine instruction mapped to a separate synthesized functional unit.

   Interestingly enough, we have chosen a spatial mapping as well for the control path. Control is thus completely distributed, one-hot-encoded. Instead of encoding control in finite-state machines sequencers, CASH chooses to use a separate state element (latch) to explicitly control the locus of control of each major program point (hyperblock). CASH uses just three simple primitive "routers" to steer control, (plus the multiplexor). These "routers" are described in Section 2.6. This style of control is related to the control synthesized by some asynchronous circuit synthesis systems such as Tangram and Balsa [], and to the logical distributed control of classic dataflow machines [27].

   Two reasons compelled us to build control in this way:

   (a) Because of the spatial nature of the datapath, the control should be distributed, co-located with the computation (and not centralized).

   (b) This type of control performs dynamic scheduling of the program statements, enabling the program operations to execute out-of-order, based just on the availability of their data inputs. Thus, the circuits synthesized by CASH perform well when confronted with unpredictable memory and network delays. The dynamic behavior of the circuits synthesized by CASH is quite similar to the behavior of a superscalar processor. We were inspired by the resilience of these processors to blocking events in designing the distributed control method synthesized by CASH.

4. CASH handles all the complex ANSI C data structures: arbitrary pointers, arrays, unions and structures. Handling C data structures is especially hard in combination with program parallelism, because the compiler has to ensure *memory coherence* in the presence of concurrent memory accesses.

   For this purpose the CASH compiler syntheses for each program a custom on-chip data packet memory access network (MAN). This network has three separate roles:

   (a) It sends read and write requests to memory out-of-order (with respect to the original program order);

   (b) It receives read replies with data from memory out-of-order;

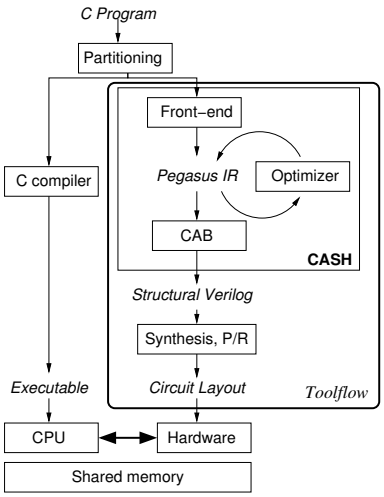

Figure 1: *The CAD toolflow. The box named* Toolflow *marks the focus of this paper.*

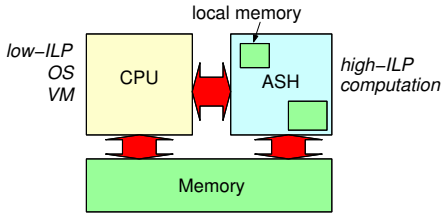

Figure 2: *ASH used in tandem with a processor for implementing whole applications. The processor is relegated to low-ILP program fragments and for executing the operating system.*

(c) It transports coherence traffic in the form of *tokens*, which ensure the memory coherence, even when the program statements execute out-of-order, and even when the memory system replies to requests out-of-order.

The memory and coherence traffic access network is a fundamental research contribution of the CASH compiler. Irrespective of whether C is a viable language for high-level synthesis, the principles and techniques embodied in the MAN for ensuring coherence in hardware for distributed computing structures can be applied in many other settings.

5. Currently CASH has a back-end which generates asynchronous circuits from Pegasus. To our knowledge, CASH is the only C to asynchronous circuits high-level synthesis tool-chain in existence. In principle, is should be easy to retarget CASH to generate synchronous circuits as well starting from Pegasus.

1.1 Toolflow Overview

Figure 1 represents the high-level view of our compilation system. The focus of this paper is the box labeled *Toolflow*, which encapsulates the transformations that produce asynchronous hardware from ANSI-C. The core of this system is CASH, which compiles ANSI-C programs into structural Verilog.

The translation is achieved in two steps: (1) C is converted into a simple intermediate representation called Pegasus; the program in this form is optimized using traditional compiler optimizations, and (2) the Pegasus constructs are synthesized by the CAB back-end (CASH Asynchronous Back-end) into pipelined asynchronous circuits. The output is a Verilog program, which is synthesized, placed and routed using commercial CAD tools.

Typical C programs rely on library functions for implementing input-output operations; these library functions at some point invoke operating system services through system calls. Since we translate our programs to a hardware target, CASH cannot synthesize system calls. Given the source code of library functions, CASH can synthesize it as any standard C code; for example the implementation of dynamic memory allocation code `malloc`, `free`, etc. can be synthesized in this way. In this paper we assume that the application that is compiled is first subjected to a process of hardware-software partitioning, and that the part of the application which requires operating system services is mapped to a traditional processor, while the rest is mapped to hardware, as shown in Figure 2. The unit of partitioning is the C function: each function is implemented either entirely in hardware or in software. The processor and the hardware have access to the same global memory space, and there is an underlying mechanism that maintains a coherent view of memory (e.g., cache coherency). Partitioned programs need to cross the hardware-software boundary, invoking services on the other side. We provide an implementation of cross-boundary calls, which resembles remote procedure calls (RPC) [67]. Similar to traditional RPC, we have built a stub compiler which can automatically and transparently synthesize the constructs needed to pass control and data across the hardware-software boundary [9].

1.2 Paper Overview

In the rest of this paper we focus on the box labeled "Toolflow", assuming that we are compiling pure ANSI C functions (i.e., which are not making system or library calls). The next section is devoted to a description of the front-end and optimizer of the CASH compiler. Section 3.1 and Section 3.2 discuss the nature of the asynchronous circuits generated by CASH. Section 2.7 is about the Memory Access Network. Section 3.3 describes the asynchronous circuits back-end of CASH, CAB. After related work we present a series of experiments in Section **??**.

2 Compiling C to Pegasus

2.1 Overview

CASH has a C front-end, based on the Suif 1 compiler [98], which performs parsing and a few optimizations. Then, the front-end translates the Suif intermediate representation into Pegasus. Most of the program optimizations performed by CASH are done on Pegasus. The optimizations include scalar-, memory- and Boolean optimizations. The CAB back-end performs some target-specific peephole optimizations, and generates structural Verilog, which is synthesis-ready.

2.2 The Suif Front-End

CASH uses the Suif 1 research compiler from Stanford University for its front-end. Suif is used for parsing C into a traditional control-flow graph (CFG) compiler representation.

CASH performs a series of simple optimizations and analyses on this representation, illustrated in Figure 3: procedure inlining, loop unrolling, intra-procedural pointer analysis, basic control-flow optimizations, call-graph computation, live variable analysis. The final phase before translating the CFG representation to Pegasus is to compute path predicates and to group basic blocks (sequences of program instructions which are not interrupted by branches) into hyperblocks.

A hyperblock is a portion of the program having a single control entry point and possibly multiple control exits, as shown in Figure 4. Hyperblocks were introduced in compilers for microprocessors that rely on predicated execution [60], such as Itanium. As we describe in the next section, the code in each hyperblock is executed speculatively, to increase the amount of parallelism. The intuition is that once control has entered a hyperblock all instructions inside the hyperblock are executed. Predication is used to prevent some instructions to have undesirable side-effects. The "arrows" inside hyperblocks are thus "erased", as shown in the right side of Figure 4.

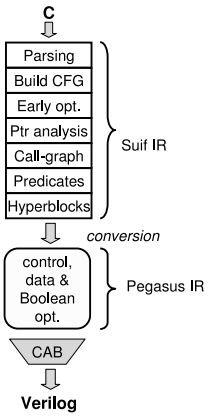

Figure 3: *Structure of the Suif front-end.*

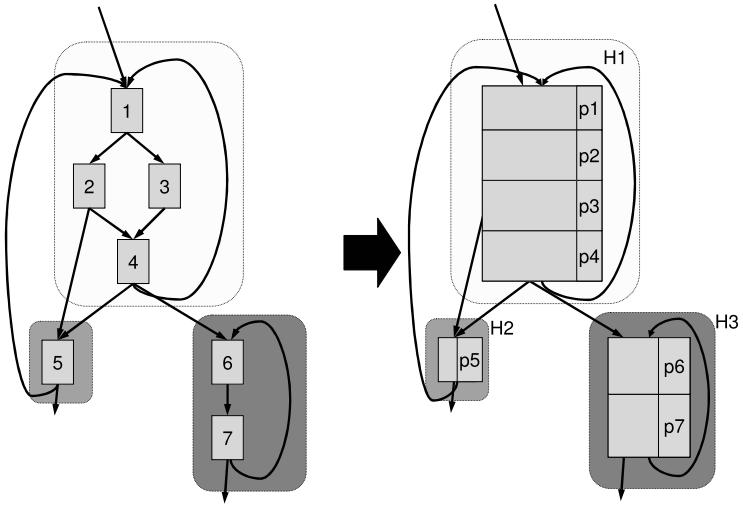

Figure 4: *Program hyperblocks are sets of basic blocks such that there is a single control-flow entry point, but possibly multiple exit points. This figure illustrates how a program is decomposed into three hyperblocks $H1$, $H2$, $H3$. The instructions in each hyperblocks are tagged with Boolean formulas path predicates $p1 \ldots p7$ indicating the basic block where they originate. Figure 5 shows how the path predicates are computed.*

Each basic block in the hyperblock is tagged with a *path predicate* formula, which summarizes how it is reachable from the entry point of the hyperblock. Figure 5 illustrates how path predicates are computed. This algorithm described in the Predicated Static Single Assigmnent paper [13], so we do not elaborate it further here. The path predicates are used in multiplexor synthesis (Section 2.5) and for predicating non-speculative operations (Section 2.7). Figure 4 shows the three hyperblocks $H1$, $H2$, $H3$, containing the original basic blocks $1 \ldots 7$, tagged with the path predicates $p1 \ldots p7$.

Figure 6 shows on the left a simple C code fragment; the CFG representation, containing 3 basic blocks, is shown in the middle. On the right is shown the hyperblock representation. There is a single hyperblock, containing all three instructions. The three path predicates for the three instructions are indicated. In Figure 9 we show how this hyperblock is translated to the Pegasus representation.

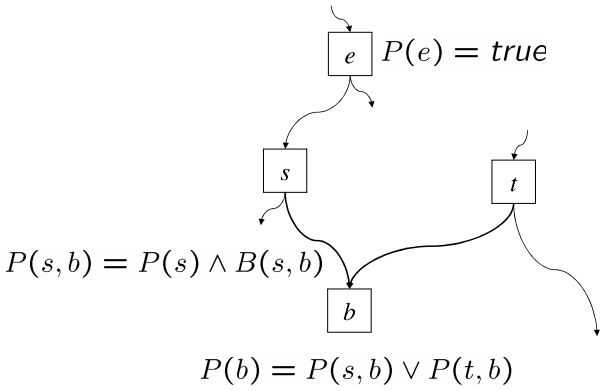

Figure 5: *The computation of the path predicates is done for each hyperblock, in topological order, ignoring the loop CFG edges. Each basic block x has an associated predicate $P(x)$, and each branch $x \rightarrow y$ has a predicate $P(x, y)$. Block s ends with a conditional branch to block b; $B(s, b)$ is the branch condition predicate, computed by the code in block s.*

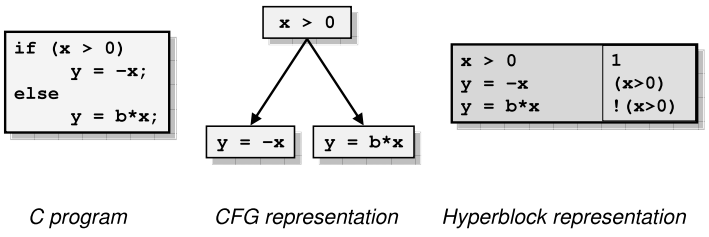

Figure 6: *(left) Sample C program fragment. (middle) CFG representation comprising 3 basic blocks. (right) Hyperblock representation containing a single hyperblock. On the right we show the path predicates for each instruction.*

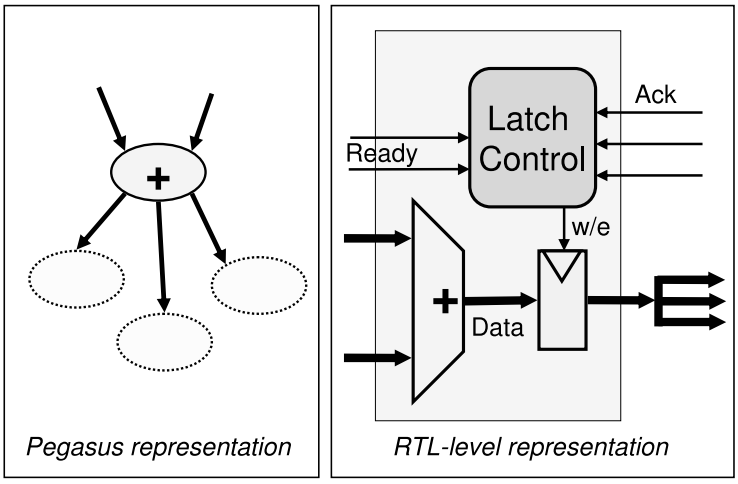

Figure 7: *(left) Pegasus representation and (right) traditional RTL-level representation of an adder operation with a fanout of three. A private latch is attached to the adder. The latch has a single writer, but multiple readers (three in this example). There is a private controller attached to each latch which prevents the master from overwriting the data before all its readers have accessed it. This figure only shows the control signals that affect the latch; there are other control signals synthesized, as shown in Section* **??**.

2.3 The Pegasus Intermediate Representation

The keystone of CASH is its intermediate representation, Pegasus. A Pegasus program is a directed graph: nodes are operations and edges show the flow of values. We believe that one significant strength of Pegasus is that it blends successfully features of two complementary worlds:

- Pegasus is a modern (software) optimizing compiler representation. It is built on top of SSA, predication and speculation.
- Pegasus is focused towards circuit synthesis. It is essentially an RTL-level circuit representation. Each Pegasus operator has a very precise and simple semantics (meaning). The Pegasus operators behave like RTL modules; their meaning is independent of the context where they are used in the program. The meaning of the whole compiled program is a composition of Pegasus operators.

This last fact is crucial, because it allows synthesis to translate each operator independently *without concern about the global program structure*. It is worth pointing out that not all intermediate compiler representations have such a modular representation, for example the CDFG representation is not an RTL representation. The compositionality of Pegasus ensures that if each operation is correctly synthesized to implement the specified semantics, the ensemble of operations correctly implements the original program. This implies that the back-end process does not need to analyze or synthesize any global structures, and thus synthesis can be performed as local term-rewriting. We discuss features of Pegasus in this section, and the translation to asynchronous pipeline stages in Section 3.3.

2.4 Scalar Variable Synthesis

Pegasus is a form of intermediate representation called Static Single Assignment (SSA) [22]. Many papers published in the last 15 years have shown that SSA dramatically simplifies many compiler algorithms. The

For each basic block b in topological order
    For each variable v live at the entry of b
        Create a mux $m(v)$
        def$(b, v) = m(v)$
        For each predecessor b' of b
            Connect *def*(b', v) to $m(v)$
            Set corresponding control predicate to $p(b', b)$
    For each instruction i in b
        If v is referred as a source in i, add edge *def*(b, v) to i
        If v is referred as a destination in i, *def*$(b, v) = i$

Figure 8: Algorithm for inserting multiplexors.

essential feature of SSA is that each variable is assigned to only once. All scalar variable operations (i.e., non-arrays) in Pegasus are represented in SSA form. A program is brought to SSA form by renaming variables: if there are multiple operations writing to the same variable, a temporary variable is used in some of these operations, and a multiplexor is used to assign to the final variable at the very end. The cited paper explains in detail how this is done in traditional compilers, and we will not reproduce the details here.

Traditional compilers use SSA because it simplifies optimizations. However, we have discovered that SSA has definite advantages from a hardware synthesis point of view as well, when used in a spatial computation context. In this model, each SSA variable has a single function unit which assigns to it. In consequence, each scalar variable will have exactly one writer in the whole program, the functional unit computing its final value. There will thus be exactly one "write enable" signal for each variable in the program, and no arbitration signal will be required. Very simple control and storage is required for handling all scalar variables.

Each Pegasus operation has an implicit associated latch that is used to store the value computed during program execution. The actual control signals are synthesized by the back-end. The high-level structure is shown in Figure 7. Each latch has a single writer, and many readers. By using a def-use analysis the compiler can determine all the readers of each scalar variable. Thus, CASH has complete knowledge about all the readers of a latch, and it synthesizes control circuitry which alternates access to the latch in a write-read-write-read cycle. The control circuit will prevent the latch from being overwritten before all the readers have accessed its content.

Variables whose address is taken and which cannot be disambiguated by the pointer analysis are transformed into pointer accesses (i.e. instead of `v=2` CASH generates code for `*(&v=2)`. Pointer handling is described in Section 2.7.

2.5 Multiplexor Synthesis

After constructing hyperblocks, the next step in building Pegasus is to connect variable definitions to their uses. When a variable is assigned in multiple basic blocks CASH must use a mux to merge the multiple definitions. The mux insertion algorithm maintains for each basic block b and each live variable v a data astructure "last definition point" *def*(b, v). (A variable is live if it is further used in the program; variable liveness is computed by the Suif front-end.) Figure 8 shows the algorithm used by CASH to insert multiplexors.

Figure 9 shows how the code in Figure 6 is represented in Pegasus after mux synthesis has been performed. A 2:1 decoded MUX is used to select only the correct result. The 2 predicates driving the MUX correspond to the two path predicates `(x<0)` and `!(x<0)`. Predicate values (i.e., Booleans) are always shown with dotted lines. Speculation enables all three arithmetic operations (comparison, negation and multiplication) to execute simultaneously (assuming inputs are available all at once). In a traditional implementation, the comparison `(x<0)` would have been on the critical path, since it is used to decide which of the other two operations should fire. Note that MUX nodes are effectively *speculation squashing points*, discarding all of their data inputs corresponding to *false* predicates.

```
if (x > 0)
   y = -x;
else
   y = b*x;
```

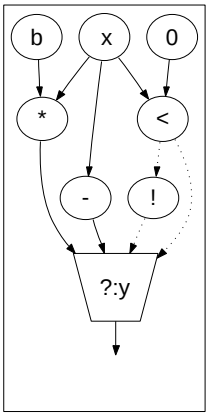

Figure 9: *Final Pegasus representation of the code in Figure 6 after multiplexor synthesis.*

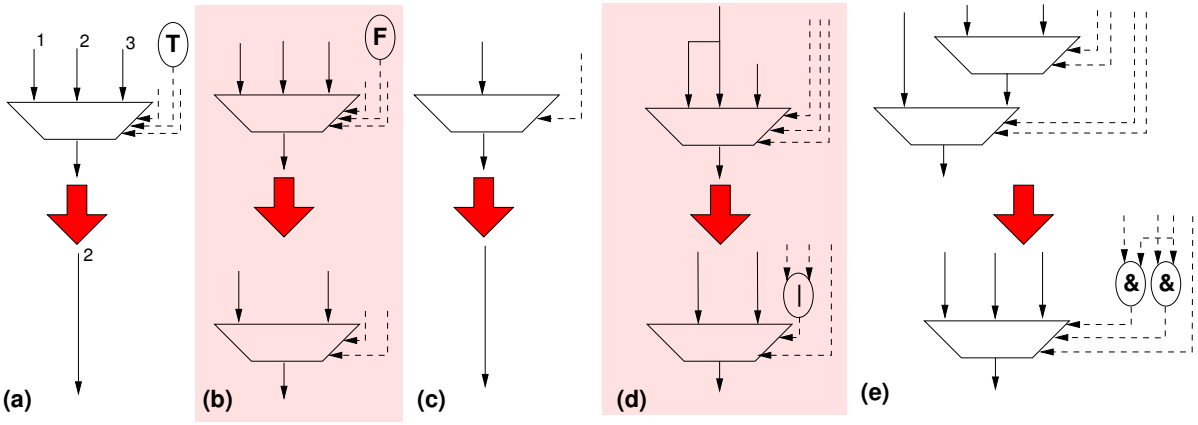

Figure 10: *Decoded mux optimizations performed by CASH: (a) a mux with constant "true" predicate is replaced with the corresponding data input; (b) mux with constant "false" predicate has the corresponding data input removed; (c) mux with only one input is short-circuited; (d) mux with two identical data inputs merges the inputs and "or"s the predicates; (e) chained muxes.*

A very important point to note is that CASH uses throughout decoded muxes. A decoded mux has n data inputs and n control inputs, i.e., control is one-hot-encoded. It is an invariant of the circuit structure that controls of a mux are mutually exclusive. However, the controls do *not* sum up to "true". A mux that has all control inputs "false" should emit an arbitrary value at the output. The muxes synthesized by the CAB back-end are also fully decoded.

There are many advantages to using decoded muxes:

- There is no need for a mux to decode the control signals, making them faster.
- The muxes are completely symmetric with respect to all their control and data signals. Since the control signals do not sum up to logical "true", the compiler does not have to decide which control signals to use to drive the encoder. This is especially important for high-fanin muxes.
- As we show in Section 3.2.4, decoded multiplexors can sometimes produce outputs before all their inputs are available. This fact can be used to produce circuits which compute faster.

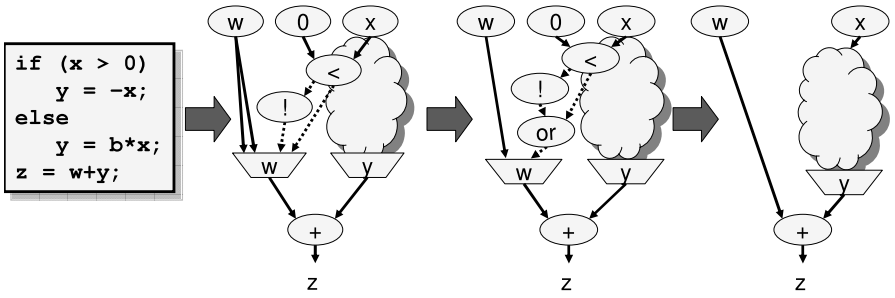

Figure 11: *Mux optimization applied to a simple code example involving conditionals. The cloud hides the Pegasus code computing the value of* `y`*, which is shown in Figure 9. The variable* `w` *is live across two conditional branches, but defined in neither. This gives rise to a mux with two identical definitions. The optimizations from Figure 10 are applied in sequence: case (d) merges two identical inputs, case (c) removes the useless mux.*

- The decoded muxes make many interesting optimizations much simpler.

Figure 10 shows a set of mux optimizations performed by CASH. In Pegasus these are just simple term-rewriting operations. Cascading decoded muxes and removing constant Boolean controls are very simple operations to perform for decoded muxes, but much harder to do for encoded muxes.

Because CASH synthesizes an abundance of muxes, one for each live variable at each basic block boundary, the mux optimizations described above are particularly important to simplify the circuit. Figure 11 shows an instance application of these optimizations for a small code example. Optimization (a), is not applied in this example, but is also very useful in practice, since nested conditionals give rise to chained muxes.

Note that the path predicates are used as control inputs to the MUX operators. Using the predicates only at the muxes, and executing the operations speculatively is a technique that has been employed before by some compilers for microprocessors under the name "predicate promotion" [59]. It is a very effective means to increase the program parallelism, but it may increase the critical path of the program as well, if the computations in the branches of the mux are unbalanced in latency. We describe in Section 3.2.4 a method which reduces the dynamic critical path: MUX and Boolean operations are synthesized in a style called "lenient", which allows them to output data early if they have enough knowledge at run-time.

2.6 Control-Flow Synthesis

Unlike traditional CDFG representations, Pegasus is a pure Dataflow representation. Pegasus has no control-flow constructs (i.e. jumps), and represents control passing as pure data-flow operators, which behave similarly to arithmetic operations. In this section we explain how the translation from control-flow to data-flow is performed.

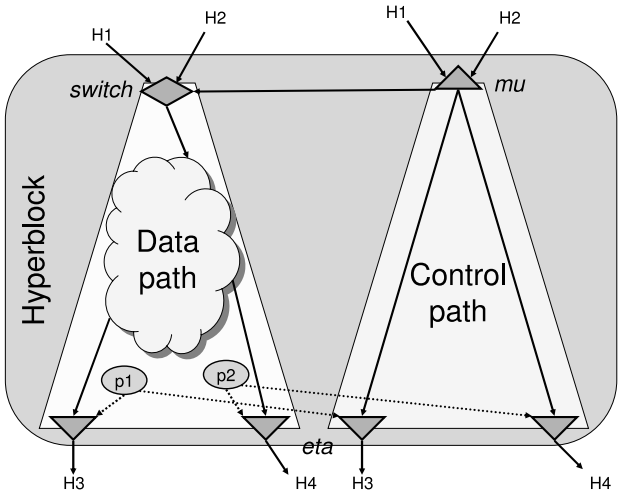

Figure 12: *The high-level structure of the control-flow Pegasus circuits for a single hyperblock. The control path is responsible for maintaining the current locus of control for the program. The data path is responsible for ensuring the correct flow of data values. The data path also computes the predicate values (i.e., branch conditions) that are responsible for steering control.*

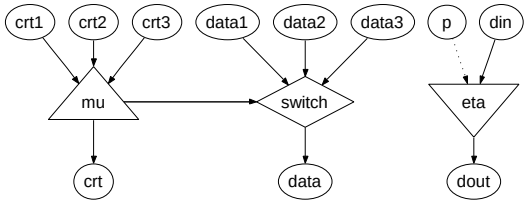

Figure 13: *Control-flow operations:* MU, SWITCH, ETA. MU *and* SWITCH *are roughly equivalent to multiplexors.* MU *operations are used on the control-path, and* SWITCH *operations are used on the data path, at hyperblock entry points. The* ETA *operations are used to gate the flow of a value based on the value of a predicate.* ETA *operations are used both on the control- and data-paths, at hyperblock exit points.*

Control-flow within a hyperblock is replaced by MUX operations, as explained above, in Section 2.5. When the execution of a hyperblock terminates, control must be passed to a successor hyperblock. Figure 12 shows the high-level structure of the Pegasus circuits used to steer data and control between hyperblocks.

2.6.1 The Control Path

The control path is used to represent the currently executing hyperblock. The collection of all control-path MU nodes in the program is used to one-hot encode the current execution point of the program.

The control path is always implemented with just two Pegasus primitive operations, MU and ETA, which are used to "steer" the flow of control. These are depicted in Figure 13.

MU nodes (up triangles) correspond roughly to *labels* in machine code: they indicate targets of control-flow. A MU has n inputs and two outputs. At any moment at most one of the n inputs is available. The semantics of MU is very simple: the available input is moved to the first output. The second output indicates the *number* of the input that last received data.

ETA nodes (down triangles) correspond to branches. They are used to steer data out of a hyperblock. An ETA has two inputs — one for data and one for a predicate — and one output. An ETA forwards the data to the

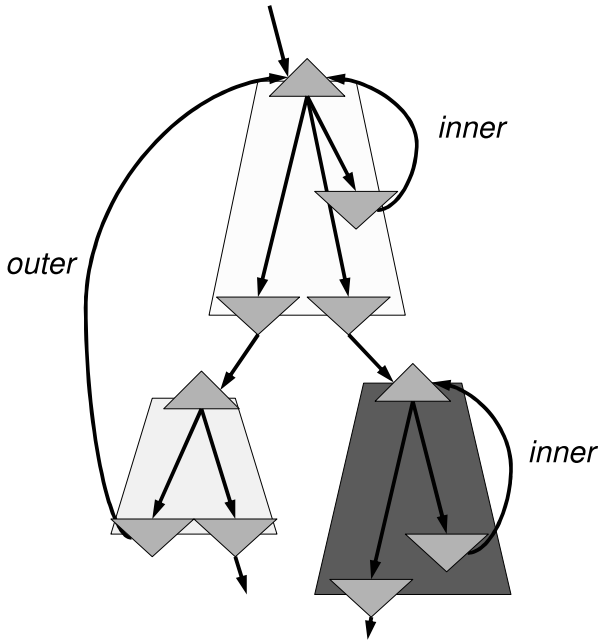

Figure 14: *Control-path synthesized for the program structure in Figure 4. The three loop edges are highlighted.*

output if the predicate is *true*; if the predicate is *false*, the data is consumed and nothing is forwarded. In general, hyperblocks have more than two exits, and the ETA predicates correspond to a one-hot encoding of the branch condition steering control out of the current hyperblock.

In Figure 12 the predicates `p1` and `p2` are guaranteed to be mutually disjoint and complementary. The control path receives control through the top MU node from either hyperblock H1 or H2. Control is passed to both ETA nodes. The data path computes predicates `p1` and `p2`. Let's say `p1` is true and `p2` is false. Then the left ETA in the control path will be opened, passing the control to the next hyperblock H3.

2.6.2 The Data Path

Here we present the operators used to steer data values between hyperblocks. To receive data into a hyperblock Pegasus uses SWITCH nodes. A SWITCH node is very similar to an encoded multiplexor. A SWITCH has n data inputs and one data output, and a control input which indicates which of the data inputs is copied to the output. However, the SWITCH nodes differ from a mux in that they only expect one data input to be present. The control input of the SWITCH node is always connected to the control output of a MU node in the control path in the same hyperblock, as shown in Figure 12. A data path will have many SWITCH nodes at the input, one for each live variable at the hyperblock entry point.

To steer data out of the hyperblock ETA nodes are used. There is an ETA node for each live variable and possible destination hyperblock. In Figure 12 the datapath sends one live variable to H3 and one to H4.

2.6.3 Synthesizing Loops

A loop is represented in the C program CFG by a branch going backwards. After hyperblock decomposition there will be two types of loops: inner loops and outer loops. Most inner loops will reside within a hyperblock; see for example in Figure 4 loops (1,2,3,4) — residing in H1, and (6,7) — residing in H3. Outer loops will be represented as inter-hyperblock control-flow transfers, such as the edge $5 \rightarrow 1$ in the same figure.

```
int squares(int n)
{
  int i = 0,
      sum = 0;

  for (i=0;i<n;i++)
      sum += i*i;
  return sum;
}
```

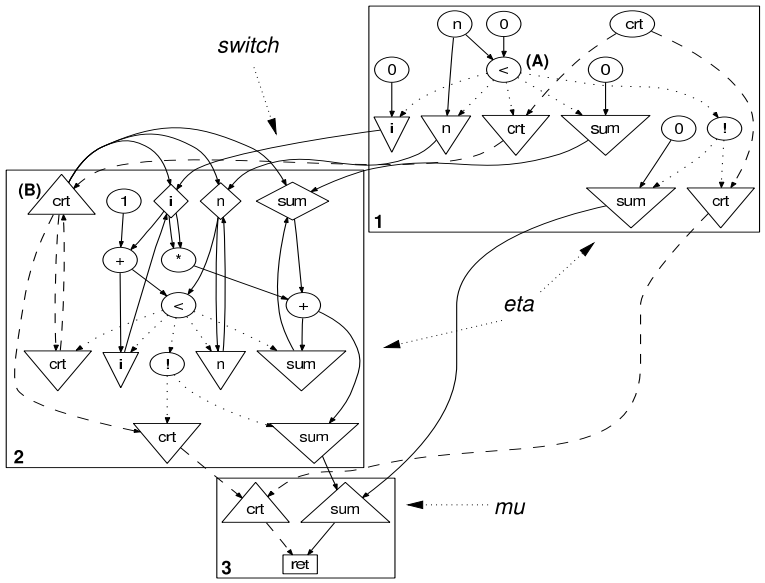

Figure 15: *A C program with a simple loop and its almost complete Pegasus representation. Each hyperblock is shown as a numbered rectangle. The dotted lines represent predicates. The dashed lines are used to represent control path. For simplicity we have ommitted from this figure the memory tokens (since the program does not access memory), the return address and the stack pointer values.*

CASH handles specially inner loops in many places, since the program will very likely spend a lot of time in such loops. But to a first degree, handling loops does not require any special treatment in Pegasus. An inner loop is just a CFG edge from the bottom of the hyperblock to the entry point. An outer loop is a CFG edge from the bottom of a different hyperblock to the entry point. To implement the these CFG edges, the ETA, MU and SWITCH operators are used as described above. Figure 14 shows the control-path synthesized for to the hyperblocks in Figure 4, including 2 inner loops and an outer loop.

Figure 15 shows a complete C function that uses `i` to count up to 10 and `sum` to accumulate the sum of the squares of `i`. On the right is the program's Pegasus representation, which consists of three hyperblocks. We label nodes with *crt* and use dashed lines for the control path. The code consists of three hyperblocks:

Hyperblock 1 initializes `sum` and `i` to 0, and seeds the control path from the input marked *crt*. It computes the predicate `n<0`. It contains procedure ARGument nodes `n`, `crt`.

- If `n<0`, the rightmost two ETAs in hyperblock 1 will steer data directly to hyperblock 3, skipping the loop body.
- Otherwise the leftmost 4 ETA nodes in hyperblock 1 steer execution to hyperblock 2, which contains the loop.

Hyperblock 2 represents the loop body. Note the control-flow input MU node *crt*, and the data path input SWITCH nodes, one for each of the live variables `sum`, `i` and `n`.

Data flows from top to bottom in hyperblock 2. The next hyperblock to execute is decided by the result of the comparison operation < between `i+1` and `n`.

- If `i<n`, the four ETA nodes in hyperblock 2 on the upper row fire and send their output back to the top of the loop (to the MU/SWITCH nodes).

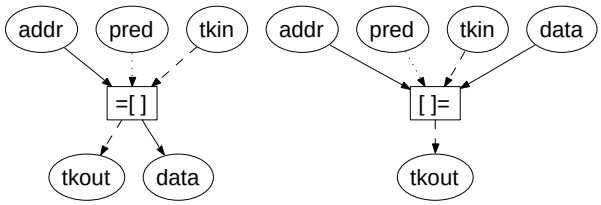

Figure 16: *Loads and store Pegasus operators. Besides address (and data for stores), each of these nodes has as additional inputs one predicate and one token. They both generate output tokens.*

- When `i` grows bigger than `n`, these four ETA operations no longer fire, but the two lower ETA operations open (since they are controlled by the complement of the output of <), letting the *crt* and `sum` values flow to hyperblock 3.

Hyperblock 3 is the function epilog, containing just the RETURN. Section 2.8 describes how the RETURN and ARGument nodes are implemented. Note that different variables are live on different exit paths from hyperblock 2.

2.7 Pointer Synthesis

The subset of Pegasus we have described so far can be used to implement arbitrary computations on scalar values. For handling pointers, more is needed. The synthesis of memory operations is done in two steps: first a simple, high-level structure is built, which is used to run many optimization algorithms. Next the high-level representation is lowered to a complete RTL-level representation, which generates a custom Memory Access Network (MAN). The MAN links the memory operations with the memory subsystem and transports memory coherence traffic. The coherence traffic is a set of control signals which ensures that memory content is always correct, even when multiple program operations are executing out-of-order, and even when memory is allowed to complete requests out-of-order.

Memory layout. The translation of C into hardware is eased by maintaining the same memory layout of all program data structures as implemented in a classical CPU-based system. The heap structures have the exact same layout. However, CASH can use arbitrarily many registers for variables, so it never has to spill registers on the stack.

CASH currently uses a single monolithing address space to store the heap, data, and stack. There is nothing intrinsic in the spatial computation model that mandates the use of a monolithic memory; on the contrary, using several independent memories (as suggested for example in [80, 2]) would be very beneficial. We have resorted to using a single memory because this is the memory model of a traditional processor; thus when using CASH in conjunction with hardware/software partitioning, as shown in Figure 2, it is easier to implement the coherent memory view between the CASH synthesized circuits and the processor. The use of multiple disjoint memories is a research topic we plan to address.

2.7.1 High-Level Memory Access Representation

In Pegasus memory accesses are represented using two operations: LOAD and STORE, as illustrated in Figure 16. These operations have a *predicate* input. Operations such as these, with side-effects, cannot be speculatively executed, and thus the path predicate has to be used to guard their execution. When the predicate input is "false", such an operation behaves essentialy like a NOP.

The compiler adds dependence edges, called *token edges*, to explicitly synchronize operations whose side-effects may not commute (i.e., which must be executed in a specific order to generate correct results). LOAD, STORE, CALL, and RETURN all have token inputs and outputs. A memory operation must collect tokens from all its potentially conflicting predecessors (e.g., a STORE following a set of LOADs). The COMBINE operator is used

```
void f(unsigned*p,
       unsigned a[],
       int i)
{
    if (p)
        a[i] += *p;
    else
        a[i] = 1;
    a[i] <<= a[i+1];
}
```

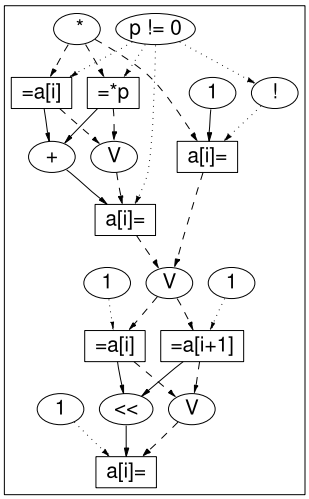

Figure 17: *Sample program and fragment of the (unoptimized) Pegasus representation highlighting the token network and predicates. The address computation structures and the* RETURN *node have been simplified. The "*" node is memory token input to the procedure. "V" denotes combine operators. A combine operator emits a token when it has received all its token inputs.*

for this purpose. COMBINE has multiple token inputs and produces a single token output; it generates an output only after it receives all its inputs.

Figure 17 shows the simplified Pegasus representation corresponding to a small program fragment. Looking, for example, at the node labeled with `=a[i+1]` representing a load from `a[i+1]`, we notice that it receives its token from a combine operator, whose inputs come in turn from two stores to `a[i]`. These tokens will ensure that the load will not occur until these stores have taken place, thereby preventing read-after-write (RAW) hazards.

Tokens encode dependences between pointers. CASH also uses tokens for a series of new memory access optimizations [8]. Furthermore, tokens are explicitly synthesized as memory coherence signals.

2.7.2 Memory Access Network Synthesis

In order to implement LOAD and STORE operations the compiler has to synthesize a global on-chip network. This network arbitrates access to the limited number of memory ports and also implements the token traffic, which is in essence a memory coherence traffic. The implementation of the MAN and MAN-specific optimizations are described in [36]. The essential architecture is presented in Figure 18. Load operations forward address requests through the access network. Each memory operation in the circuit has a different access port in the access network. The access network is a tree of arbiters. The requests eventually make it to the root of the MAN. This is a unique serialization point, which generates tokens through the coherence network. The serialization point establishes the memory ordering of the requests.

The memory station forwards the requests to a traditional cache hierarchy. When the requests return, possibly out-of-order, the replies are injected in the value network, which is just a demultiplexor tree, which routes requests back based on the original node identity which injected the request.

Simultaneously, the coherence network, another demultiplexor tree, forwards the coherence token to its output port. There a fanout node forwards the token to all dependent operations.

The MAN is represented in Pegasus as a collection of simple ports, arbiters (muxes), demuxes, simple routers, and the somewhat more complex access tree root node, which interfaces with the cache hierarchy.

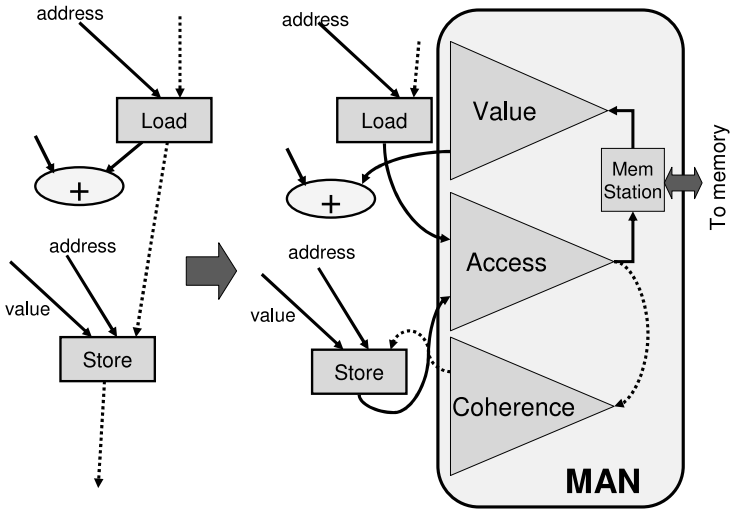

Figure 18: *The memory operations are synthesized into a Memory Access Network, which comprises three independent sub-networks: an access network, for forwarding requests, a value network, for receiving replies, and a coherence (token) network, which implements the high-level token network.*

2.8 Synthesizing Procedures

CASH does procedure synthesis also in two stages. First, CASH uses four operations to represent control transfer across procedures. Second, these four nodes are further decomposed into primitive control-flow constructs based exclusively on the already described nodes MU, SWITCH and ETA.

Procedure-level control-flow is initially represented through the following kinds of nodes: CALLs and CONTinuations, ARGuments and RETURNs. A CONT node is always paired with a CALL, and is used to receive the returned value.

Procedure arguments include:

- Scalar arguments, passed on wires, or on the stack, depending on the calling convention used.
- *crt* (the control flow signal).
- The memory token (*).
- The current stack pointer (a scalar value).
- A *return address*. The return address encodes the identity of the CONTinuation node, and is used by the RETURN to steer control back to the caller.

An example complete representation of a call-return node is shown in Figure 19.
These nodes are lowered to primitive nodes using the following transformations:

- The CALL node is synthesized as set of ETA nodes, one for each argument. The eta nodes outputs are connected to the corresponding ARG nodes of the callee.
- The ARG nodes become hyperblock entry points, and are synthesized simply as MU/SWITCH nodes. They have an input connected from each of the possible callers.
- The RETURN node is synthesized similar to a C `switch` statement (not a SWITCH node), based on all possible return addresses (continuations). For each possible continuation a new ETA node is built.

```
voif g()
{
}

void f(int p)
{
    if (p)
        g();
}
```

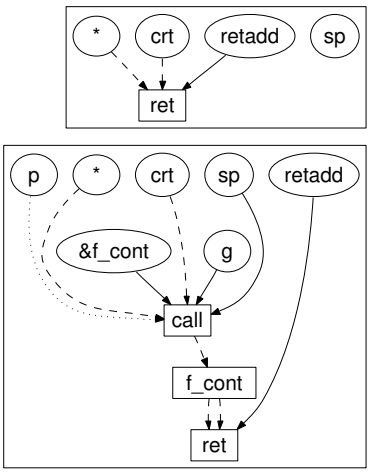

Figure 19: *Sample procedures and high-level Pegasus representation. The callee* `g` *does not need a stack frame, and thus its "sp" stack pointer* ARGument *has no output.*

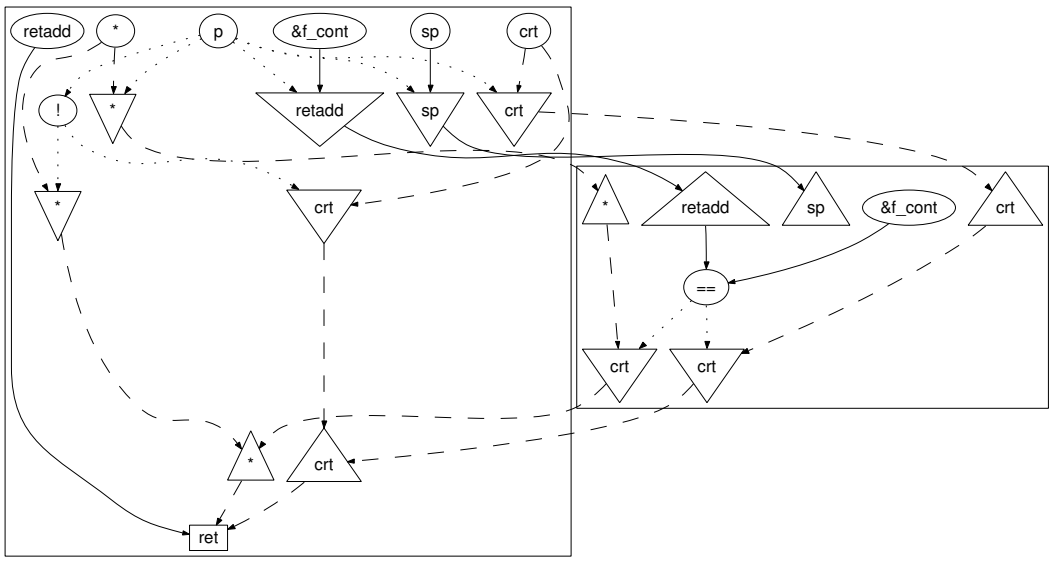

Figure 20: *Synthesis of a call from Figure 19 (the* ARG *and* RETURN *nodes from procedure* `f` *have not been synthesized to simplify the example). We have assumed there are no other callers of* `g` *in the program.*

- Finally, the CONT node is synthesized as MU/SWITCH nodes. Each receives inputs from all possible callees.

2.8.1 Recursion

Recursion is fully supported through the use of a memory stack, as illustrated by Figure 21. The stack is currently stored in the global memory. Before a recursive call, CASH generates code to store on the stack (using STORE operations) all the values live at the point of a recursive call (including the return address). The live values are restored (using LOAD operations), after the call returns. A recursive call is always placed at the beginning

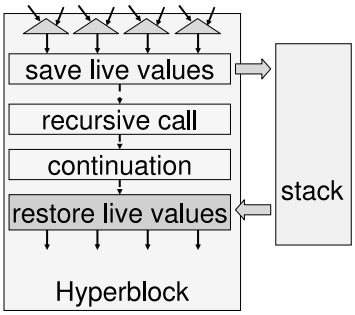

Figure 21: *Recursion is implemented by using a call stack and saving and restoring live values.*

of a hyperblock. Determining which calls are recursive is done based on the call-graph, computed by the Suif front-end.

2.8.2 Function Pointers

Currently function pointers are handled by using the call-graph to enumerate all possible callees of each pointer. Then the call through a pointer is compiled as a large C `switch`-like statement which invokes each possible callee, reducing the CALL to the constant function case.

This solution is not scalable to large programs, so we are considering as future work an implementation where the CALL and RETURN statements are routed on a dynamic on-chip network.

2.9 The Dataflow Semantics of Pegasus

All Pegasus constructs have a precise and concise operational semantics. A formal definition of this semantics if available in an appendix of [7]. Informally, at run-time each edge of the graph either holds a value or is $\perp$ ("empty"). An operation begins computing once all of its required inputs are available. The operation can latch a newly computed value only when its output is $\perp$. The computation consumes the input values (setting the input edges to $\perp$) and produces a new output value.

The precise semantics is useful for reasoning about the correctness of compiler optimizations and is a precise specification for compiler back-ends. Currently CASH has three back-ends: (1) a graph-drawing back-end, which generates drawings in the `dot` language [33]; (2) a simulation back-end, which generates a C functional simulator (effectively, an interpreter of the graph structure); and (3) the asynchronous circuits back-end (CAB), described in detail in the rest of this document.

2.10 CASH Optimizations

CASH performs a wide range of program optimizations. The Suif-based front-end performs: procedure inlining, loop unrolling, call-graph computation, and basic control-flow optimizations, intraprocedural pointer analysis, and live-variable analysis.

After translation to Pegasus, all major scalar optimizations based on SSA are implemented. The scalar optimizations include: global constant propagation, copy propagation, constant folding, dead code elimination, dead variable removal, strength reduction (for constant multiplies, divides, modulo), loop-invariant code motion, algebraic optimizations, re-association, multiplexor optimizations, induction variable strength-reduction,

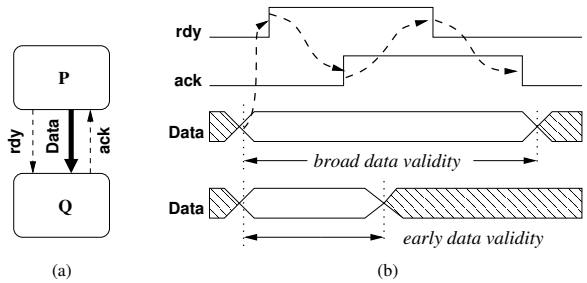

Figure 22: *Synchronization: 4-phase bundled-data handshaking*

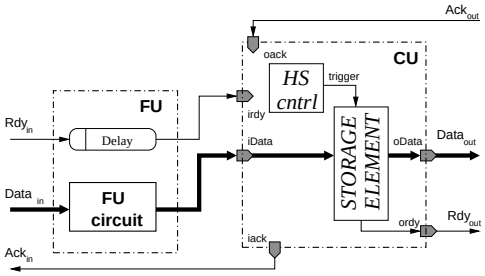

Figure 23: *A typical P-stage*

common subexpression elimination, partial redundancy elimination, unreachable code removal, straightening, branch chaining, removal of empty conditionals, loop decoupling [8], bitwidth reduction [10], pipeline balancing (also known as "slack matching") [58].

Memory-based optimizations are supported by an intra-procedural pointer analysis, and include: memory disambiguation, scalar expansion, register promotion, partial redundancy elimination for memory-resident values [8], pipelining memory accesses.

Boolean optimization is used to simplify predicate computations; it consists in a heuristic *don't care* discovery, and also employs the `espresso` [4] optimizer.

3 CAB: The Cash Asynchronous Back-End

The CASH Asynchronous Back-end (CAB) translates Pegasus into actual circuits: each Pegasus node is synthesized into a separate pipeline stage, called P-stage, and each edge is translated into a point-to-point communication channel. In this section we describe this translation process. We start by describing the actual circuit generated by CAB.

3.1 Pipe Stage Architecture

This section describes the P-stages, the fundamental circuit building block.

3.1.1 Handshake

Dataflow synchronization is achieved through 4-phase bundled data handshake [79] between P-stages. Figure 22a demonstrates this handshake between two stages, P and Q. The handshake is implemented using three signals: a data bundle ($Data$) and two control signals (rdy and ack). A complete 4-phase handshaking proceeds as follows: when P has generated new data on the $Data$ bus, it raises rdy to indicate data validity. Once Q has used this new data, it acknowledges by raising the ack signal. Finally, both rdy and ack are lowered before a new iteration can

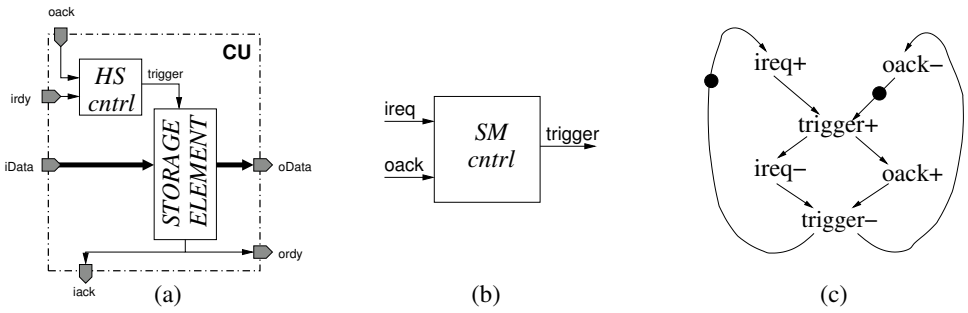

Figure 24: *Sutherland Micropipelines (a) CU implementation, (b) Block-level view of the handshaking controller, (c) STG specification for the handshaking controller*

begin. The waveforms for one complete handshake are shown in Figure 22b. Notice that the waveforms show two different waveforms for the $Data$ signals: one for *broad data validity*, where the data produced by P stays valid from before $rdy \uparrow$ until after $ack \downarrow$, and one for *early data validity*, where data is valid from before $rdy \uparrow$ until after $ack \uparrow$ (i.e. only half of the handshake, hence the name early). Both these protocols are used by CAB circuits.

This protocol assumes an implicit timing dependency between $Data$ and its corresponding rdy signal: it requires that $Data$ indicate a stable value from before rdy is raised. Hence, the producer must first put valid data on $Data$ before raising rdy. This dependency is called a *bundling constraint*.

A typical P-stage is shown in Figure 23. It consists of two main blocks:

FU is the functional unit, which performs the pipeline stage computation. The block takes as input the P-stage's data input(s) and its rdy signal, and outputs the result of the stage and a done signal which indicates when the result is stable. The FU consists of a combinational block, which implements the actual computation (e.g. an adder) and a *matched delay*, which matches the propagation delay through the FU's combinational block.

CU is the control block that performs the 4-phase bundled-data handshaking. It has three input ports: (1) input ready signal, $irdy$; (2) output acknowledge signal, $oack$; and (3) data generated by the functional unit, $iData$; and, it generates three outputs: (1) output ready, $ordy$, the request signal to the following stage; (2) input acknowledge, $iack$, the acknowledge to the previous stage; and (3) output data to consumer, $oData$.

3.1.2 Control Unit

The CU in each pipeline stage performs two functions: it implements the 4-phase handshaking between adjacent stages, and it stores data items produced by the current stage for the following stage to use. The implementation of the storage element is described in Section 3.1.3.

The implementation of the handshaking controller in each stage has implications on both the latency and the throughput of the system. Currently, CAB uses two types of handshake controllers: Sutherland Micropipelines High-Capacity pipelines. The implementation of the control unit for the two styles is shown in Figure 24 and Figure 25.

Sutherland Micropipeline (SM). In [87], Sutherland has proposed an asynchronous pipeline style implementation where adjacent stages communicate using 2-phase bundled-data handshaking. We have extended this style to use 4-phase handshaking. Figure 24(a) shows the implementation of the control unit. The handshaking controller (*SM cntrl*) accepts as inputs $ireq$ from the functional unit and $oack$ from the following stage (Figure 24b), and produces a single output $trigger$, which activates the storage element. In turn, the storage element signals when it is done storing data; this event is used both to activate the following stage ($ordy+$), and to acknowledge the previous stage ($iack+$). Figure 24c shows the STG specification of the handshaking controller. The controller has a very simple implementation: a single C-element with the $oack$ input inverted.

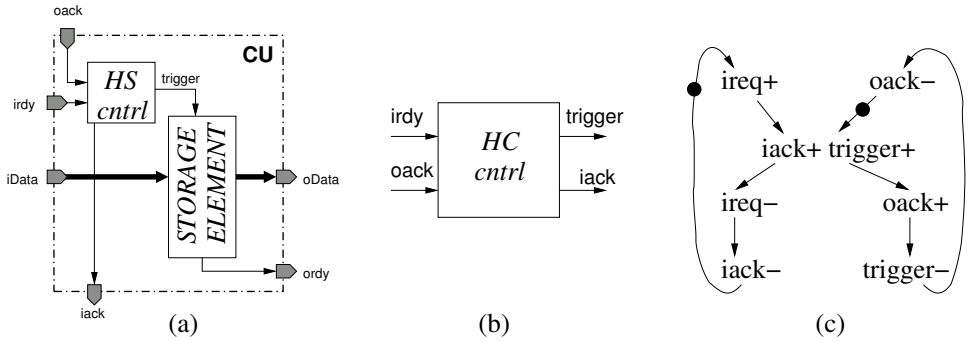

Figure 25: *High-capacity micropipelines: (a) CU implementation, (b) block-level view of the handshaking controller, (c) STG specification for the handshaking controller*

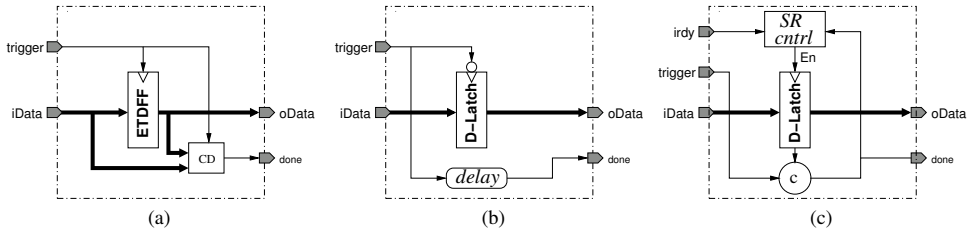

Figure 26: *Storage element implementations*

The bottleneck to achieving high throughput is the return-to-zero (RTZ) phase of the handshake. The acknowledgement to the producer cannot be lowered (i.e. $iack-$ event) until the consumer acknowledges this stage's output. Stage N cannot start processing the next iteration until stage $N+2$ has acknowledged the current iteration. The occupancy of such a pipeline style is at most 50% (i.e., one of every two consecutive pipeline stages can hold a useful data item).

High Capacity (HC). An improved control unit design is shown in Figure 25a, which allows for 100% occupancy. Compared with Figure 24a, there are two differences. First, the handshaking controller has been changed. Second, the acknowledgement to the previous stage is now produced directly by *HC cntrl*, instead of by the storage element. These changes are reflected in the block diagram of the handshaking controller (Figure 25b), which now has an extra output, the $iack$ acknowledge signal to the previous stage.

The STG specification of this controller is shown in Figure 25c. The handshaking starts when the stage has finished computing ($ireq+$) and the next stage is empty ($oack = 0$). Upon synchronizing these two conditions, *HC cntrl* concurrently enables the register to store data and acknowledge the previous stage. After that, the controller finishes the handshake with the previous stage ($ireq- \rightarrow iack-$) concurrent with the handshake with the next stage ($oack+ \rightarrow trigger+ \rightarrow oack-$). Once both handshakes are finished, the controller can resume its operation. Notice that the return-to-zero phases of the two handshakes are no longer synchronized, which enables this controller to obtain 100% pipeline occupancy.

We have used Petrify [20] to synthesize the STG specification into actual circuits.

3.1.3 Storage Element

Each P-stage uses a data storage element to hold the output data until its receipt has been acknowledged by consumers down-stream. CAB can use one of three types of storage elements that can be combined with any of the two pipeline styles described above. Figure 26 shows the the three register styles.

Register with Completion Detection (CD). The register shown in Figure 26a is the most robust implementation of the storage element. It consists of an edge-triggered D flip-flop (ETDFF), and a completion detection (CD)

element. The $trigger$ signal from the handshaking controller is used both as a "clock" signal for the ETDFF's, and as an input to CD. The storage element first receives the data input from the functional unit. When $trigger$ goes high, data is stored in the ETDFF. CD simply compares the data before and after the ETDFF, and the result of the comparison is synchronized with the trigger through an assymetric C-element, which outputs $Done$ to indicate that the new data item is safely stored. The implementation is very robust with respect to the latency of the ETDFF's.

Normally Transparent Latches. The drawback CD-based storage element is that the CD circuit is on the critical path of each stage. The implementation of Figure 26b eliminates the CD overhead by using normally transparent latches to store data, eliminating the CD circuit. In this design, the D-latches are normally open (i.e. they transfer data from input to output) when the trigger signal is low (i.e. when the stage has not yet finished computation). When the trigger signal becomes high, the latches become opaque, thus keeping the data item unchanged for the next stage computation. Once the next stage has finished computation, the latches become transparent again. This implementation changes the data validity on the channels from broad to early.

Self-Resetting (SR) Latches. While the normally-transparent latch improves the latency of a stage, it also increases the power consumption, since each data glitch during the computation of a stage is passed unfiltered to the next stage, triggering useless computation. The implementation in (Figure 26c) eliminates spurious transitions in the functional units, while retaining the latency benefits of the transparent latch.

The SR latch implementation introduces a new controller. (*SR cntrl*). The latches are now normally opaque (no inversion on the "clock" signal). The controller has two inputs ($ireq$: the done signal from the functional unit, and $Done$: the done signal of the storage element), and a single output En. During a cycle, the controller opens the latches as soon as the functional unit has finished computation ($ireq+$), and closes them as soon as the new data item is safely stored ($Done+$). After these events, the controller waits for both signals to reset to zero before starting again. By allowing the latches to be open only after a new data item is computed, all the glitches produced by the synchronous functional unit are filtered out, and the power consumption decreases.

The triggering signal no longer controls the SR latches; instead, it is simply delayed in order to produce the $Done$ signal. This change, while simplifying logic, also introduces a timing constraing, described in Section 3.1.4.

3.1.4 Timing Constraints

The circuits synthesized by CAB are not QDI, as they have several timing constraints. These constraints fall into four categories:

Matched Delay. Each P-stage uses a delay element to match the data propagation delay (see Figure 23) of the FU. This delay is determined by post-synthesis timing estimates of the FU. Actual post-layout delay may violate this constraint. However, we have engineered the circuits to reduce the probability of such a violation by ensuring that the FU is a localized circuit, whose output has a fanout of exactly one. This reduces the discrepancies in post-synthesis and post-layout timing estimates due to poorly estimated capacitive loads. Hence, we can obtain an early estimation of the matched delays which still respects this timing constraint.

Bundling Constraint. The bundled data protocol works under the assumption that data *always* arrives at the destination before the corresponding Rdy signal. The completion detection circuit (see Figure 26a) ensures that this constraint is obeyed at the output of a pipeline stage. However, layout could violate these constraints if the wires connecting pipeline stages are arbitrarily routed. The problem can be addressed by supplying hints to the layout tool — for example, by ensuring that the weight on the data wire is bigger than that on the Rdy wire (assuming that the tool optimizes for minimum wire-length).

Constraints in High-Capacity Pipelines. The *HC cntrl* controller in each such stage produces an acknowledge to the previous stage in parallel with enabling the register to store data. Thus, there are two ways the timing assumptions may be violated for a pipeline stage.

- First, under QDI assumptions and with broad data validity, it is possible that the handshake with the previous stage finishes, and a new data item is produced by the current stage even before the storage element

finishes storing the current data item. Thus the timing constraint for correct behavior is: $\delta(iack+ \rightarrow irdy- \rightarrow iack- \rightarrow irdy+) > \delta(REG)$.

- Second, under QDI assumptions and with early data validity (the Normally-Transparent Latches register implementation), it is possible that the acknowledge to the previous stage opens the respective latches, and the current stage produces a new data item even before the storage element in the current stage has time to close the latches. The timing constraint for correct behavior is: $\delta(iack+ \rightarrow irdy-) > \delta(REG)$. In practice, these timing constraints are extremely easy to meet, since the delay of a functional unit is usually many times longer than that of a register.

Constraints in Self-Resetting Latches. As noticed above, the implementation of storage elements with self-reseting latches introduces a race between the enabling of the latches to pass data and the $trigger$ signal, which indicates when the data is safely latched. In practice, the timing constraint is: $\delta(irdy+ \rightarrow En+) < \delta(irdy+ \rightarrow trigger+$, which simply states that the delay through the handshaking controller must be longer than the delay through the *SR cntrl* contrller. In our current implementations, this condition holds.

In all of the circuits that we have synthesized (described next), we have never encountered any post-layout timing violations.

3.2 P-stage Families

We classify a collection of pipeline elements as a *complete family* if it can be used to construct non-linear pipelines, where the stages can be a mixture of strict and lenient (i.e. some can can compute before all its inputs have arrived). Such a family must contain at least the following four classes of elements:

1. ALU: for standard ALU computation.
2. Fork and Join: building blocks for non-linear pipelines (allowing operators with multiple inputs and with fan-out).
3. Input Selection: for speculation computation or resource sharing (e.g., MUX, MU, SWITCH).
4. Conditional Output: for building control-flow (e.g., ETA).

The P-stages used in our circuits form a complete family. The ALU stage architecture was essentially described in Section 3.1. In this section, we describe P-stages from the remaining classes.

3.2.1 Fork and Join

Extending the basic P-stage, shown in Figure 23 to handle multiple inputs and outputs is straight-forward, and is well-known from the asynchronous circuits literature.

Join patterns allow the implementation of operators with multiple inputs, e.g., add. To build an operator with multiple inputs a tree of C-elements collects all the Rdy_{in}'s, and the output of this tree feeds the input to the $Delay$.

A fork pattern allows an operator to fanout (copy) its data to multiple consumers. To implement fanout the output data$Data_{out}$ and down-stream ready signal Rdy_{out} are broadcast to all consumers. The acknowledgement signals from all the consumers are collected using a tree of C-elements, and the output of the root of this tree feeds the $oack$ port of CU.

3.2.2 Input Selection

There are many types of pipelines stages whose task is to select one of many inputs and pass its value to the output. They differ subtly in semantics. Each of them is useful in a different context.

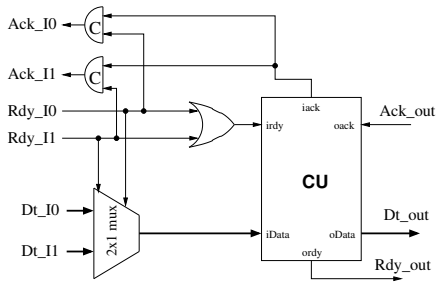

Figure 27: *A* 2×1 CALL *P-stage, used to implement a Pegasus* MU *operator.*

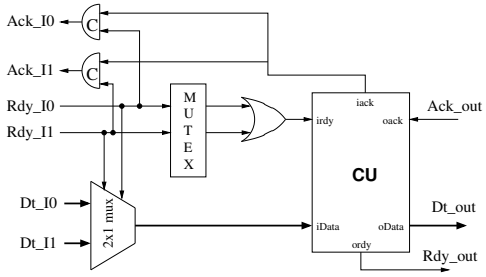

Figure 28: *A* 2×1 ARBITER *P-stage, used to implement the Pegasus memory tree arbiter nodes.*

Resource Sharing. The first two elements, MU[1] and ARBITER, deal with sequencing accesses to a shared resource. They have N passive inputs ports, and any input that arrives must be transmitted to the output; for MU, the inputs are assumed to be mutually-exclusive, while for ARBITER, the inputs might be concurrent and the stage has to arbitrate between them. Implementations of 2×1 MU and ARBITER are shown in Figures 27 and 28 respectively. They are identical except that the latter uses mutual exclusion element to arbitrate between concurrent inputs.

Data Multiplexing. The MUX and SWITCH, differ from resource sharing elements in that all their inputs are expected to arrive and participate in computation round. They both have N data inputs and additionally there is an auxillary selection input that specifies which input must be copied to the output. Hence, there may be an additional N selection inputs, if one-hot encoding is used. The difference between MUX and SWITH is that MUX acknowledges all of its inputs and hence discards the unused inputs, while SWITCH acknowledges only the copied data input and the selection input.

Figure 29 shows the implementation of the SWITCH element. It waits for the selection input, and based on its value, transmits the corresponding data input (once it has arrived) to the output. Note that some of the other data inputs, which are not selected, may have already arrived; however, they are not acknowledged until they are selected by the selection input, and therefore are never discarded. Thus, for a SWITCH, one of the input channels may have a pending input for multiple consecutive computation cycles. This makes the SWITCH an essential ingredient of the implementation of looping constructs.

The selection input is sampled by an edge triggered flip-flop in order to avoid any hazards for the duration of the entire handshake. The selection is synchronized with the corresponding Rdy signals on the data channels through an assymetric C-element. Otherwise the operation of the SWITCH is identical to that of the resource sharing elements.

MUX is described in Section 3.2.4, in the context of lenient operations.

[1]The MU node is traditionally known in the asynchronous circuits literature as a "call" node. The "mu" name comes from the dataflow machines literature.

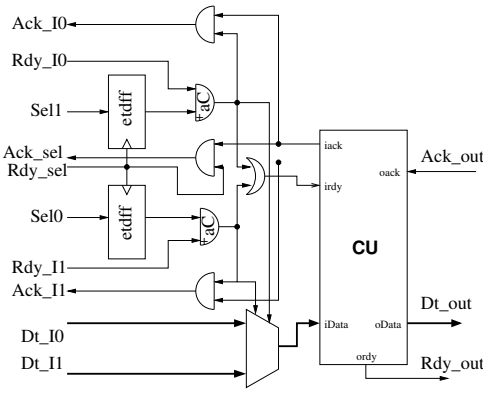

Figure 29: *A* 2×1 SWITCH *P-stage*

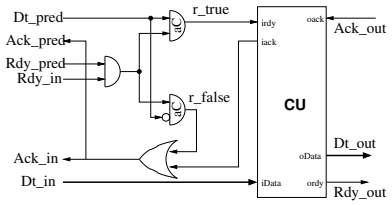

Figure 30: *Conditional-Output Pipeline Stage*

3.2.3 Conditional-Output Element

The Conditional-Output (ETA) element has a passive data input port, a passive predicate input port, and a single active output port. If the predicate input is true, then the data input is sent down-stream through the output port. If the predicate input evaluates to false, then no output is generated, and the inputs are acknowledged. Recall that ETA is used to translate branches (control-flow).

Figure 30 shows the implementation of the Conditional-Output pipeline stage. The stage first synchronizes the requests on the two ports. Using two asymmetric C-elements, the value of the predicate is checked: if true, the *r_true* signal starts the handshake in the CU controller, which eventually will acknowledge the input ports through the OR gate. At the output port CU simply latches the data and performs a full handshake.

3.2.4 Lenient (Early) Evaluation

By definition, a **strict** operation generates its output only after all inputs arrive. In contrast, a **lenient** (or early termination) operation can generate an early output using only a subset of the inputs. As an example, a MUX can generate its output as soon as the selection information and the selected input have arrived. The output can be generated immediately since we know that all the other inputs will be discarded upon arrival. Lenient operations are essential to reduce the *dynamic* critical path.

In general, there is a finite set of conditions under which an operation can be triggered early. If we define I as the set of all inputs and their data ready signals, then we can define the set of triggering conditions as:

$$C = C_{all} \bigcup_{I_k \subset I} \{C_i | C_i = F(I_k)\}$$

where $F(I_k)$ is a Boolean function which computes a lenient triggering condition, and C_{all} is the strict case when all inputs have arrived. The set on the right side of the union describes the lenient triggering conditions. If any one of the conditions in this set is true, then we say that a sufficient input condition has been met to begin

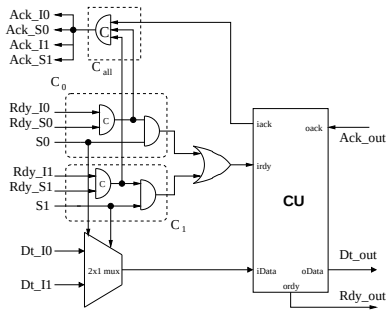

Figure 31: *A lenient 2 × 1 mux: There are two sufficient conditions, C_0 and C_1, for the early triggering of the operation, and one necessary condition, C_{all}, for acknowledging all inputs*

computation. In a strict operation, the set $C = C_{all}$. While lenient operations may generate their output early, our current protocol requires that a lenient operation's inputs are acknowledged only after all inputs have arrived.

In the asynchronous circuits literature, leniency was proposed under the name "early evaluation" [72] and "or causality" [103]. A recently proposed enhancement [5] can send early acks, potentially speeding up computation at the cost of more machinery to quash partially executed operations.

The controller for a lenient pipeline stage essentially embodies the lenient triggering conditions, as shown in Figure 31. The figure shows a MUX with two data inputs, $D0$ and $D1$, and two selection inputs $S0$ and $S1$. Each of these inputs is accompanied by their respective Rdy signals. For each pair of data and selection inputs, the stage first synchronizes their request signals through a C-element, and then checks whether the predicate is true (using an AND gate). Thus[2]:

$$C_0 = (Rdy_{s0} \odot Rdy_{D0}) \wedge S0$$
$$C_1 = (Rdy_{s1} \odot Rdy_{D1}) \wedge S1$$
$$C = \{C_{all}, C_0, C_1\}$$

If either condition C_0 or condition C_1 holds, the controller CU starts handshaking with the down-stream consumer. Notice that C_{all} produces the acknowledge to the previous stages only when all the inputs have arrived *and* the CU has started the handshake with the consumer.

A strict implementation of the MUX is simpler. All the data and selection Rdy signals feed into a single C-element, whose output represents the condition C_{all}, and wired to the $ireq$ input port of CU. The $iack$ signal is the input acknowledgment, which is directly broadcast to all producers up-stream.

3.3 From Verilog to Layout

Figure 32 shows the tool flow from CAB down to laid out circuits.

CAB : The back-end of CASH translates Pegasus into pipelined, asynchronous circuits, described in structural Verilog. The final circuits are an interconnection of pipeline stages, without global controllers. Each P-stage is the direct correspondent of a Pegasus node. The data transfer between two P-stages is mediated by a local handshake.

Synthesis : The Verilog circuit generated by CAB is a mixed tech-mapped/behavioral description. The control-path is fully synthesized, while the datapath logic is behavioral. Synopsys Design Compiler (version 2002.05-SP2) is employed to synthesize the datapath logic. The result is a fully tech-mapped, gate-level Verilog description of the circuit.

[2]In the equation, we use $\odot$ to denote a C-element operation.

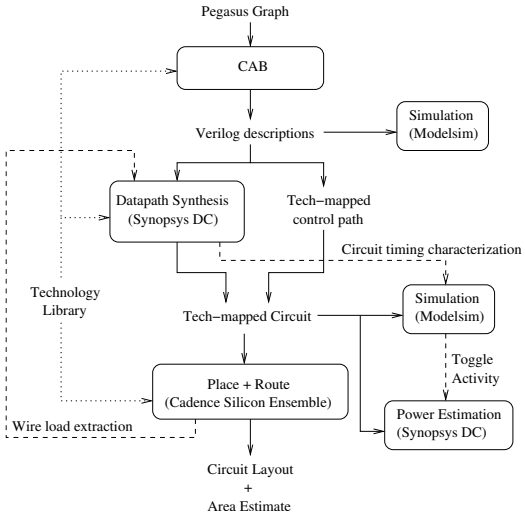

Figure 32: *The back-end tool flow*

Layout : The gate-level circuit is placed and routed using Cadence Silicon Ensemble (version 5.4). This produces a layout specification.

ynthesis and re-layout: The information from the actual layout is used to back-annotate the circuit, and the synthesis phase is repeated to generate a new circuit with more accurate layout-dependent timing characterizations. The circuit is placed and routed again.

The circuit is simulated after the completion of each of the above phases using a Verilog simulator (Modelsim SE version 5.8b, in our current tool flow). After processing by CAB, the circuit is not fully tech-mapped; the mixed structural/behavioral simulation is useful for debugging. After synthesis, the critical path delay can be accurately estimated and used to obtain precise performance evaluations. After layout, the simulation uses the actual wire load information. Simulation produces a toggle activity file that is used to estimate dynamic power consumption by the Synopsys Design Compiler.

We have adapted commercial CAD tools customarily used for creating synchronous circuits to our asynchronous design flow. This requires some awareness and caution. For example, we cannot allow Synopsys to synthesize (let alone infer) registers, and we cannot synthesize any combinational block with feedback loops (basic asynchronous blocks like C-elements contain feedback loops). We get around this problem by ensuring that only the datapath is synthesized by Synopsys, while CAB synthesizes the control path and registers.

We use a standard cell library designed for use in synchronous circuits. In consequence, some commonly used gates in asynchronous circuits, such as C-elements, are not available in the library. Although we can create an implementation using existing library cells, our experiments show that such C-element implementations are 2-3 times slower and 2 times larger than a custom transistor-level design.

Our investigation indicates that the place/route phase is considerably simplified compared to a synchronous circuit design flow. No clock-tree is required, routing does not need to be timing driven, and there are no clock-related timing violations. Timing closure is thus not necessary.

Overall, the experience with they layout of our asynchronous circuits has been very positive: not only have we had no timing violations, but often the post-layout circuit has better latency and power consumption than pre-layout estimates. It seems that the local communication and self-timed nature of our circuits simplify layout considerably.

References

[1] Jonathan Babb, Martin Rinard, Csaba Andras Moritz, Walter Lee, Matthew Frank Rajeev Barua, and Saman Amarasinghe. Parallelizing applications into silicon. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 1999.

[2] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and C. McMullin. *Logic Minimization Algorithms for Digital Circuits*. Kluwer Academic Publishers, Boston, MA, 1984.

[3] C.F. Brej and J.D. Garside. Early output logic using anti-tokens. In *International Workshop on Logic Synthesis*, pages 302–309, May 2003.

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

[5] Mihai Budiu and Seth Copen Goldstein. Optimizing memory accesses for spatial computation. In *International ACM/IEEE Symposium on Code Generation and Optimization (CGO)*, pages 216–227, San Francisco, CA, March 23-26 2003.

[6] Mihai Budiu, Mahim Mishra, Ashwin Bharambe, and Seth Copen Goldstein. Peer-to-peer hardware-software interfaces for reconfigurable fabrics. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 57–66, Napa Valley, CA, April 2002.

[7] Mihai Budiu, Majd Sakr, Kip Walker, and Seth Copen Goldstein. BitValue inference: Detecting and exploiting narrow bitwidth computations. In *European Conference on Parallel Processing (EUROPAR)*, volume 1900 of *Lecture Notes in Computer Science*, pages 969–979, Münich, Germany, 2000. Springer Verlag. An expanded version is in TR 00.

[8] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Jeanne Ferrante. Predicated static single assignment. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1999.

[9] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. *IEICE Transactions on Information and Systems*, E80-D:315–325, 1997.

[10] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[11] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *25 Years of the International Symposia on Computer Architecture (selected papers)*, pages 125–131. ACM Press, 1998.

[12] Emden Gansner and Stephen North. An open graph visualization system and its applications to software engineering. *Software Practice And Experience*, 1(5), 1999. http://www.research.att.com/sw/tools/graphviz.

[13] Girish Venkataramani, Tobias Bjerregaard, Tiberiu Chelcea, and Seth C. Goldstein. Hardware compilation of application-specific memory access interconnect. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 25(5):756–771, 2006.

[14] Ruibing Lu and Cheng-Kok Koh. Performance optimization of latency insensitive systems through buffer queue sizing. In *IEEE/ACM International Conference on Computer-aided design (ICCAD)*, 2003. 3D.3.

[15] Scott A. Mahlke, Richard E. Hauk, James E. McCormick, David I. August, and Wen mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *International Symposium on Computer Architecture (ISCA)*, pages 138–149, Santa Margherita Ligure, Italy, May 1995. ACM.

[16] Scott A. Mahlke, David C. Lin, William Y. Chen, Richard E. Hank, and Roger A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *International Symposium on Computer Architecture (ISCA)*, pages 45–54, Dec 1992.

[17] B. J. Nelson. Remote procedure call. Technical Report CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, California, 1981.

[18] Robert B. Reese, Mitch A. Thornton, and Cherrice Traver. Arithmetic logic circuits using self-timed bit level dataflow and early evaluation. In *International Conference on Computer Design (ICCD)*, page 18, Austin, TX, September 23-26 2001.

[19] C.L. Seitz. *System Timing*. Introduction to VLSI Systems. Addison-Wesley, 1980.

[20] Luc Séméria, Koichi Sato, and Giovanni De Micheli. Synthesis of hardware models in C with pointers and complex data structures. *IEEE Transactions on VLSI*, 2001.

[21] Ivan Sutherland. Micropipelines: Turing Award Lecture. *Comm. of the ACM*, 32 (6):720–738, June 1989.

[22] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

[23] A. Yakovlev, M. Kishinevsky, A. Kondratyev, L. Lavagno, and M. Pietkiewicz-Koutny. *Formal Methods in Systems Design*, volume 9, chapter On the Models for Asynchronous Circuit Behaviour with OR Causality, pages 189–234. Kluwer, 1996.