Pegasus: An Efficient Intermediate Representation

Mihai Budiu Seth Copen Goldstein

April 2002

CMU-CS-02-107

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We present Pegasus, a compact and expressive intermediate representation for imperative languages. The representation is suitable for target architectures supporting predicated execution and aggressive speculation. In Pegasus information about the global dataflow of the program is encoded in local structures, enabling compact and efficient algorithms for program optimizations. As a proof of the versatility of Pegasus, we have used it in a compiler translating C programs to hardware implementations.

This work was supported in part by NSF CAREER award CCR-9876248 and DARPA grant N00014-01-1-0659.

Keywords: optimizing compilers, intermediate representation, dataflow analysis, predication, static-single assignment, program optimization

1 Introduction

The choice of intermediate representation (IR) has an enormous impact on all aspects of compiler design and implementation. A good IR can create opportunities for new optimizations and analysis algorithms by summarizing important program properties. For example, the static single assignment representation (SSA) [1, 21] and a fast algorithm for its computation [8] enabled a wealth of efficient and powerful program optimizations.

The importance of a good IR is magnified when the target architecture of the compilation process has significantly different semantic properties from the source language. For example, the generation of code for parallel computers starting from a sequential language is a very challenging task. Another example is compilation for predicated architectures, which is hard to support using classical IRs.

In this paper we present Pegasus [Predicated Explicit GAted Simple Uniform Ssa], a new IR which makes explicit — in a single representation — the control flow, the data flow, and the synchronization of operations which interfere through side-effects. More importantly, Pegasus has a clean semantics, independent of the target architecture. This enables its use, for example, to bridge the gap between C programs and hardware implementations, enabling the convertion from an imperative, single-threaded model of computation to a highly-parallel, asynchronous, explicitly synchronized target.

Pegasus combines predicated SSA [5] and gated SSA [17], making explicit the switching of data values, enabling the compiler to use predication and aggressive speculation for exposing instruction-level parallelism (ILP). In the next section we describe the basic components of Pegasus and how the IR can be constructed starting from an imperative language. The operational semantics of Pegasus is described, informally in Section 3, and formally in Appendix A.

Section 4 shows that the compactness of Pegasus enables the use of extremely simple and efficient algorithms for many major compiler optimizations; in particular, we exhibit linear-time algorithms for almost all of the scalar optimizations from Muchnick [16]. The versatility of Pegasus is demonstrated in Section 5, where we describe its use in a compiler which translates ANSI C programs into hardware.

1.1 Contributions

In this paper we make the following contributions:

(1) We unify in one sparse intermediate representation Predicated SSA and the explicit representation of gating functions from GSA [17, 24]. The resulting representation summarizes many important program properties without the need for any external annotations.

(2) We show that Pegasus is flexible enough to express concepts such as control-flow, dataflow, def-use chains, predication and speculative execution using a single simple data structure.

(3) We show that Pegasus permits concise and efficient implementations of most major program optimizations. Moreover, because Pegasus summarizes important program dataflow information, program transformations implicitly recompute what in other representations would be changes in the dataflow information.

(4) We give a formal semantics for Pegasus.

(5) We show how the versatility of Pegasus can be used to synthesize application-specific hardware directly from ANSI C programs.

2 Building Pegasus

In this section we present an algorithm which constructs a Pegasus graph starting from a conventional control-flow graph (CFG) of each procedure; we introduce the basic components of Pegasus one by one, as they are built by this algorithm. The semantics of the basic components is presented informally in Section 3 and formally in Appendix A.

The construction of the Pegasus IR from a program is achieved in several stages: first the CFG is decomposed into large fragments (Section 2.2), some of which will be the innermost loops. Each fragment is compiled into speculatively executed code, in order to extract the maximum instruction-level parallelism (ILP). For this purpose the control-flow inside each fragment is transformed into data-flow (Section 2.3), and branches are transformed into multiplexors (Section 2.4), resulting in an SSA representation. In order to maintain the correct program execution, predicated execution is used for the operations which cause side-effects or exceptions, (Sections 2.5 and 2.6). Finally, the fragments are linked together into a complete Pegasus graph by using special control-flow operators (Section 2.7). The resulting representation is used by both the optimizer and the back-end.

The various algorithms and data structures we describe have the following asymptotic complexities:

- The complexity of the entire construction algorithm is given by the complexity of the component phases, the most complicated of which is a dataflow analysis used to determine points-to sets. Given the results of this analysis, constructing the IR takes linear time in the size of the resulting structure.

- The Pegasus graph is a sparse representation: its space complexity is similar to a classical SSA representation.

- All optimization algorithms we present in Section 4 have linear time and space complexity in the size of the Pegasus representation.

2.1 The Pegasus Graph

Pegasus' representation is a directed graph whose nodes are operations and whose edges indicate value flow. A node may fanout the data value it produces to multiple nodes; the fanout is represented as multiple graph edges, one for each destination. An edge runs between the producer of the value and the consumer of the value. We call the incoming edges of a node "inputs" and the outgoing edges "outputs."

The data is produced by the source of an edge, transported by the edge and consumed by the destination. Once the data is consumed, it is no longer available. In general, nodes need all their inputs to compute a result, though there are some exceptions.

Each node performs a computation on its inputs to produce its outputs. We assume the existence of primitive nodes for all the major arithmetic and logic operations. We also have nodes which represent constant values and some special nodes for complex operations, such as memory access or procedure calls. Parameter nodes denote procedure arguments.

All the pictures of Pegasus graphs in this paper were automatically generated by our prototype compiler, using the *dot* graph drawing tool [10] as a back-end (some pictures were hand-edited to remove redundant details).

2.2 Decomposing the CFG into Hyperblocks

Building the Pegasus representation of a procedure starts by partitioning the CFG into a collection of *hyperblocks*. Hyperblocks were introduced in compilers that targeted predicated execution [15]. A hyperblock is a contiguous portion of the CFG, having a single entry point and possibly multiple exits. Hyperblocks are thus directed, acyclic graphs. In Pegasus a hyperblock is the unit of (speculative) execution: once execution enters a hyperblock, all component instructions are executed to completion. We will come back to this aspect in Section 2.5.

As pointed out in the literature about predicated execution, there are many choices for hyperblock boundary selection. At one extreme, each basic block is a hyperblock on its own, and thus there will be no speculative execution. At the other extreme, the CFG is covered with a minimal number of hyperblocks, i.e., each hyperblock is as large as possible. This is done by using structural analysis and computing minimal CFG intervals [16]. Each resulting interval is a hyperblock. No code duplication or loop peeling is performed. After this algorithm is executed, each reducible innermost loop body in the CFG is a hyperblock. Outer loops will span multiple hyperblocks. Our current implementation creates maximal hyperblocks.

After partitioning we distinguish two types of CFG edges: *internal edges:* whose ends are within the same hyperblock and *cross-edges:* connecting two different hyperblocks.

2.3 Block and Edge Predicates

The next compilation step analyzes each hyperblock separately. It associates a *predicate* to each basic block and control-flow edge. These predicates are similar to the full-path predicates from PSSA [6] and to the gating paths from GSA [17, 24]. Let us denote the set of CFG edges in a hyperblock by E[1].

The predicate associated to basic block b, denoted by $p(b)$, describes the condition under which b is on some (non-speculative) execution path starting at the entry point of the hyperblock. In other words, b is (non-speculatively) executed when $p(b)$ evaluates to "true". Similarly, the predicate $p(b, c)$ associated with a hyperblock edge b, c evaluates to "true" when the edge is on some non-speculative execution path starting at the entry point. Notice that these predicates are defined with respect to a hyperblock, and not with respect to the CFG.

The predicates are defined recursively, and can all be computed in $O(E)$ time in depth first order, as follows:

$p(hyperblock\ entry) = true$
  hyperblock entry is always executed (if the hyperblock itself is executed)
$p(x, y) = p(x) \wedge B(x, y)$
  edge (x, y) is executed if block x is executed and x branches to y
$p(y) = \vee_{(z,y) \in E}\, p(z, y)$
  block y is executed if some predecessor z branches to it

The predicates $B(x, y)$ correspond to the branch conditions in the code; their computation is part of the computation of the basic block x. If x branches to y unconditionally, $B(x, y) = true$.

Figure 1 shows a program fragment and its associated CFG, which is a hyperblock. Some of the predicates associated to the blocks and edges are:

[1] E comprises all internal edges as well as the cross-edges originating in the current hyperblock.

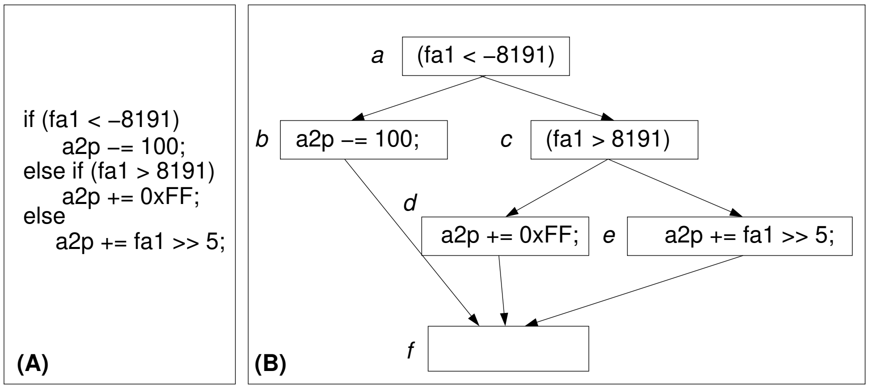

Figure 1: *(A) A code fragment (B) Its control-flow graph.*

$$
\begin{aligned}
p(a) &= True \\
B(a,b) &= (\texttt{fa1} < -8191) \\
p(a,b) &= p(a) \wedge B(a,b) = B(a,b) \\
B(a,c) &= \neg B(a,b) \\
p(a,c) &= p(a) \wedge B(a,c) = B(a,c) \\
p(b) &= p(a,b) \\
p(c) &= p(a,c) \\
B(c,d) &= (\texttt{fa1} > 8191) \\
p(c,d) &= p(c) \wedge B(c,d) = B(a,c) \wedge B(c,d) \\
p(d) &= p(c,d) \\
p(e) &= p(c,e) \\
B(b,f) &= B(d,f) = B(e,f) = True \\
p(b,f) &= p(b) \wedge B(b,f) = p(b) \\
p(d,f) &= p(d) \wedge B(d,f) = p(d) \\
p(e,f) &= p(e) \wedge B(e,f) = p(e) \\
p(f) &= p(b,f) \vee p(d,f) \vee p(e,f) = p(b) \vee (p(d) \vee p(e)) = p(b) \vee p(c) = True
\end{aligned}
$$

Predicate expressions are simplified using algebraic manipulations and by exploiting the CFG structure. The latter simplifications are based on the fact (noted in [11]) that if basic blocks a and b are control-equivalent in the hyperblock-induced CFG, then $p(a) = p(b)$ (by definition, a is control-equivalent to b if a dominates b and b post-dominates a). Notice that a and b can be control-equivalent in a hyperblock while not being control-equivalent in the procedure CFG. The reverse however is always true. In example 1, blocks f and a are control equivalent, and thus have the same predicate.

The predicate computations are synthesized as expressions explicitly represented in Pegasus. In our figures edges carrying predicate values are denoted by dotted lines.

2.4 Multiplexors

The next step in building the Pegasus graph is to convert each hyperblock into Static Single Assignment [8] form; that is, to connect variable definitions to uses. Pegasus explicitly represents the conditions under which each variable flows from a definition to a use; this is similar to other intermediate representations such as GSA [17], DFG [19], TGSA [12], or VDG [26]. We are currently

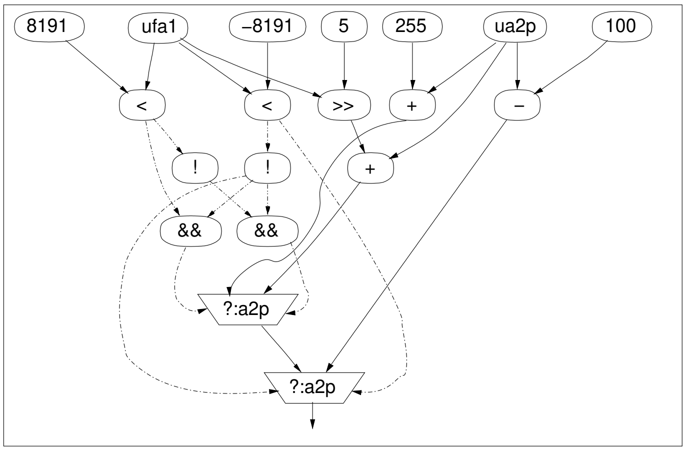

Figure 2: *Pegasus representation of the program fragment from Figure 1. The trapezoids are multiplexors; the dashed lines carry predicate values, while the solid lines carry data.*

using a simple implementation, which does not directly build a minimal SSA representation but adds multiplexors at all join points in the CFG. A subsequent optimization phase (see Section 4.2 and Figure 5) removes the redundant multiplexors. In practice, the performance of our implementation is very good, especially since the complexity is bounded by the hyperblock size, and not by the whole procedure CFG size; if necessary, we can implement the more efficient algorithm for multiplexor placement described in [24].

Pegasus explicitly uses *decoded multiplexors* instead of the SSA ϕ functions (our multiplexors correspond to the γ operations in GSA, PDW [17] and VDG). A decoded multiplexor selects one of n reaching definitions of a variable; it has $2n$ inputs: one data and one predicate input for each definition. When a predicate evaluates to "true", the multiplexor selects the corresponding input. Although this representation is redundant, its explicitness makes graph transformations during the optimization phase significantly simpler to express.

Figure 2 illustrates how edge predicates are used to drive the multiplexors.

2.5 Speculation

Speculative execution exposes ILP through the removal of control dependences [13] and reduces the control-height [22] of the code. Pegasus aggressively exploits speculation for these purposes.

Normally each instruction in block b is predicated on the value of $p(b)$, i.e. the instruction should be executed if and only if $p(b)$ evaluates to "true". Predicate promotion, described in [14], replaces the predicate guarding an instruction with a weaker one, causing the instruction to be executed even if the block including it is not. This is a form of control speculation, allowing the instruction to be executed before knowing the outcome of all controlling branches in the hyperblock.

In Pegasus we promote all instructions without side-effects or exceptions, by replacing their predicate with "true". This aggressive speculation enables many program optimizations, because all the speculated instructions become part of a large "straight-line" code region.

Some instructions cannot be speculatively executed; these are control-flow transfers (e.g., inter-hyperblock transfers, procedure calls and returns) and memory operations (including some reads, since reading an illegal address can cause an exception). For these instructions we keep the controlling predicate unchanged.

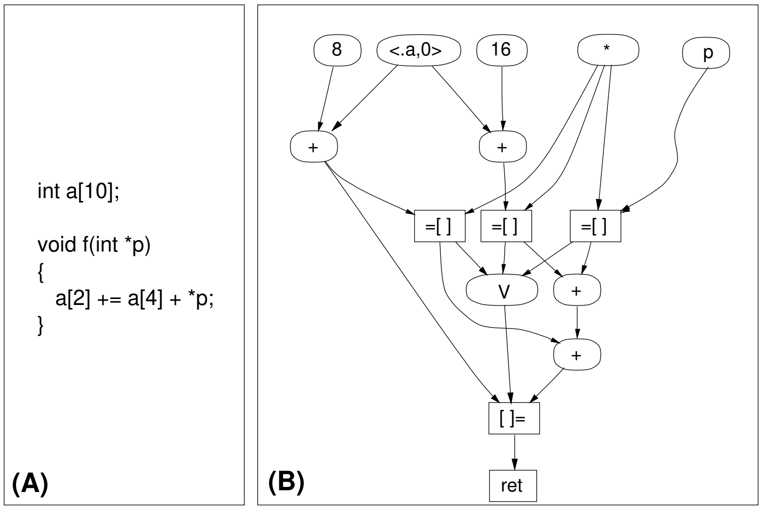

Figure 3: *(A) Program using arrays and pointers and (B) Pegasus representation. We have omitted all the predicates, which are constant "true". Note the parallel scheduling of all the load operations (denoted by* `=[ ]` *) and the dependence from all of them to the store (denoted by* `[ ]=`*, going through the combine "V" operator). The "*" node is the token input. The pointer analysis has assumed that* `p` *and* `a` *may alias.*

2.6 Tokens

As a result of the previous transformations we have completely removed all control-flow dependences from each hyperblock. The only constraint on the execution order of the operations is now given by the data dependences. However, not all data dependences are explicit: operations with side-effects could interfere with each other through memory. Pegasus has to represent information maintaining program order for operations whose side-effects do not commute. For this purpose we use a technique described in [19], adding edges between instructions that may depend on each other. Such edges do not carry data values, instead, they carry an explicit synchronization *token*. The presence of a token at the input is required for these instructions to be executed; on completion, these instructions generate an output token to enable execution of their dependents.

As described in the previous section, when multiple CFG paths join, we normally use a multiplexor to join the data generated in different predecessors. However, because tokens can have only one value, we can optimize their circulation by using a new type of operation, *combine*, instead of multiplexors. A combine operation (denoted by "V" in our figures) only computes on tokens. A combine has n inputs and one output; it generates an output when all inputs are present (at which time all the inputs are consumed).

Each Pegasus graph has an associated special token value, which is depicted by a "star" in our figures. These tokens are used to serialize the execution of consecutive hyperblocks: a hyperblock passes control to its successor, by sending it a token.

Pointer analyses can help reduce the number of synchronized pairs of instructions. Currently, we use the results of a simple intra-procedural points-to analysis: we insert a token edge between two instructions only if the their points-to sets overlap and they do not commute. This method of handling may-dependences is related to the one used in [23] to represent location-sets together with memory access operations.

```
int fib(int k)
{
    int a=0;
    int b=1;
    while (k) {
        int tmp = a;
        a = b;
        b = b + tmp;
        k--;
    }
    return a;
}
```

(A)

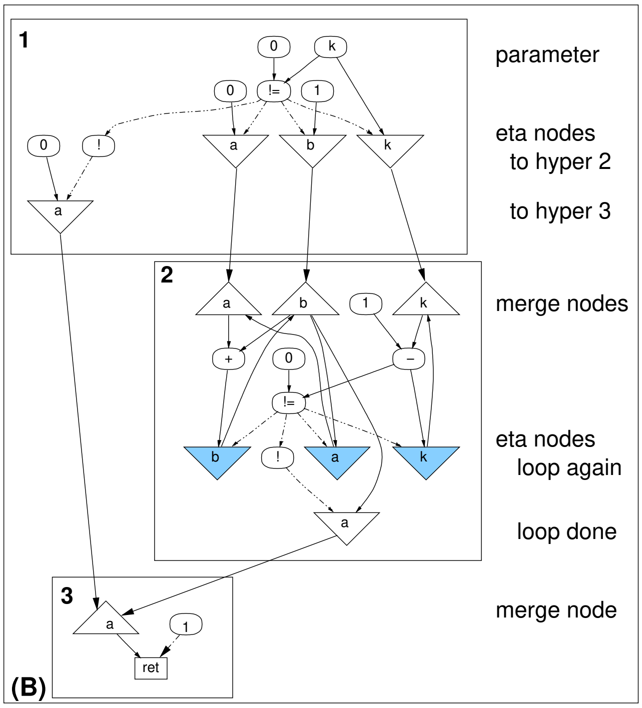

Figure 4: *(A) Iterative C program computing the k-th Fibonacci number and (B) its Pegasus representation comprising 3 hyperblocks. We are not representing the token nodes for simplicity.*

Figure 3 illustrates the translation of a C program with pointers and arrays. Because the store operation depends on all the loads, a *combine* operation has been inserted, with the loads as sources, and with the store as destination.

2.7 Control-flow and Cross-Edges

The compilation phases described so far deal with hyperblocks in isolation. This phase stitches the hyperblocks together into a Pegasus graph representing the whole procedure by creating dataflow edges connecting each hyperblock to its successors, i.e., it synthesizes objects corresponding to the CFG cross-edges. A special instance of cross-edges are loop back-edges, which are also synthesized in this step.

Cross-edges are just like all other regular dataflow edges, but each is cut by a special *eta*-node. Eta nodes were originally introduced in the Gated-Single Assignment form [17]; they are depicted by triangles pointing down (see Figure 4). Eta-nodes have two inputs—a value and a predicate—and one output. When the predicate evaluates to "true", the input value is moved to the output; when the predicate evaluates to "false", the input value and the predicate are simply consumed, generating no output.

Hyperblocks with multiple predecessors may receive control from one of several different points. To indicate such join points we use *merge* nodes, shown as triangles pointing up.

Figure 4 illustrates the representation of a program comprising multiple hyperblocks, including a loop. The eta nodes in hyperblock 1 will steer data to either hyperblock 2 or 3, depending on the test `k != 0`. Note that the etas going to hyperblock 2 are controlled by this predicate, while the eta going to hyperblock 3 is controlled by the complement. There are merge nodes in hyperblocks 2 and 3. The ones in hyperblock 2 accept data either from hyperblock 1 or from the back-edges in hyperblock 2 itself. The back-edges denote the flow of data along the "while" loop. The merge node in hyperblock 3 can accept control either from hyperblock 1 or from hyperblock 2. The constant "1" feeding the "return" instruction is a predicate, showing that the return is unconditionally executed, as soon as its input data is available.

The merge and eta nodes essentially transform control-flow into data-flow. The Pegasus IR tends to be more verbose than the classical CFG-based representations because the control-flow for each variable is represented separately. Notice for example how in hyperblock 2 of Figure 4 there are eta and merge nodes for all three loop induction variables, `a`, `b` and `k`. There are many benefits in this explicit representation, which naturally exposes loop-level parallelism; see Section 5 for an example on how this parallelism is exploited by our C to hardware compiler.

3 Pegasus Semantics and Lenient Evaluation

In this section we describe some aspects of the Pegasus semantics that were not covered previously. See the Appendix for a formal definition.

Most Pegasus operators are strict, i.e. they require all their inputs in order to generate an output. An exception is "merge", which generates the output with only one input present. All operators consume their inputs when computing a result. An "eta" consumes its inputs without producing anything, if the controlling predicate is "false".

The semantics explicitly precludes an operator from computing unless all its previous output values have been consumed. The producer of data must signal that data is valid, while the consumer(s) must signal that they have received the data, i.e., that the channel can be reused.

This type of operational semantics, where data is explicitly produced and consumed, is also used in some types of asynchronous hardware [9] (this protocol is essentially the Two-Phase Bundled Data convention) and in dataflow machine architectures [25].

Most arithmetic operations need to have all their inputs before they can decide the output value. However, some operations can compute their output earlier, for some special values of the inputs. A classical example are the boolean "and" and "or" operations: if one operand of an "and" is zero, the value of the other one is irrelevant. The generation of the output based only on some inputs is called *lenient evaluation*.

Several basic operations in our computational model can be evaluated in this way: the boolean operations and the multiplexors (as soon as a selector evaluates to "true" and the corresponding input is available a multiplexor can generate the output). Our semantics also allows for lenient evaluation of all predicated operations: as soon as the guarding predicate is known to evaluate to "false", the side-effect is not produced, and a *don't care* result can be generated, together with a token.

Well-structured Pegasus graphs compute the same values irrespective of whether the lenient or strict definition is used for operations. In practice lenient implementations reduce the dynamic critical path length, speeding up the computation.

4 Optimizations

Pegasus's strength is apparent when we examine how it is used by a compiler to implement optimizations. The advantages of our representation are the following:

- Pegasus represents a lot of useful "global" information locally, making many dataflow analyses unnecessary.

- Pegasus is self-contained. Thus, it is not necessary to incrementally update dataflow properties during program optimization.

- The formal semantics allows us to reason about the correctness of the optimizations. An optimization is correct if it does not change the outcome of the computation. The notion of outcome can be defined formally. We relegate such proofs to future work.

In this section we discuss informally most of the scalar compiler optimizations described by Muchnick [16]. We can classify these into three categories:

- Unchanged optimizations, performed in the same way as in a traditional setting.

- "Free" optimizations, that happen automatically during the construction of the Pegasus graph.

- Efficient optimizations, which are substantially easier to describe and implement using Pegasus, compared to other representation forms, as described in much of the related work and in [16].

The first type of optimization does not require global information and thus does not benefit from the increased expressiveness of Pegasus. It includes constant folding, strength reduction, re-association and loop-induction variable optimizations.

Before reviewing the optimizations it is important to note that Pegasus converts all control flow into data flow, so some optimizations no longer strictly correspond to their original definition. For example, straightening is an operation that chains multiple basic blocks in a single basic block under some appropriate conditions. However, there are no basic blocks in Pegasus, so we cannot speak of straightening any longer. However, we define a transformation $\mathcal{S}$ on Pegasus and claim it is equivalent to straightening, in the following sense: if we apply straightening on the source and next translate to Pegasus we obtain the same result as if we translate the source to Pegasus and next apply $\mathcal{S}$. In other words, the following diagram commutes:

$$\begin{tikzcd}
\text{source} \arrow[r] \arrow[d, "\text{Straightening}"'] & \text{Pegasus} \arrow[d, "\mathcal{S}"] \\
\text{optimized source} \arrow[r] & \text{optimized Pegasus}
\end{tikzcd}$$

4.1 Free Optimizations

These optimizations are implicitly executed during the creation of the Pegasus representation. They do not require any code to be implemented.

Global copy and constant propagation occur automatically during the construction of the SSA form.

Straightening and branch chaining occur because unconditional branches are transformed into constant "true" predicates which are removed from all path expressions by boolean formula simplifications.

Unreachable code elimination occurs during the covering with hyperblocks, because the depth-first traversal does not assign unreachable blocks to any hyperblock.

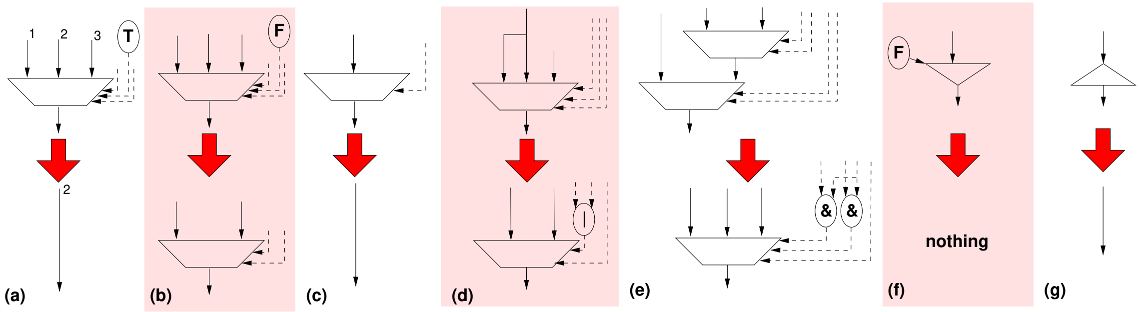

Figure 5: *Some new algebraic simplifications: (a) multiplexor with constant "true" predicate, (b) multiplexor with constant "false" predicate (c) multiplexor with only one input, (d) multiplexor with two identical inputs, (e) chained multiplexors, (f) "eta" with constant "false" predicate, (g)* merge *with a single input.*

4.2 Efficient Optimizations

These optimizations can be expressed very concisely on the Pegasus form. In fact *the description of each optimization given below is complete and unambiguous*. The actual code implementing these optimizations is extremely compact and easy to understand: all the optimizations below, minus loop-invariant code motion, which we haven't yet implemented, but including constant folding and strength reduction are implemented in less than 1000 lines of C++ code.

Dead code elimination is a trivial optimization: each side-effect-free operation whose output is not connected to another operation can be deleted.

Removal of `if` statements with empty branches occurs due to predicate simplification and dead-code elimination. Basic blocks before and after an empty conditional are control-equivalent, thus the conditional expression from an empty `if` will not appear in the expression for the selectors of any multiplexor and thus becomes dead code.

Constant conditionals in `if` statements are eliminated by a special case of algebraic simplification for multiplexors: these translate into constant selectors for a multiplexor, which can be simplified as in Figure 5(a,b).

Further unreachable code elimination following simplifications of constant "if" conditions is achieved by removing the hyperblocks having *merge* nodes with no inputs.

SSA form minimization is obtained after repeatedly simplifying multiplexors as shown in Figure 5(b,c,d).[2]

Global common-subexpression elimination, value numbering, partial redundancy elimination, redundant memory operation removal are actually all the same operation, due to the speculation and copy-propagation. The algorithm to implement all of these is trivial: two identical operations can be merged into one if all their corresponding in-edges originate from the same sources (e.g., they have the same source for the first input and the same source for the second input).[3]

[2] A minimal SSA representation results for each individual hyperblock; the form is not globally minimal.

[3] A linear time implementation can be obtained by hashing nodes based on their inputs.

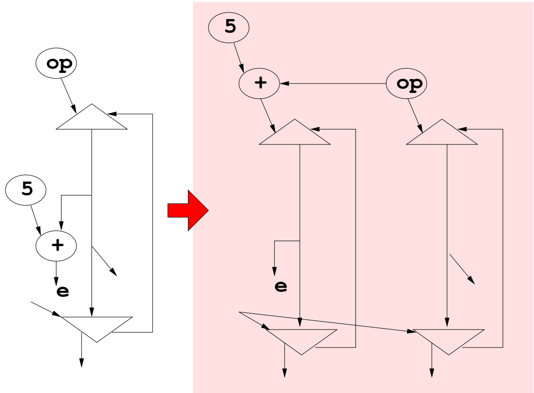

Figure 6: *Loop-invariant code motion.*

Even operations with side-effects (memory loads and stores, function calls and returns) that have identical inputs (including the predicate) are coalesced by common-subexpression elimination.

Code hoisting and tail merging are subsumed by common subexpression elimination, because expressions which compute the same value in different basic blocks of the same hyperblock become identical after predicate promotion.

Stored value forwarding can be coded as a simple adaptation of common-subexpression elimination: a load immediately dependent on a store to the same address (and sharing the same predicate) can be replaced with the stored expression.

Loop-invariant code motion is illustrated in Figure 6. A loop-invariant expression is defined [16] as (1) a constant, (2) an expression not changed within the loop, or (3) a computation having all inputs loop-invariant. In Pegasus, expressions of type (2) are represented as a cycle *merge–eta* without intervening operations.

Finding the maximal loop-invariant code then amounts to (a) marking all constant and simple loop-invariant expressions and (b) traversing the graph forward and marking the expressions reachable only from already marked expressions. Moving a loop-invariant expression before the loop requires the creation of a new *merge–eta* cycle, which carries around the value of the expression.

5 Compiling C to Hardware

Pegasus is the intermediate representation of the CASH compiler, which translates C programs into hardware circuits. CASH is built using the SUIF 1.3 compiler framework [27]. CASH's back-end does not yet generate hardware for creating stack frames, thus we currently translate only programs without recursive procedures. Support for these is in progress; when completed, this compiler will handle the complete ANSI C language.

The compiler builds a Pegasus representation for the source program starting from the low-Suif intermediate representation, which is akin to three-operand code. We perform a simple flow-sensitive intra-procedural pointer analysis before constructing Pegasus. Then each procedure is transformed into a Pegasus graph which is optimized as described in Section 4.

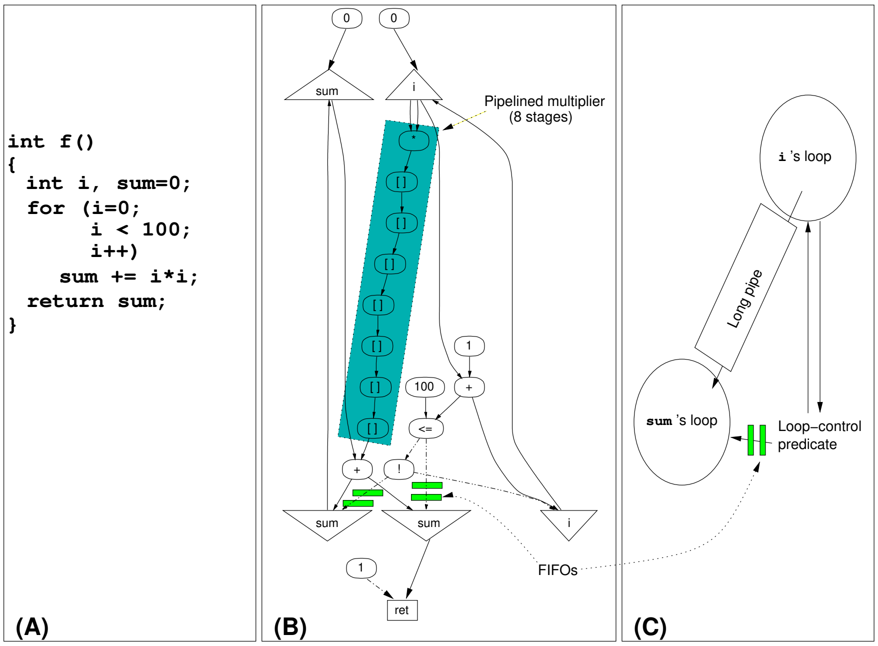

Figure 7: *(A) C program (B) its implementation using an 8-stages pipelined multiplier (C) abstract loop structure.*

In the most straightforward translation, the resulting intermediate representation is almost in a one-to-one correspondence with the target hardware. Each Pegasus operator gives rise to a corresponding hardware circuit computing the same function. Each operator also has an output register, used to store the result of the computation until the output channel is free. Handshaking between producers and consumers is synthesized using a simple finite-state machine.

The boolean operations and multiplexors are implemented leniently; their finite-state control is thus slightly different from the other operators. The result of using lenient computations is that the dynamic critical path is not the same as the static critical path. In this way, we automatically solve one of the main problems of speculated implementations, the imbalance of control-paths [3].

Because each induction variable has a separately synthesized loop, the hardware implementation is naturally pipelined, and can execute multiple loop iterations at once. Figure 7 shows how pipelining is achieved. The computation of the loop induction variable `i` is very fast, and can proceed in advance of the rest of the computation. The `i`-loop feeds a second data item to the multiplier before the first result is computed.

Unfortunately the circuit in Figure 7 does not fully exploit the pipelining potential. The problem stems from the loop-control predicate value, which controls whether looping should continue. The loop-control predicate is computed based on `i`'s value. The predicate value is needed by `sum`'s loop as well. This causes a throttling of the `i` loop, because the predicate computation circuit cannot accept new inputs as long as its previous outputs haven't been consumed.

We can decouple the two loops by using a FIFO buffer to store the consecutive predicate values, in this way allowing `i`'s loop to advance further ahead and fill the multiplier pipeline. The simulated performance of the resulting circuit grows with the depth of the inserted FIFOs, until the complete multiplier pipeline is full. At this point the `i` loop is several iterations ahead of the `sum` loop, although they were produced from the same original loop. Notice that by using FIFOs the compiler has a certain control on the amount of exposed parallelism. The depth of the FIFOs corresponds to the k-bounded loops of dataflow machines [7].

6 Related Work

The ideas in this paper unify significant prior work in five different areas: (a) dataflow machine architecture, (b) predicated execution, (c) static single-assignment, (d) intermediate program representation forms, and (e) hardware synthesis. It is difficult to fairly enumerate every related contribution in all these subjects; we cite some of the most influential.

For a survey of the dataflow machine literature, see for instance [25]; most notably, the various types of operations we propose (merge, eta), and the semantics of Pegasus are influenced by this work.

From the area of predicated execution we borrow and extend the ideas of hyperblock [15], if-conversion, and the combination of predication and speculation through predicate promotion [14, 2].

Static Single Assignment has been introduced in [1, 21, 8]; variations on this theme that introduce explicit predicates into the ϕ/γ functions include the Gated Single Assignment and Program Dependence Web [17], Thinned Gated Single Assignment [12], and Value Dependence Graphs [26].

The idea of program-representation transformations preserving program semantics has been suggested, for example, in the context of the Program Dependence Graph [18].

Some of the connections between the enumerated areas have been explored before: for example, Dependence Flow Graphs [19] interpret the intermediate representation as a dataflow machine (which is a particular case of our representation when hyperblocks are reduced to basic blocks), while Predicated Static Single Assignment [5, 6] gives a uniform treatment of predicates and SSA in the context of the hyperblock.

Using hyperblocks as a unit of speculation for synthesizing hardware has been done in the context of the Garp reconfigurable hardware project and compiler [4]. Our compiler CASH extends this work by generating dynamically scheduled hardware circuits, and compiling whole programs, including procedure calls.

7 Conclusions

We have presented Pegasus, a novel intermediate program representation, which combines static single assignment with predication using explicit predicate representations. Pegasus encodes in a compact way important program properties enabling efficient implementations of many program optimizations.

- The Pegasus representation is sparse (it has the same space complexity as static-single assignment).
- All optimizations presented in Section 4 take linear time in the size of the representation.
- The complete implementation of all the described optimizations required less than 1000 lines of C++ code.

We have shown how Pegasus can be given precise semantics, modeling the program as an asynchronous dataflow machine. We have shown the versatility of Pegasus by using it as the intermediate representation of an optimizing compiler which translates ANSI C programs into hardware implementations.

References

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *In Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages POPL*, pages 1–11, 1988.

[2] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, June 1998.

[3] D. I. August, W. mei W. Hwu, and S. A. Mahlke. A framework for balancing control flow and predication. In *Proceedings of the 30th International Symposium on Microarchitecture*, December 1997.

[4] T. J. Callahan and J. Wawrzynek. Instruction level parallelism for reconfigurable computing. In Hartenstein and Keevallik, editors, *FPL'98, Field-Programmable Logic and Applications, 8th International Workshop, Tallinin, Estonia*, volume 1482 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1998.

[5] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Predicated static single assignment. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, October 1999.

[6] L. Carter, E. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming, special issue*, 28(6), 2000.

[7] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of ISCA 1988*, pages 141–150, 1988.

[8] 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*, 13(4):451–490, 1991.

[9] A. Davis and S. M. Nowick. An introduction to asynchronous circuit design. Technical Report UUCS-97-013, Computer Science Department, University of Utah, September 1997.

[10] E. Gansner, E. Koutsofios, and S. North. GraphViz graph visualisation tool. http://www.graphviz.org.

[11] D. Gillies, D. Ju, R. Johnson, and M. Schlansker. Global predicate analysis and its application to register allocation. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 114–125, 1996.

[12] P. Havlak. Construction of thinned gated single-assignment form. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 477–499, Portland, Ore., 1993. Berlin: Springer Verlag.

[13] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of 19th International Symposium on Computer Architecture*, 1992.

[14] S. A. Mahlke, R. E. Hauk, J. E. McCormick, D. I. August, and W. mei W. Hwu. A comparison of full and partial predicated execution support for ILP processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture ISCA 22*, pages 138–149, Santa Margherita Ligure, Italy, May 1995. ACM.

[15] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 45–54, Dec 1992.

[16] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc, 1997.

[17] K. J. Ottenstein, R. A. Ballance, and A. B. Maccabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the Conference on Programming Language Design and Implementation PLDI 1990*, pages 257–271, 1990.

[18] R. Parsons-Selke. A rewriting semantics for program dependence graphs. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[19] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: An algebraic approach to program dependencies. In *Proceedings of Principles of Programming Languages POPL*, volume 18, 1991.

[20] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, 1981.

[21] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages POPL*, January 1988.

[22] M. Schlansker, V. Kathail, and S. Anik. Height reduction of control recurrences for ILP processors. In *Proceedings of the 27th annual international symposium on Microarchitecture MICRO*, pages 40–51, 1994.

[23] B. Steensgaard. Sparse functional stores for imperative programs. In *In Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, pages 62–70, 1995.

[24] P. Tu and D. Padua. Efficient building and placing of gating functions. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 47 – 55, 1995.

[25] A. H. Veen. Dataflow machine architecture. *ACM Computing Surveys*, 18 (4):365–396, 1986.

[26] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value Dependence Graphs: Representation without taxation. In *Proceedings of the Twentyfirst Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 297–310, January, 1994.

[27] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *ACM SIGPLAN Notices*, volume 29, pages 31–37, December 1994.

A Formal Semantics

In this section we describe precisely the meaning of the basic constructs of Pegasus. We use a Plotkin-style operational semantics [20].

We denote the set of program values by V, (values manipulated by the actual program: integers, floating point numbers, etc.). To this set we add some distinguished special values:

- the booleans $\mathbf{T}$ and $\mathbf{F}$;
- an abstract token value τ;
- a *don't care* value $\triangle$;
- a special "undefined" value, denoted by $\perp$;
- *node names*: are also values in order to implement "call" and "return" operations: the `call` must name the parameter nodes of the called procedure, while the `return` indicates the `call` node to which control is returned;

- a **wait** value, used for implementing lenient evaluation (Section 3).

A state associates to each edge $e \in E$ a value in V; that is, a state is a mapping $\sigma : E \to V$. We denote the fact that edge e has value v by $\sigma(e) = v$. A computation is defined by a state transition. We describe the semantics of each operation by indicating: (a) when the computation can be carried and (b) how it changes the state. We denote by

$$\textit{op} \qquad \frac{\textit{precond}}{\textit{change}}$$

the fact that operation *op* requires precondition *precond* to be executed and then incurs the *change* modification to the state. The transition relation is nondeterministic: at some point several operations may be eligible for execution and the semantics does not specify the execution order. It is expected that well-formed programs have a confluence property, i.e. the result is the same irrespective of the execution order.

We write $\sigma' = \sigma[e \mapsto v]$ to indicate a state σ' which coincides with σ, except that it maps e to v.

If node N has inedges i_1 and i_2 and outedge o, we write this as $o = N(i_1, i_2)$. We use $addr$ to denote values which represent memory addresses; we write $\texttt{update}(addr, v)$ to indicate that value v is stored in memory address $addr$, and $\texttt{lookup}(addr)$ for the contents of memory location $addr$.

We use suggestive edge names: p denotes predicates, t tokens, i and o input and respectively output edges, *pc* edges whose values denote node names (from Program Counter). For example, $\sigma(pc) =$ main indicates that *pc* is the node representing the "main" procedure. In this case, we denote by $\sigma(pc).arg_1$ the first argument of "main", $\sigma(pc).t$ the token argument of "main", etc.

For simplicity we allow in this model only a fanout of one for all nodes, except a special "fanout" node, which can have an unlimited number of outputs, and which copies the input to all the outputs.

In the initial state all values are undefined, i.e. $\forall e.\sigma(e) = \bot$.

We abbreviate $\sigma(e) \neq \bot$ with def(e), (i.e. the edge e has a "defined" value), and erase(a) as a shorthand for $[a \mapsto \bot]$ (i.e., a state in which a becomes undefined). We abuse the notation and use def() and erase() with multiple arguments.

$$o = \text{un\_op}(i) \quad \frac{\text{def}(i), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \text{un\_op}(\sigma(i))] \circ \text{erase}(i)}$$

$$o = \text{bin\_op}(i_1, i_2) \quad \frac{\text{def}(i_1, i_2), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \text{bin\_op}(\sigma(i_1), \sigma(i_2))] \circ \text{erase}(i_1, i_2)}$$

$$o_1, \ldots, o_n = \text{fanout}(i) \quad \frac{\text{def}(i), \neg\text{def}(o_1, \ldots, o_n)}{\sigma' = \sigma[o_1 \mapsto \sigma(i)][\ldots][o_n \mapsto \sigma(i)] \circ \text{erase}(i)}$$

$$o = \text{constant} \quad \frac{\neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \text{constant}]}$$

$$o = \text{merge}(i_1, \ldots, i_n) \quad \frac{\exists k.\text{def}(i_k), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \sigma(i_k)] \circ \text{erase}(i_k)}$$

All Pegasus graphs maintain the invariant that exactly one of the inputs of a *merge* can be present at a time.

$$o = \text{eta}(p, i) \quad \frac{\sigma(p) = \mathbf{T}, \text{def}(i), \neg\text{def}(o)}{\sigma' = \sigma[o \mapsto \sigma(i)] \circ \text{erase}(i, p)}$$

$$o = \text{eta}(p, i) \quad \frac{\sigma(p) = \mathbf{F}, \text{def}(i)}{\sigma' = \sigma \circ \text{erase}(i, p)}$$

$$t_0 = \text{combine}(t_1, \ldots, t_n) \quad \frac{\neg\text{def}(t_0), \text{def}(t_1, \ldots, t_n)}{\sigma' = \sigma[t_0 \mapsto \tau] \circ \text{erase}(t_1, \ldots, t_n)}$$

$$o = \text{mux}(i_1, p_1, \ldots, i_n, p_n) \quad \frac{\neg\text{def}(o), \text{def}(i_1.p_1, \ldots i_n, p_n), \exists k.\sigma(p_k) = \mathbf{T}}{\sigma' = \sigma[o \mapsto \sigma(i_k)] \circ \text{erase}(i_1, p_1, \ldots, i_n, p_n)}$$

All Pegasus graphs maintain the invariant that no more than one selector of a mux can be $\mathbf{T}$ at one time.

$$o = \text{mux}(i_1, p_1, \ldots, i_n, p_n) \quad \frac{\neg\text{def}(o), \text{def}(i_1.p_1, \ldots i_n, p_n), \forall k.\sigma(p_k) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \triangle] \circ \text{erase}(i_1, p_1, \ldots, i_n, p_n)}$$

All selectors can be $\mathbf{F}$.

$$t_0 = \text{store}(addr, p, v, t) \quad \frac{\neg\text{def}(t_o), \text{def}(addr, v, t), \sigma(p) = \mathbf{T}}{\sigma' = \sigma[t_0 \mapsto \tau] \circ \text{erase}(addr, p, t, v), \texttt{update}(\sigma(addr), \sigma(v))}$$

$$t_0 = \text{store}(addr, p, v, t) \quad \frac{\neg\text{def}(t_o), \text{def}(addr, v, t), \sigma(p) = \mathbf{F}}{\sigma' = \sigma[t_0 \mapsto \tau] \circ \text{erase}(addr, p, t, v)}$$

$$(o, t_0) = \text{load}(addr, p, t) \quad \frac{\neg\text{def}(o, t_0), \sigma(p) = \mathbf{T}, \text{def}(t, addr)}{\sigma' = \sigma[o \mapsto \texttt{lookup}(addr)][t_0 \mapsto \tau] \circ \text{erase}(addr, p, t)}$$

$$(o, t_0) = \text{load}(addr, p, t) \quad \frac{\neg\text{def}(o, t_0), \sigma(p) = \mathbf{F}, \text{def}(t, addr)}{\sigma' = \sigma[o \mapsto \triangle][t_0 \mapsto \tau] \circ \text{erase}(addr, p, t)}$$

$$o = \text{input} \quad \frac{\neg\text{def}(o), \text{def}(\text{input})}{\sigma' = \sigma[o \mapsto \sigma(\text{input})] \circ \text{erase}(\text{input})}$$

$$\text{return}(i, t, p, pc) \quad \frac{\text{def}(i, t, pc), \sigma(p) = \mathbf{T}}{\sigma' = \sigma[\sigma(pc).o \mapsto \sigma(i)][\sigma(pc).t \mapsto \tau] \circ \text{erase}(i, t, p)}$$

$$\text{return}(i, t, p, pc) \quad \frac{\text{def}(i, t, pc), \sigma(p) = \mathbf{F}}{\sigma' = \sigma \circ \text{erase}(i, t, p)}$$

$$(o, t_0) = \text{call}_k(t, p, pc, i_1, \ldots, i_n) \quad \frac{\text{def}(pc, t, i_1, \ldots, i_n), \sigma(p) = \mathbf{T}}{\begin{array}{c}\sigma' = \sigma[\sigma(pc).arg_1 \mapsto \sigma(i_1)][\ldots][\sigma(pc).arg_n \mapsto \sigma(i_n)] \\ \circ[\sigma(pc).t \mapsto \tau][\sigma(pc).pcin \mapsto \text{call}_k] \circ \text{erase}(i_1, \ldots, i_n, t, p, pc)\end{array}}$$

$$(o, t_0) = \text{call}_k(t, p, pc, i_1, \ldots, i_n) \quad \frac{\text{def}(pc, t, i_1, \ldots, i_n), \sigma(p) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \triangle][t_0 \mapsto \tau] \circ \text{erase}(i_1, \ldots, i_n, t, p, pc)}$$

Semantics for lenient evaluation; we present just formulas for the case of input 1 being available first.

$$o = \text{lenient\_or}(i_1, i_2)\ \frac{\sigma(i_1) = \mathbf{T}}{\sigma' = \sigma[o \mapsto \mathbf{T}][i_1 \mapsto \mathbf{wait}]}$$
$$o = \text{lenient\_or}(i_1, i_2)\ \frac{\text{def}(i_2), \sigma(i_1) = \mathbf{wait}}{\sigma' = \sigma \circ \text{erase}(i_1, i_2)}$$
$$o = \text{lenient\_and}(i_1, i_2)\ \frac{\sigma(i_1) = \mathbf{F}}{\sigma' = \sigma[o \mapsto \mathbf{F}][i_1 \mapsto \mathbf{wait}]}$$
$$o = \text{lenient\_and}(i_1, i_2)\ \frac{\text{def}(i_2), \sigma(i_1) = \mathbf{wait}}{\sigma' = \sigma \circ \text{erase}(i_1, i_2)}$$

For simplicity we show just a two-way multiplexor.

$$o = \text{lenient\_mux}(i_1, p_1, i_2, p_2)\ \frac{\neg\text{def}(o), \text{def}(i_1), \sigma(p_1) = \mathbf{T}}{\sigma' = \sigma[o \mapsto \sigma(i_1)][i_1 \mapsto \mathbf{wait}][p_1 \mapsto \mathbf{wait}]}$$
$$o = \text{lenient\_mux}(i_1, p_1, i_2, p_2)\ \frac{\neg\text{def}(o), \text{def}(i_2, p_2), \sigma(p_1) = \mathbf{wait}, \sigma(i_1) = \mathbf{wait}}{\sigma' = \sigma \circ \text{erase}(i_1, i_2, p_1, p_2)}$$