Filter Fusion

Todd A. Proebsting Scott A. Watterson

TR 95-11

Abstract

Filters are a common data-manipulation abstraction that read data from a single source and write data to a single destination. In filter applications, data flows from a source to a sink through intermediate filters. Logically, filters are separate, modular entities. We present a new compiler optimization, Filter Fusion, that eliminates the overhead of a modular design of independent filters. Our algorithm automates the integration of arbitrary, independently designed filters. `FFC`, our Filter Fusion compiler, composes filters and produces code that is as efficient as hand-integrated code. The optimized code can achieve up to a two-fold improvement over independent filters.

September 22, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

1 Introduction

Filters are a common data-manipulation abstraction in networking, operating systems, and simulation software. Filters read data from a single source and write data to a single destination. In filter applications, data flows from a source to a sink through intermediate filters. Logically, filters are separate, modular entities. Modular implementations unfortunately suffer a substantial performance penalty relative to integrated implementations. Where performance matters most, systems programmers will sacrifice the modular design for the greater speed of an integrated design.

We present a new compiler optimization, Filter Fusion, that eliminates the overhead of a modular design of independent filters. Our algorithm automates the integration of arbitrary, independently designed filters. `FFC`, our Filter Fusion compiler, composes filters and produces code that is as efficient as hand-integrated code. The optimized code can achieve up to a two-fold improvement over independent filters.

Network protocol layers are often filters. Typically, each protocol layer performs some data manipulation by traversing the message from beginning to end. Programmers have traditionally merged these filters by hand to produce efficient code. Integrating filters allows data to be read once, manipulated many times, and then stored once — thus avoiding loads and stores for each filter's manipulations. Excessive memory accesses cripple the performance of network code. Filter Fusion eliminates unnecessary memory accesses.

Manually integrating filters is a time-consuming, error-prone process. In addition, hand-integrated programs are difficult to maintain and modify because small changes in a single filter can result in global changes in the integrated program. `FFC` automates the integration process and therefore eliminates this concern. Furthermore, automatic integration enables the maintenance of a library of useful filters (protocol layers) that can be composed freely to develop specialized protocols. Each library component is maintained separately, and yet integration and optimization is automatic. The programmer designs and optimizes in a modular fashion, without sacrificing performance in the final composition.

While Filter Fusion is well suited for systems software applications, no assumptions about its problem domain are made. `FFC` places few restrictions on the filters it integrates — it handles arbitrary control flow and data manipulations within each each filter.

2 Background

`FFC` is part of the compiler suite of the Scout project [MMO+95]. Scout aims to deliver high-performance systems software — especially communications-oriented operating systems. The Scout compilers do non-traditional optimizations, like Filter Fusion, to increase software performance and to liberate the programmer from tedious, error-prone tasks [OPM94].

Network applications often require many simple manipulations of each network packet. These manipulations form the protocol stack. Redundant memory access can dominate the processing time for these applications. A technique called Integrated Layer Processing (ILP) optimizes these data manipulations [CT90]. ILP, a generalization of loop jamming or loop fusion, does increase performance [CT90, CJRS89, DAPP93].

Clark and Tennenhouse report dramatic performance improvements from ILP [CT90]. Based on their results, they argue for *less* modular programming — when efficiency is critical and sequential data manipulations are too costly, the programmer must abandon abstraction and merge protocols. By automating ILP, Filter Fusion allows the programmer to retain modular design without sacrificing performance.

Abbot partially automated ILP for network applications [Abb93]. His system has two significant drawbacks, however: it cannot handle arbitrary control-flow within a filter, and it assumes the typical network data layout that partitions *header* and *data*. His protocols had three stages: initial, data manipulation, and final. The integrated code performed the initial and final stages serially with only the data manipulation stages truly integrated. Not all protocols (e.g., message re-assembly), and certainly not all filters, fit into this framework. Filter Fusion has no such restrictions.

Filter Fusion is similar to *deforestation* [Wad90]. Deforestation transforms functional programs to eliminate intermediate trees; Filter Fusion transforms filters to eliminate intermediate arrays of data. Unlike deforestation, Filter Fusion operates on imperative programs.

These prior implementations have proven the efficacy of ILP, but they have not fully generalized or automated the optimization. Thus, a tension exists between modular software design and integrated high-performance implementation. `FFC`, an implementation of Filter Fusion, provides a solution. While maintaining a clean, intuitive model for protocol construction, it provides both modularity and performance.

3 Filters

A linear composition of filters specifies the path data will follow from source to sink:

$$Source \to Filter_1 \to Filter_2 \to \cdots \to Filter_N \to Sink$$

In a modular implementation, the source produces *all* of the data before passing it to the first filter. That filter then processes all the data before passing it to the next filter. This continues until the sink ultimately consumes the data. Unfortunately, this implementation requires that each filter read and write data. It is much more efficient to merge these filters to perform all the data manipulations at once.

3.1 Filter Specifications

A filter specification is simply a parameterless procedure extended by three operations: `put`, `get`, and `FILTER`. A `put` produces data for the next filter, and a `get` retrieves data from the previous filter. (Filter Fusion will merge filters so that matching `put`'s and `get`'s can be replaced by assignments.) `FILTER` is a special predicate that guides Filter Fusion. `FILTER` guards statements that either require more input or may produce more output. `FILTER` is explained further in section 5.

The first filter of a composition, the *source*, cannot contain any `get`'s. The last filter, the *sink*, cannot contain any `put`'s. Figure 1 contains source and sink filters for simple array reading and writing.

Data manipulation filters exist between the source and the sink. Typical filters may do encryption, compression, checksumming, or data marshaling (e.g., byte swapping). In addition, *glue* filters are useful for combining filters that may require special invariants. For instance, the simple filter for swapping pairs of adjacent bytes, `2ByteSwap`, requires an even

```
Filter ReadFromArray
Decls
   int i;
Code
   i = 0;
   while (i < 10000)
        put input[i];
        i++;
   end-while
End-Filter
```

```
Filter WriteToArray
Decls
   int j;
Code
   while FILTER
        get output[j];
        j++;
   end-while
End-Filter
```

Figure 1: Source and Sink Filters

```
Filter Evener
Decls
   int c, k;
Code
   k = 0;
   while FILTER
        get c;
        put c;
        k++;
   end-while
   if (k%2)
        put 0;
End-Filter
```

```
Filter 2ByteSwap
Decls
   int x, y;
Code
   while FILTER
        get x;
        get y;
        put y;
        put x;
   end-while
End-Filter
```

Figure 2: Sample Filters

number of bytes as input. The `Evener` is a glue filter that always writes an even number of bytes by simply copying its input to its output and conditionally appending a single zero. Thus, the `Evener` typically precedes `2ByteSwap` to ensure proper functioning. Figure 2 gives the specifications for `2ByteSwap` and `Evener`. Lightweight filter design encourages modular design and separation of concerns.

Typical network protocols such as CRC32 checksum and MD-5 encryption are also filters. Other functions we have implemented as filters include Run-length Decoding and Run-length Encoding, simple checksumming, and data marshaling. Filter Fusion allows the programmer to create arbitrarily complex compositions of these independently developed filters; `FFC` will integrate them into a single optimized function.

Efficiency and modularity are advantages of using `FFC`. Without `FFC`, reorganizing a protocol stack requires re-integrating the stack by hand. With `FFC`, reorganizing a stack simply requires changing the individual filters (if necessary) and specifying a new composition.

4 Sample Fusion

Filter Fusion is an optimization based on a symbolic execution of the filters. Filter Fusion integrates two filters — a *producer* and a *consumer* — at a time. The goal is to match the `put`'s of the producer with the `get`'s of the consumer and to replace them with assignments. Using dynamic programming, Filter Fusion follows all possible control flow paths through both filters while tracking the flow of values via the `put`'s and `get`'s. Filter Fusion composes the control-flow graphs of the filters into new, larger graph. Where necessary, Filter Fusion replicates filter code.

As an example, we will merge the `Evener` and the `2ByteSwap` filters in Figure 2. Figure 3 gives their control-flow graphs. Rectangles denote nodes from `2ByteSwap` throughout this example; ovals denote `Evener` nodes.

The final control-flow graph is composed of nodes from the two original graphs, except that the appropriate `put`'s and `get`'s are replaced with assignments to temporary variables. Basically, the dynamic programming executes each filter symbolically — alternating between the producer and consumer at `put`'s and `get`'s, respectively. For each node that is symbolically executed, a copy of that node is placed into the fused graph. Bookkeeping information maintained at each node of the final graph controls the composition. Each added node is annotated with three pieces of information: the last node executed in the producer, the last node executed in the consumer, and which filter this node came from. This information is a *configuration*. Two nodes are equal if their configurations are identical.

The producer symbolically executes until it reaches a `put` or `end` operation. After reaching a `put` in the producer, execution switches to the consumer, which must execute until it reaches a `get` (or `end`). The `put` that suspended the producer is matched with the consumer's `get` for subsequent replacement by an assignment. This alternating execution continues until all possible execution paths are exhausted.

The `FILTER` predicate will represent a conditional node in a control flow graph of either the producer or the consumer. The state of a suspended producer determines the value of a consumer's `FILTER` predicate. If a consumer is executing while the producer is suspended at a `put`, then `FILTER` evaluates to true; if the producer is suspended at its end, then `FILTER` evaluates to false. `FILTER` predicates in the producer remain undetermined.[1]

Figure 4 depicts the control flow of the fused filter after the producer has followed all possible paths to `put`'s or `end`'s. Symbolic execution must now switch to the consumer.

When expanding the consumer (`2ByteSwap`), the first node to be executed is a `FILTER` predicate. Thus, all three paths will add a `FILTER` node. On the left-most path, the producer had suspended at a `end`, but on the center and right-most paths, the producer suspended at a `put`. Therefore, consumer will continue along the false branch when expanding the left-most path, and it will continue along the true branch when expanding the others.

Along the left-most path, the consumer immediately encounters an `end` node. This path is complete. Along the other paths, the consumer, following the true branch, immediately hits a `get`. The `get` matches the suspended `put` of the producer, so execution suspends at the consumer and resumes at the producer along both paths. Figure 5 gives the flow graph

[1]This discussion assumes that the producer is driving Filter Fusion. If the consumer were driving Filter Fusion, then the `FILTER` predicates in the producer would be determined by whether or not the consumer were suspended at a `get`.

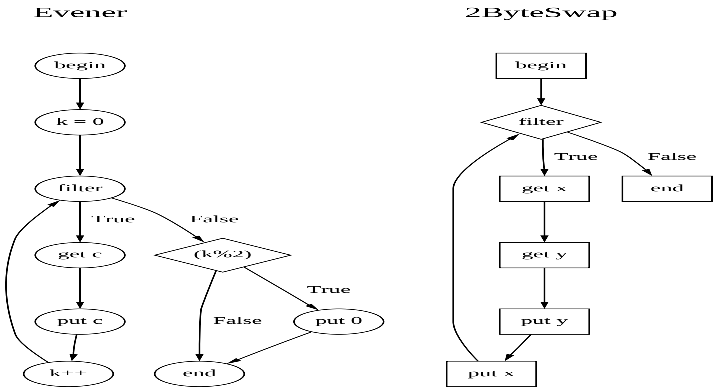

Figure 3: Original Control Flow Graphs

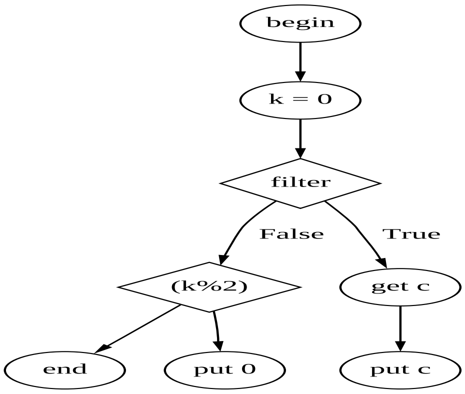

Figure 4: Stage 1

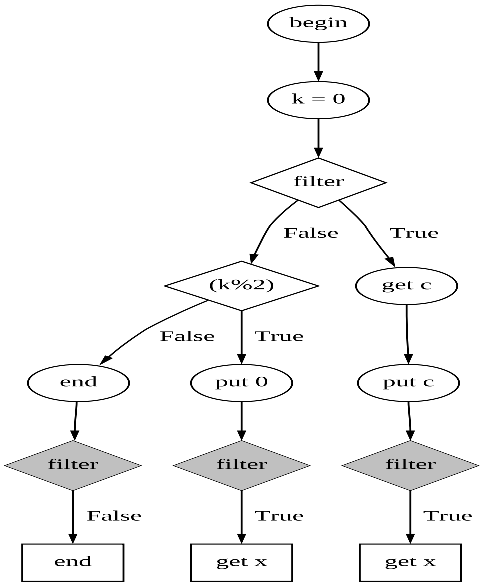

Figure 5: Stage 2

at this point.

The producer must now resume execution by exploring all possible control paths from its suspended `put`. Control continues to switch back and forth until no more progress can be made. A configuration labels each new node. Prior to adding a new node, its configuration is checked against the nodes already in the new graph — upon a match, the existing node is used rather than the new node. An existing node is re-used by having control flow directly to that node rather than to the new node.

Figure 6 shows the graph resulting from this composition. Filter Fusion is not finished at this point, however. Some paths reach a `get` without a corresponding `put`. These paths are removed from the control flow, since they make no sense. Trimming often creates a conditional for which only one branch remains — in these cases, we may remove the

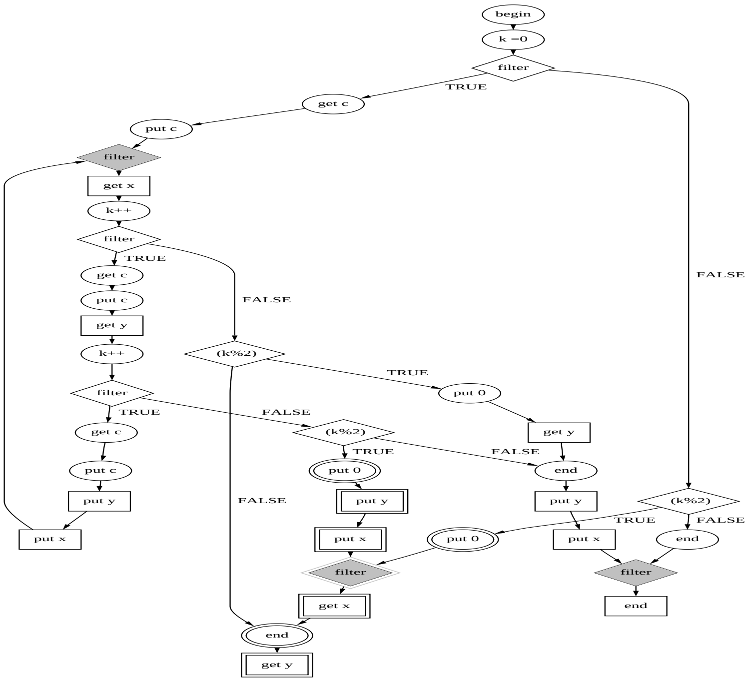

Figure 6: Untrimmed Control Graph

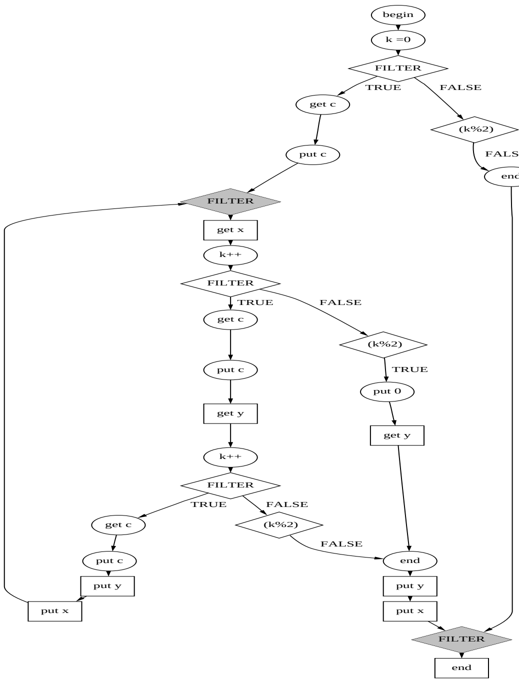

Figure 7: Final Control Graph

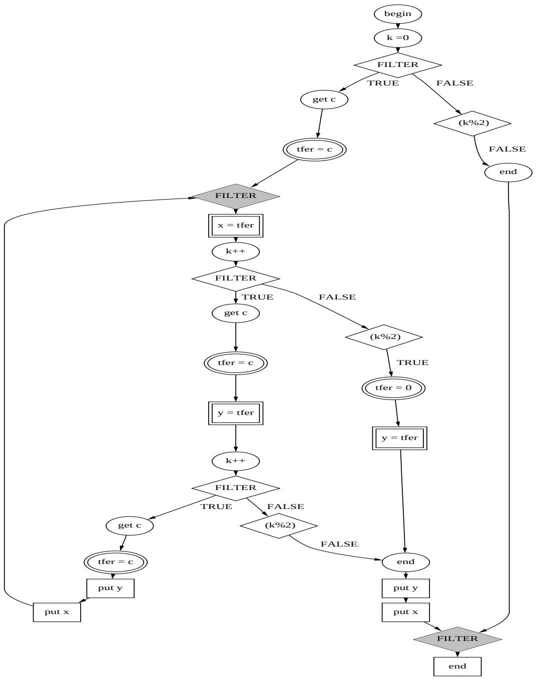

Figure 8: Assignment Substitution

conditional too. In general, trimming conditionals is an unsafe optimization. If, however, filters are properly composed such that `put`'s must always reach `get`'s (as they do here), the optimization can be both safe and effective. The nodes to be safely trimmed have double borders in Figure 6. Figure 7 gives the trimmed graph.

The final step of Filter Fusion is transforming the matched `put`'s and `get`'s into assignments to and reads from a temporary, respectively. The temporary is unique to a particular filter composition. Each suspended `put` that is copied into the composition graph becomes a write to the temporary, and all `get`'s become reads of the temporary. Figure 8 highlights the transformed nodes in the resulting graph with double borders.

5 Algorithm

Filter Fusion is done pairwise, starting with the source and its immediate consumer. Because the composition of a source and a general filter is itself a source, this method can compose arbitrarily many filters. (Filter Fusion can operate in the opposite direction too, but giving the less general algorithm here is simpler.)

`FFC` implements Filter Fusion with a work-list algorithm. Elements of the work-list represent configurations that have been added to the control-flow graph (CFG), but whose successors have not. The algorithm is responsible for computing the successors and adding them to the CFG and the work-list, when necessary. No computed configuration already in the CFG will be added to the work-list, since the previous instance can be reused in its place. This ensures termination. It also bounds number of nodes in the fused graph by the product of the number of nodes in the input graphs. (In practice, the code size will not increase to this maximum, particularly when merging filters with the same size data units.)

Figure 9 gives the algorithm. Let x be a CFG node. Its configuration is defined by *x.orig[producer]*, *x.orig[consumer]*, and *x.tag*. *x.orig[producer]* and *x.orig[consumer]* represent the last nodes visited in the two filters when this node was generated. *x.tag* indicates which filter generated this node. Additional attributes of x, *insn* and *successors*, denote the node's actual instruction and its CFG successors.

The algorithm begins by adding a *start* configuration that represents the initial nodes of each of the input graphs to both the CFG and the work-list. *start* will be the beginning node of the resulting graph. While elements remain in the work-list, they are removed one at a time, to compute their successors. Recall that successor nodes may or may not come from the same control flow graph as a node, x, itself (e.g., the successor of a `put` in the producer comes from the consumer, but the successor of a simple statement in the producer would also come from the producer). $trigger[producer]$ represents the set of nodes that cause control to switch from the producer to consumer, `put` and `end`. $trigger[consumer]$ is a set consisting only of `get`. "not tag" alternates between producer and consumer.

After computing the source of x's successors, the algorithm simply follows the control flow from the last executed statement in that source graph to find the actual successor instructions. Each successor has a configuration that is checked against the CFG to determine if it already exists. If the configuration already exists, the control flow arc out of x simply points to the existing configuration. If the configuration is new, it is added to the CFG and the work-list. The new configuration is also the target of the arc from x.

```
Procedure Fusion()
    start.orig[producer] := producer's start node      // Initialize start node's configuration
    start.orig[consumer] := consumer's start node
    start.tag := producer
    start.insn := empty instruction
    CFG := { start }                                   // Seed CFG and worklist.
    worklist := { start }
    repeat
        x := Pop(worklist)
        if x.insn ∉ trigger[x.tag] then                // {put,end} for producer; {get,end} for consumer.
            this := x.tag                              // Stay with current filter.
            other := not x.tag
        else                                           // Switch to other filter.
            this := not x.tag
            other := x.tag
        endif
        ∀i ∈ x.orig[this].successors do                // Follow all paths.
            node := new node
            node.orig[this] := i                       // Store current nodes.
            node.orig[other] := x.orig[other]
            node.tag := this                           // Tag which filter derived node.
            node.insn := i.insn
            if node ∉ CFG then
                CFG := CFG ∪ node
                Append(worklist, node)
                x.successors := x.successors ∪ node
            else                                       // Reuse existing node.
                x.successors := x.successors ∪ CFG[node]
            endif
        end ∀
    until worklist = φ
end Fusion
```

Figure 9: Algorithm

The algorithm describes the steps to compute the untrimmed graph. Trimming the graph of *dangling* `put` nodes is straightforward. Also, a little additional bookkeeping is necessary to transform `put`'s and `get`'s into assignment and reads of temporaries.

6 Experimental Results

`FFC` is a 200 hundred line Icon program [GG90]. `FFC` is a preprocessor that generates C code from a compact specification language. We tested `FFC`'s code against modular and hand-integrated implementations on a variety of platforms and compilers. Because `gcc`

Program	C size (in lines)	Alpha Binary Size (in bytes)	Sparc Binary Size (in bytes)
Modular Implementation	36	2,784	1,955
Hand Integrated	28	2,592	1,898
Filter Fusion	197	2,976	2,323
Fused & Tuned	144	3,040	2,127

Table 1: Code Size

consistently produced worse code than the vendor compilers, we aborted its use. (`gcc` had difficulty re-ordering basic blocks to avoid chains of jumps. It also did not handle copy propagation and dead-code elimination as well as the vendor compilers.)

To test `FFC`-generated code, we create the following composition.

`ReadFromArray` → `Evener` → `2ByteSwap` → `CRC32` → `WriteToArray`

These filters (1) read bytes from an array, (2) pad the output, (3) swap bytes, (4) compute CRC32 checksumming, and finally, (5) write the bytes to another array. Appendix A contains the specification for `CRC32` and the composition. Figure 10 gives the final flow graph. Note that some chunks of code are replicated multiple times and that the graph is quite complicated given the simple nature of its constituent filters. Table 1 shows the size of several fused filters on both a DEC Alpha and the Sun SPARCsystem 10. Since `FFC` may replicate the same code multiple times, the final fused filter may contain a great amount of C code. Although the C code produced by the Filter Fusion compiler was much larger than that of the modular and hand-integrated implementations, the object code sizes were very nearly comparable. Compiler optimizations eliminate much of the redundancy.

`FFC`-generated code must be optimized because of its heavy reliance on temporary variables and arbitrary control flow. The code particularly stresses — and finds deficiencies in — a compiler's copy propagation and dead code elimination optimizations. Unfortunately, in many cases, all of the available compilers failed to eliminate useless counters or to propagate copies. In addition, the compilers did not appear to unroll unstructured loops. Therefore, `FFC`-generated code's performance suffered. To determine how well `FFC`'s code would do if properly optimized, we performed these optimizations by hand on the generated code. (We only performed optimizations that we thought *any* optimizing compiler should have done.) We timed four different implementations of the five-filter composition: modular, hand-integrated, `FFC`-generated integration, and hand-tuned `FFC` integration. Table 2 gives the results of running these filters 10,000 times over a 10,000 element array. All tests were run on four different architectures using the vendor's C compilers.

`FFC`-generated output is always superior to modular code. `FFC`-generated output typically is slower than hand-integrated code, but only because of the C compiler's shortcomings. Filter Fusion allows the programmer to maintain a modular design and implementation without sacrificing performance.

This exhaustive computation of all possible execution paths is tedious and error-prone when done by hand. Fortunately, an implementation of Filter Fusion automates this transformation. Filter Fusion allows the programmer to forget about this complex work, and

Architecture	Fusion Technique			
	No Integration	Hand Integration	Filter Fusion	Filter Fusion with Tuning
DEC/Alpha	21.9	8.3	10.3	8.3
Sun/Sparc	26.2	12.0	18.5	12.1
HP/700	38.5	19.9	31.0	20.0
Mips R2000A	66.7	33.9	46.9	34.1

Table 2: Experimental Results (in sec.)

focus on optimizing independent filters in a modular fashion.

References

[Abb93] Mark B. Abbott. *A Language-Based Approach to Protocol Implementation.* PhD thesis, University of Arizona, 1993.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachusetts, 1986.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of tcp processing overhead. *IEEE Communications Magazine*, June 1989.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the SIGCOMM '90 Symposium*, pages 200–208, September 1990.

[DAPP93] Peter Druschel, Mark B. Abbott, Michael A. Pagels, and Larry L. Peterson. Network subsystem design: A case for an integrated data path. *IEEE Network Magazine*, July 1993.

[GG90] Ralph E. Griswold and Madge T. Griswold. *The Icon Programming Language.* Prentice Hall, 1990.

[MMO+95] A. B. Montz, D. Mosberger, S. W. O'Malley, L. L. Peterson, and T. A. Proebsting. Scout: A communications-oriented operating system. In *HOTOS95*, pages 58–61. IEEE Computer Society Press, may 1995.

[OPM94] Sean O'Malley, Todd A. Proebsting, and A. Brady Montz. USC: A universal stub compiler. In *Proceedings of SIGCOMM 94 Conference on Communications Architectures, Protocols and Applications*, pages 295–306, August 1994.

[Wad90] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

A Five-Filter Specification

The FFC specifications of the `Evener`, `2ByteSwap`, `ReadFromArray`, and `WriteToArray`. are given in the paper. The specification below describes CRC32, as well as for the composition used in the timings. Figure 10 shows the composition's flow graph.

```
Compose prodeven ← ReadFromArray Evener
Compose prodevenBS ← prodeven Byteswap
Compose prodevenBSCRC ← prodevenBS CRC32
Compose fulltest ← prodevenBSCRC WriteToArray

Filter CRC32
Decls
   unsigned long crc = 0;
   unsigned char idx;
   int tx = 0;
   unsigned char CRC32temp;
Code
   while filter
        get CRC32temp
        tx += 1;
        idx = (CRC32temp ∧ crc);
        idx &= 0xff;
        crc ≫= 8;
        crc ∧= crctable[idx];
        put CRC32temp
   endwhile
   put crc & 0xff
   put (crc ≫ 8) & 0xff
   put (crc ≫ 16) & 0xff
   put (crc ≫ 24) & 0xff
End-Filter
```

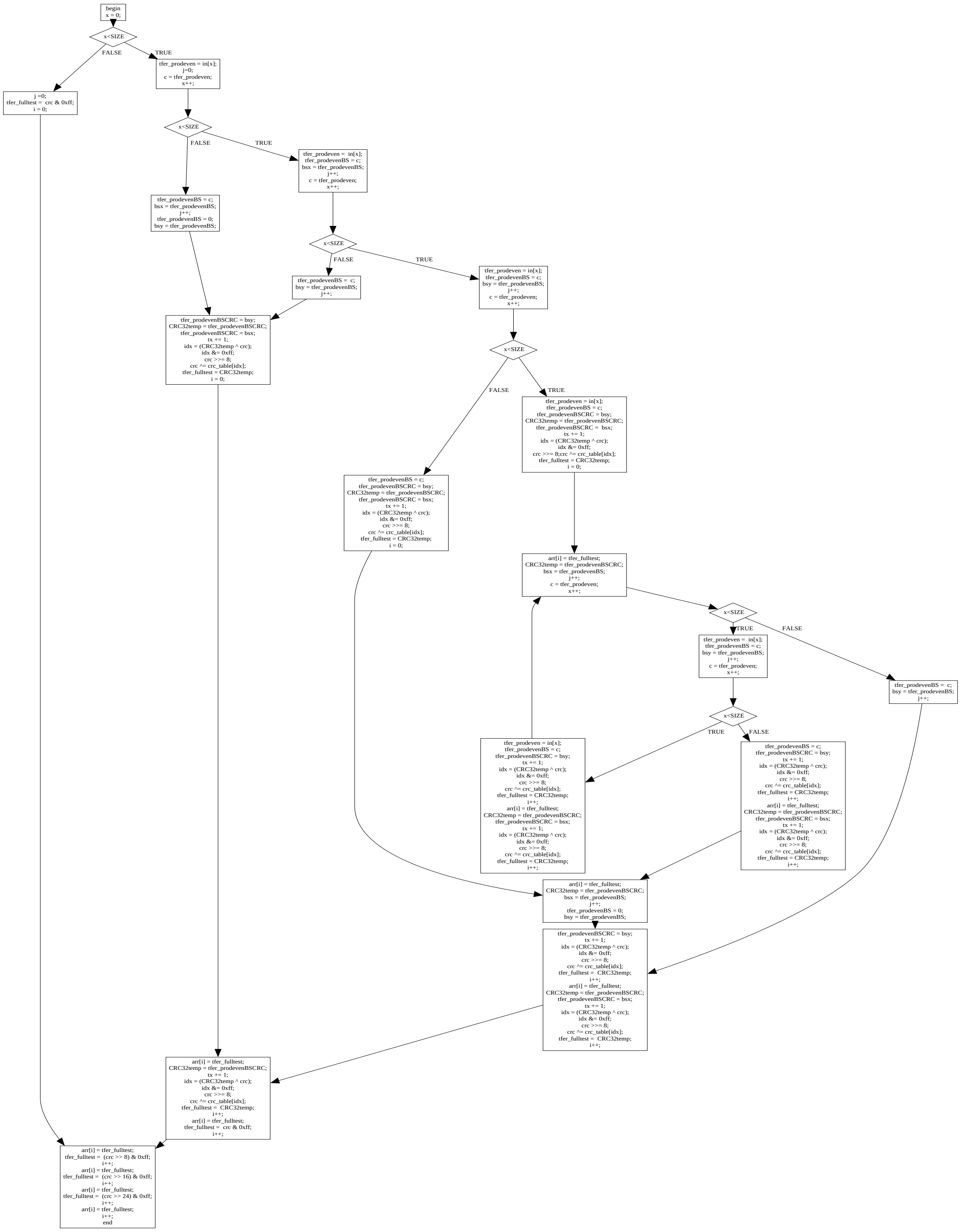

Figure 10: Final Composition