Programming by Sketching for Bit-Streaming Programs

Armando Solar-Lezama[1] Rodric Rabbah[2] Rastislav Bodík[1] Kemal Ebcioğlu[3]

[1]Computer Science Division, University of California, Berkeley
[2]Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology
[3]T.J. Watson Research Center, IBM Corporation
{asolar, bodik}@cs.berkeley.edu, rabbah@mit.edu, kemal@us.ibm.com

Abstract

This paper introduces the concept of *programming with sketches*, an approach for the rapid development of high-performance applications. This approach allows a programmer to write clean and portable reference code, and then obtain a high-quality implementation by simply *sketching* the outlines of the desired implementation. Subsequently, a compiler automatically fills in the missing details while also ensuring that a completed sketch is faithful to the input reference code. In this paper, we develop StreamBit as a sketching methodology for the important class of bit-streaming programs (e.g., coding and cryptography).

A sketch is a *partial* specification of the implementation, and as such, it affords several benefits to programmer in terms of productivity and code robustness. First, a sketch is easier to write compared to a complete implementation. Second, sketching allows the programmer to focus on exploiting algorithmic properties rather than on orchestrating low-level details. Third, a sketch-aware compiler rejects "buggy" sketches, thus improving reliability while allowing the programmer to quickly evaluate sophisticated implementation ideas.

We evaluated the productivity and performance benefits of our programming methodology in a user-study, where a group of novice StreamBit programmers competed with a group of experienced C programmers on implementing a cipher. We learned that, given the same time budget, the ciphers developed in StreamBit ran $2.5\times$ faster than ciphers coded in C. We also produced implementations of DES and Serpent that were competitive with hand optimized implementations available in the public domain.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Software Architectures, Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors; D.3.2 [*Programming Languages*]: Language Classifications

General Terms Languages, Design, Performance

Keywords Stream Programming, StreamIt, Synchronous Dataflow, Sketching, Domain Specific Language, Domain Specific Compiler

PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006...$5.00.

1. Introduction

Applications in domains like cryptography and coding often have the need to manipulate streams of data at the bit level. Such manipulations have several properties that make them a particularly challenging domain from a developer's point of view. For example, while bit-level specifications are typically simple and concise, their word-level implementations are often daunting. Word-level implementations are essential because they can deliver an order of magnitude speedup, which is important for servers where security-related processing can consume up to 95% of processing capacity [20]. Converting bit-level implementations to word level implementations is akin to vectorization, but the characteristics of bit-streaming codes render vectorizing compilers largely ineffective. In fact, widely used cipher implementations often achieve performance thanks to algorithm-specific algebraic insights that are not available to a compiler.

Additionally, correctness in this domain is very important because a buggy cipher may become a major security hole. In 1996, a release of the BlowFish cipher contained a buggy cast (from unsigned to signed characters) which threw away two thirds of the encryption-key in many cases. As a result, 3/4 of all keys could be broken in less than 10 minutes [17]. This accident illustrates not only the severity of cipher bugs, but also the difficulty with cipher testing. The buggy BlowFish cipher actually worked correctly on all test cases, exemplifying a common issue with many ciphers, especially those based on Feistel rounds. These ciphers can encrypt and decrypt a file even when coded incorrectly, thus complicating correctness checking and verification.

In this paper we present StreamBit as a new programming methodology that simultaneously (i) ensures correctness and (ii) supports semi-automatic performance programming. We achieve these goals by allowing the user to guide the compilation process by *sketching* the desired implementation. To explain how sketching works, consider the following bit manipulation example we refer to as DropThird. It is concisely described as: "produce an output stream by dropping every third bit from the input stream". Somewhat surprisingly, this seemingly simple problem exhibits many issues arising in the more involved bit-stream manipulations algorithms, including parity checking and bit permutations.

To illustrate why this is a non-trivial example, consider a general-purpose processor with the usual suite of bitwise operations: *and*, *or*, *xor*, left/right *shift*—this is the target machine assumed in this paper. In order to obtain a high-performing implementation on such a machine, we must satisfy several trade-offs. For example, we can carefully prearrange the bits into words and therefore exploit more parallelism within a word. However, the cost of prearrangement may not be offset by word-level parallelism alone. In general, arriving at a good implementation requires employing locally sub-optimal implementation strategies. Further-

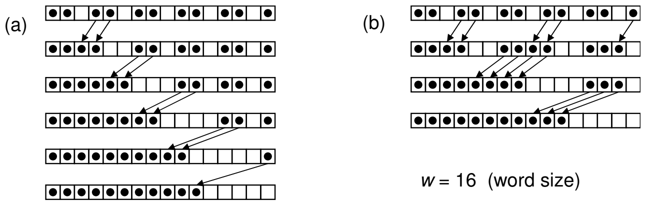


Figure 1. Two implementations of DropThird: (a) naive implementation with a $O(w)$ running time; and (b) log-shifter implementation with a $O(\log w)$ running time.

more, the exact way in which the prearrangement is implemented can lead to significant differences in performance. This is illustrated in Figure 1. The naive scheme shown on the left needs $O(w)$ steps, while the smarter scheme (on the right) requires $O(log(w))$ steps; $w = 16$ in our example and it represents the word size. The smarter algorithm, known to hardware circuit designers as a *log-shifter*, is faster because it utilizes the word parallelism better: it shifts more bits together while ensuring that no bit needs to be moved too many times—again, a tricky trade-off. Finding a better implementation is worth it: on a 32-bit Pentium processor, the log-shifter implementation is $1.6\times$ faster than the naive implementation; on a 64-bit Itanium architecture, it is $3.5\times$ faster; and on an IBM SP processor, it is $2.3\times$ faster. The *log-shifter* is also significantly faster than our table-lookup implementations, on all our platforms.

Besides finding an implementation algorithm with a suitable trade-off, the programmer faces several tedious and bug-prone tasks:

- *Low-level details.* Programming with low-level codes results in implementations that are typically disproportionate in size and complexity to the algorithm specification. For example, a FORTRAN implementation of a log-shifter (for a DropThird variant) requires 140 lines of code. The complexity is due to the use of different bit-masks at each stage of the algorithm.
- *Malleability and portability.* The low-level code is hard to modify, even for simple changes to the program specification (e.g., changing the specification to drop the second of every three bits). Furthermore, porting a 32-bit implementation to a 64-bit machine typically requires rewriting the entire implementation (not doing so will halve the performance of DropThird).
- *Instruction-level parallelism.* Micro-architectural characteristics may significantly influence performance. For example, the naive implementation in Figure 1 may win on some machines because it actually offers more instruction-level parallelism than the log-shifter (there are less data dependencies). To find a suitable implementation, the programmer typically implements and measures the performance of various implementation ideas.

When programming with StreamBit, the programmer first writes a full behavioral specification of the desired bit manipulation task. This specification, called the *reference program*, is written in the StreamIt dataflow language [19], and is nothing more than a clean, unoptimized program describing the task at the level of bits (rather than at the level of words). In case of ciphers, the reference program is usually a simple transcription of the cipher standard. In the absence of a Sketch, we compile the reference program to low-level C code with a *base compiler*. The base compiler exploits word-level parallelism using a simple greedy strategy based on local substitution rules.

Once the reference program is complete, either the programmer or a *performance expert* sketches an efficient implementation. *The goal of sketching is to provide only a loosely constrained template of the implementation, with the compiler filling in the missing details*. The details are obtained by ensuring that when the sketch is resolved, it implements the reference program; that is, the resolved sketch is behaviorally equivalent to the reference program.

A challenge with sketching the outline of an implementation is that the programmer may want to sketch implementation ideas at different levels of abstraction. StreamBit affords such a convenience by allowing the user-provided sketch to transform the program at any point through the base compilation process. Effectively, as the base compiler proceeds to transform the program to low-level C through local substitutions, the sketches will prompt additional transformations, which will result in an implementation that will approach the level of performance that the expert desires. The underlying intuition is that the local substitutions of the base compiler will perform a more optimal translation if the difficult global decisions are supplied via sketches—this is especially important when the complexity of an optimization is well beyond the scope of what a compiler can autonomously discover.

In Figure 2, we illustrate the key ideas behind sketching. On the very left is the reference program in visual form. This bit manipulation task is translated by the base compiler into a task that is equivalent, but also word-aligned. Now, if the base compiler is allowed to continue, it produces the slow implementation strategy shown in Figure 1(a). Instead, the programmer supplies a sketch of how the word-aligned task should be implemented. With sketching, the programmer states no more than a rough idea of the task: it is potentially possible to implement the DropThird permutation by first shifting some bits by one position, then some bits by two positions, and so on. The sketch does not specify which bits must shift in each step; in fact, it may not be possible to satisfy the sketch—log-shifting of course does not work for all permutations. If the sketch is satisfiable, then the details (i.e., which bits to shift) are derived automatically. Otherwise, it is rejected, and thus the user cannot introduce bugs into the implementation.

In summary, the StreamBit methodology offers the following benefits: (1) it obviates the need to code low-level details thereby making developers more productive, (2) it rejects buggy sketches and guarantees correctness by construction, (3) it allows the programmer to rapidly test and evaluate various implementation ideas, even before they are known to be correct—without the fear of introducing bugs. In addition, the sketches are robust because they are devoid of low-level details. For example, note that even though the implementation of the log-shifter is slightly different for different words, the same sketch applies to all of them because the general strategy remains the same; only the details of exactly which bits to shift change. In fact, if DropThird is modified to drop the second of every three bits, we simply need to change the reference program; the original sketch remains applicable. Thus, the sketch has a level of re-usability across a family of algorithms.

This paper makes the following contributions.

- We describe programming by sketching, a method for rapidly developing correct-by-construction implementations of bit-streaming programs.
- We develop a constraint solver for compiling sketch. The solver uses the reference program and the sketch to derive a complete (and optimized) implementation.
- We develop a sketching language that allows programmers to specify a large number of implementations.
- We report the results of a user study which showed that in the same amount of time, StreamBit programmers were more productive, producing code that ran $3\times$ faster than C programmers.
- We show that sketched implementations of DES and Serpent perform as well as (and in some cases outperform) heavily optimized and widely used implementations of these ciphers.

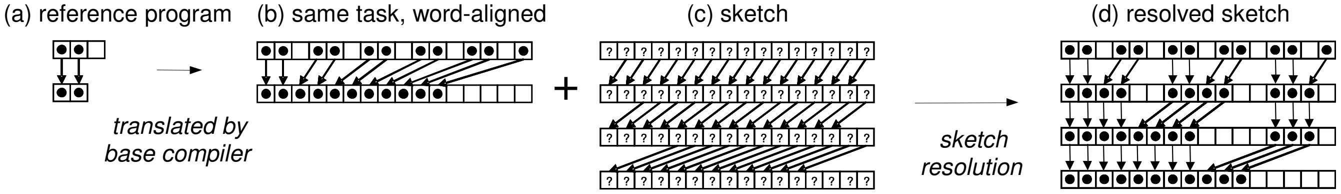


Figure 2. An illustration of programming with sketches on our running example. The compiler translates the reference program into machine code in several steps; at each step, it checks whether there is a sketch on how to implement the given program. In the figure, the compiler first unrolls the reference program to operate on words, rather than bits. Then, this word-aligned task (a full specification of a bit-stream manipulation function) is combined with the sketch (a partial specification of the implementation) to derive the full implementation. The compiler then continues to translate the resolved sketch into machine instructions.

The remainder of the paper is organized as follows. Section 2 describes the StreamIt programming language and model of computation. Sections 3 and 4 describe the program representation and base compiler infrastructure. Section 5 details the sketching methodology. Section 6 presents our evaluation and results. Lastly, a discussion of related work appears in Section 7, and our summary and concluding remarks appear in Section 8.

2. StreamIt

The StreamBit compiler is concerned with the domain of bit-streaming programs, with a focus on private-key ciphers. Such programs generally consume input bits from a stream (in blocks ranging in size from 32 to 256 bits), and produce output bits via a sequence of transformations on the stream. The transformations generally include permutations, substitutions, or arbitrary boolean functions. Additionally, in the case of ciphers, some functions will involve mixing an encryption key with the stream. An important characteristic of ciphers is that they tend to avoid data dependent computations, since this makes them vulnerable to side channel attacks; in particular, loop trip counts are fixed at compile time [8].

The StreamBit system uses StreamIt as the language for writing the reference programs. StreamIt [19] embraces a synchronous dataflow model of computation. In this model, a program consists of a collection of *filters* that produce and consume data through output and input ports respectively, and a set of FIFO communication channels that connect the output and input ports of different filters. A filter is an autonomous unit of computation that executes (or fires) whenever there is a sufficient amount of data queued on the input channel. In StreamIt, a filter is a single input-single output component with statically bound I/O rates—that is, the amount of data produced and consumed during a single filter firing is determined at compile time. The static I/O bounds allow the compiler to orchestrate the execution of the entire program, and to implicitly manage the buffers between filters. The compiler can thus lessen the burden on the programmer in terms of interleaving filter executions or allocating adequate buffer space between filters.

Figure 3 illustrates a simple filter that implements DropThird; this filter is our running example. The *work* function defines the output of the filter as a function of its input. The work function may contain arbitrary code, but sketching is only allowed when the loop bounds in the work function are not dependent on the input data, and when the filters do not maintain state (i.e., the output is strictly a function of the input). These requirements, along with the requirement of statically defined input and output rates, are not an impediment in our domain, since ciphers tend to avoid data dependent computation, and are generally defined as a sequence of functions that do not resort to any saved state.

```
bit->bit filter DropThird {
  work push 2 pop 3 {
    for (int i=0; i<3; ++i) {
      bit x = peek(i)
      if (i<2) push(x);
      pop();
    }
  }
}
```

Figure 3. The reference program of DropThird, our running example, expressed here as a StreamIt filter that drops every third bit.

In StreamIt, the communication channels are defined implicitly by composing filters into *pipelines* or *splitjoins*. A pipeline is a sequence of filters that are linked together such that the output of filter i is the input of filter $i+1$. The StreamIt program below illustrates a pipeline that consumes an input stream and drops every third bit from it, and subsequently drops every third bit from the resulting stream.

```
bit->bit pipeline DropTwice {
  add DropThird();
  add DropThird();
}
```

A splitjoin consists of a *splitter* which distributes its input stream to N children filters, and a *joiner* which merges the output of all N children. There are two types of splitters: *duplicate* splitters and *roundrobin* splitters. A duplicate splitter passes identical copies of its input to every one of its children. In contrast, a roundrobin splitter distributes the input stream to the individual filters in a roundrobin fashion, according to the specified rate for each filter. Similarly, a roundrobin joiner reads data in a roundrobin fashion from every branch in the splitjoin and writes them in order to the output stream (such a joiner effectively concatenates its inputs). In the splitjoin example below, every three consecutive bits x, y, z in the input stream are processed as follows: the duplicate splitter sends identical copies of these three bits to the two children filters. The roundrobin joiner reads the two bits x, y from the output of the *DropThird* filter, and one bit z from the output of the *KeepThird* filter, and concatenates them, producing x, y, z again in the output stream.

```
bit->bit splitjoin FunnyIdentity {
  split duplicate;
  add DropThird();
  add KeepThird();
  join roundrobin(2,1);
}
```

In StreamIt, filters, pipelines, and splitjoins can be nested, leading to a hierarchical composition of a stream program. The language exposes the communication between filters and the task level parallelism in the program, while also affording modularity, malleability, and portability. It is therefore well suited for our purpose, and is also rich and expressive enough to serve as an intermediate representation throughout the compilation process.

3. Program Representation

The synchronous dataflow programming model of the StreamIt language serves as a convenient foundation for the intermediate representation used throughout the StreamBit compilation process. We use an abstract syntax tree (AST) to represent the three possible stream constructs: leaves represent filters, and internal nodes represent pipelines and splitjoins.

3.1 Filters

In general, a filter can represent an arbitrary mapping between its input bit vector and its output bit vector. However, there is class of filters whose output is an *affine* function of their input. Such filters are important because they have useful algebraic properties, and can serve as building blocks to represent *permutations* and arbitrary *boolean functions*, and for this reason they will constitute the leaves of the AST.

Affine Filters Formally, an affine filter f is a filter whose work function performs the following transformation:

$$[\![f]\!](x) = Mx + v$$

where x is a n-element boolean (i.e., binary) vector, M is a $m \times n$ boolean matrix, and v is a m-element "offset" boolean vector. Note that all vectors in this paper are column vectors but are sometimes written as row vectors for compactness. We denote an affine filter with $f = (M, v)$, and the work function with $[\![f]\!](x)$. If the offset vector is the zero vector (i.e., $v = [0, \ldots, 0]$), then we simply write $f = M$.

We distinguish three types of affine filters: `and`, `or`, and `xor`. These filters differ in the boolean operations used to carry out the scalar multiplication and scalar addition that make up the vector operations. The three types of filters are summarized as follows. The type of a filter f is obtained via *typeof* (f).

typeof (f)	scalar multiplication	scalar addition
`and`	$\vee$	$\wedge$
`or`	$\wedge$	$\vee$
`xor`	$\wedge$	$\oplus$

For the rest of the paper, unless otherwise noted, we assume filters are of type `xor`.

Permutation Filters In cipher implementations, permutations are a particularly important class of affine filters. In this paper, a permutation is a filter that optionally removes some bits from the n-bit input vector, and arbitrarily reorders the remaining input bits to obtain an m-bit result. In StreamBit, as a matter of convention, a permutation filter is of type `xor`, but the type is not important because permutation filters have a zero offset vector v, and a work matrix M with at most one non-zero value per row, and hence no addition is involved [1].

As an example, the work matrix for the DropThird permutation filter is:

$$M_{\textsf{DropThird}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

A natural way to interpret the matrices is to realize that each row corresponds to single output bit, and that the non-zero element in the row represents which of the input bits (if any) will appear in the output.

Arbitrary Boolean Functions Filters that implement arbitrary boolean functions can be transformed to pipelines of affine filters. This is possible because any boolean function is expressible in disjunctive normal form, i.e., as a disjunction of minterms. In StreamBit, a pipeline of three affine filters implements functions in disjunctive normal form as follows: first, an `xor` filter negates input bits; second, an `and` filter computes the minterms; and finally an `or` filter performs the disjunction. The StreamBit compiler uses a more complicated algorithm based on symbolic execution—which is not exponential, unlike minterm expansion—to produce a pipeline of `and`, `or` and `xor` filters for work functions with arbitrary boolean expressions.

3.2 Pipelines and Splitjoins

In our intermediate representation, the internal nodes of the AST are pipelines and splitjoins. A pipeline AST node with children x_1 to x_n is denoted $PL(x_1, \ldots, x_n)$. The children may in turn represent filters, pipelines, or splitjoins (since StreamIt allows for a hierarchical composition of stream components). For example a pipeline of n filters is represented as

$$PL((M_1, v_1), ..., (M_n, v_n)),$$

whereas a pipeline of n pipelines is represented as

$$PL(PL_1(...), ..., PL_n(...)).$$

In both cases however, the communication topology is clear: the output of child x_i is the input of child $x_i + 1$.

A splitjoin AST node connects its children x_1 to x_n to a splitter *sp* and a joiner *jn*. A splitjoin is denoted $SJ_{sp,jn}(x_1, \ldots, x_n)$, with duplicate splitters denoted *dup*, and roundrobin splitters (or joiners) with rates $k_1, \ldots, k_n$ denoted $rr(k_1, \ldots, k_n)$. For example, the splitjoin for `FunnyIdentity` in section 2 is described as:

$$SJ_{dup,rr(2,1)}(M_{\textsf{DropThird}}, M_{\textsf{KeepThird}}).$$

When the rates of a roundrobin splitter or joiner are equal ($k_1 = ... = k_n$), we simply write $rr(k_1)$. We will use the notation $rr[i]$ to refer to the i^{th} rate in the round-robin. In StreamBit, we also support a *reduction* joiner which is denoted *red*. There are three types of reducing joiners—namely `and`, `or`, and `xor`—each of which applies the corresponding boolean operator to n bits from the input stream (one bit from each of n input streams) and commits the result to the output stream. In other words, a reduction joiner is semantically equivalent to a $rr(1)$ joiner whose output is consumed by an affine filter $f = M$ where $typeof(f)$ corresponds to the type of the reduction joiner, and every entry of the binary work matrix M is non-zero.

4. The Base Compiler

This section describes the base compilation algorithm that is used by the StreamBit system to produce code from the reference program. The base algorithm is very fast and predictable, but the code it produces is not optimal. The next section will address this by explaining how StreamBit builds on top of this algorithm to allow for sketching of user-defined implementations.

The base compilation works by applying local substitution rules on the AST to convert the reference program into an equivalent StreamIt program with the property that each of its affine filters corresponds to an *atomic operation* in our machine model. A program with these characteristics is said to be in *Low-Level Form (LL-form)*. More formally, an LL-form program is defined as a program with the following two properties:

(1) Its communication network (AST internal nodes) transfers data in blocks of a size that can be handled directly by the atomic operations in the model machine.

[1] The only caveat to this statement is that if we want to represent a permutation with an `and` filter, we need to negate all the entries in the matrix and offset vector, i.e. we must have one *zero* value per row, and the offset vector must be all ones.

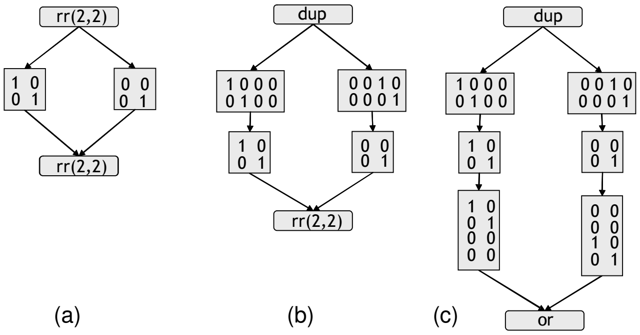


Figure 4. The Splitjoin Resizing transformations. The original filter (a) is transformed using Split Equivalence (b); the result is then transformed using Join Equivalence (c).

(2) Its filters (AST leaves) correspond to a single atomic operation.

Throughout this paper, as well as in our current implementation, we use as our machine model a processor that can move data in blocks of size w (the word-size), and apply any of the following atomic operations on it: right and left logical shift on words of size w, *and*, *or* and *xor* of two words of size w, and *and*, *or* and *xor* of one word of size w with a fixed constant.

Programs in LL-form are easily mapped to C code, because their dataflow graphs consist of atomic operations linked together through communication channels that define how values flow among them. Our choice of machine model makes translation into C easy, because all the atomic operations can be expressed in a single C statement. We chose C as the target of our compilation, both for portability reasons, and to take advantage of the register allocation and code scheduling performed by good C compilers, which allows us to focus on higher level optimizations.

The base compilation algorithm will transform any program to satisfy the two properties of LL-form by following three simple steps, all based on local substitution rules:

i) Splitjoin Resizing;
ii) Leaf Size Adjustment;
iii) Instruction Decomposition.

Figure 6 shows the complete algorithm. Before describing each of the stages in detail, we point out that the first two stages will be responsible for enforcing property (1), while the third stage will be responsible for property (2).

Splitjoin Resizing This stage will apply local rewrite rules to the internal nodes of the AST when it is necessary in order to remove a violation of property (1). This, it turns out, will only be necessary in the case of splitjoins with roundrobin splitters or joiners where there is an i for which $rr[i]$, the number of bits to deliver, is not a multiple of w (w equals the number of bits our assumed machine can handle in one instruction). In this case, we apply a transformation based on the Join and Split Equivalences of Figure 5 to transform the splitter/joiner to one that makes no mention of rates, i.e., a duplicate spliter or a reduction joiner. Figure 4 illustrates the effect of the transformations on this stage for a simple splitjoin. As it will become apparent shortly, pipelines and other types of splitjoins do not need to make substitutions to the internal nodes of the AST to satisfy property (1), and therefore are not affected by this stage. In particular, this transformation is not needed when compiling DropThird because DropThird has no splitjoins.

Leaf Size Adjustment Once all splitjoins are transformed via Splitjoin Resizing, we complete the transformation to LL-form using only local rewrite rules on the leaves of the AST.

We can do this because in the case of pipelines and splitjoins with *dup* splitters or *red* joiners, property (1) is trivially maintained as long as all their children produce and consume bits in multiples of the w. Therefore, a simple inductive argument shows that we don't need to make any more changes to the internal nodes to satisfy property (1); we only need to guarantee that the leaf nodes produce and consume bits in multiples of w.

The Leaf Size Adjustment stage will enforce property (1) by forcing each leaf node to consume *exactly* w bits and produce w bits through the application of the UNROLL, COLSPLIT, and ROWSPLIT substitution rules as shown in the pseudocode in Figure 6.

To understand how the LEAF-SIZE-ADJ procedure works, consider the running example as illustrated in Figure 7. The figure shows the complete sequence transformations we would perform on the DropThird filter to compile it into instructions for a 4 bit wide machine.

First, in step (a), we use the unroll equivalence from Figure 5 to substitute DropThird with an equivalent filter whose size is a multiple of the word size.

$$M_u := \begin{bmatrix} M_{\mathsf{DropThird}}\ 0\ 0\ 0 \\ 0\ M_{\mathsf{DropThird}}\ 0\ 0 \\ 0\ 0\ M_{\mathsf{DropThird}}\ 0 \\ 0\ 0\ 0\ M_{\mathsf{DropThird}} \end{bmatrix} = M_{\mathsf{DropThird}}$$

Then, in step (b), we use the column equivalence to replace the affine filter with an equivalent splitjoin of filters whose input size is equal to the word size

$$SJ_{rr,or}(M_1, M_2, M_3) = M_u$$

where the M_i are defined such that

$$M_u = [M_1, M_2, M_3]$$

Finally, in step (c) we use the row equivalence to substitute each of the new affine filters with a splitjoin of filters whose input and output size is equal to w.

$$\begin{aligned} &SJ_{rr,or}(\\ &\quad SJ_{dup,rr}(M_{1,1}, M_{1,2}), \\ &\quad SJ_{dup,rr}(M_{2,1}, M_{2,2}), \\ &\quad SJ_{dup,rr}(M_{3,1}, M_{3,2}) \\ &) = SJ_{rr,or}(M_1, M_2, M_3) \end{aligned}$$

Where

$$M_i = \begin{bmatrix} M_{i,1} \\ M_{i,2} \end{bmatrix}$$

Note that all we have done is apply local substitution rules to the leaves until all the leaf filters take w bits of input and produce w bits of output, and as a consequence, property (1) is fully satisfied. Also note that the validity of the substitutions can be ascertained by looking only at a single node and its immediate children in the AST, so the process is very fast.

Instruction Decomposition Once all the leaf nodes in the AST map one word of input to one word of output, we need to decompose them into filters corresponding to the atomic bit operations available in our assumed machine model.

The pseudocode in Figure 6 shows the Instruction Decomposition as simply applying to every leaf-node one last substitution called IDECOMP.

The IDECOMP transformation is based on the observation that any matrix can be decomposed into a sum of its diagonals. Each diagonal in turn can be expressed as a product of a matrix corresponding to a bit-mask with a matrix corresponding to a bit shift. Thus, any matrix m of size $w \times w$ can be expressed as $m = \sum_{i=-(w-1)}^{w-1} a_i s^i$, where the a_i are diagonal matrices, and s^i is a matrix corresponding to a shift by i. Given this decomposition, we use the sum and product equivalences from Figure 5 to

Unroll Equivalence

UNROLL[n]$((M, v))$
Where **n** is the number of times to unroll.

$$(M, V) \rightarrow (\begin{bmatrix} M & 0 & \dots \\ 0 & M & 0\dots \\ \dots & \dots & \dots \\ 0 & \dots & M \end{bmatrix}, \begin{bmatrix} v \\ v \\ \dots \\ v \end{bmatrix})$$

Column Equivalence

COLSPLIT[$s_1, \dots, s_n$**]**$((M, v))$
Where the s_i define the partition of matrix M into the M_i.

$$([M_1, M_2, \dots, M_n], \sum_{i=1}^{n} v_i) \rightarrow SJ_{rr,red}((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n))$$

COLMERGE(SJ)
Precondition:
$rr[i] = insize(M_i)$ and
$typeof(M_i) = typeof(red) \quad \forall i$

$$SJ_{rr,red}((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n)) \rightarrow ([M_1, M_2, \dots, M_n], \sum_{i=1}^{n} v_i)$$

Row Equivalence

ROWSPLIT[$s_1, \dots, s_n$**]**$((M, v))$
Where the s_i define the partition of matrix M into the M_i.

$$(\begin{bmatrix} M_1 \\ M_2 \\ \dots \\ M_n \end{bmatrix}, \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix}) \rightarrow SJ_{dup,rr}((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n))$$

ROWMERGE(SJ)
Precondition:
$rr[i] = outsize(M_i)$ and
$typeof(M_i) = typeof(M_j) \quad \forall i, j$

$$SJ_{dup,rr}((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n)) \rightarrow (\begin{bmatrix} M_1 \\ M_2 \\ \dots \\ M_n \end{bmatrix}, \begin{bmatrix} v_1 \\ v_2 \\ \dots \\ v_n \end{bmatrix})$$

Product Equivalence

PCOLLAPSE(PL)
Precondition:
$typeof(M_i) = typeof(M_j) \quad \forall i, j$

$PL((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n)) \rightarrow$
. $(M_n * M_{n-1} \dots * M_1, \sum_{i=1}^{n} (M_n * M_{n-1} \dots * M_{i+1}) v_i)$

PEXPAND[$M_1, \dots, M_n$**]**$((M, v))$
Where $M_1, \dots, M_n$ define the partition of the M_i.

$(M_n * M_{n-1} \dots * M_1, \sum_{i=1}^{n} (M_n * M_{n-1} \dots * M_{i+1}) v_i) \rightarrow$
. $PL((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n))$

Sum Equivalence

SCOLLAPSE(SJ)
Precondition:
$typeof(M_i) = typeof(M_j) \quad \forall i, j$

$$SJ_{dup,red}((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n)) \rightarrow (\sum_{i=1}^{i=n} M_i, \sum_{i=1}^{n} v_i)$$

SEXPAND[$M_1, \dots, M_n$**]**$((M, v))$
Where $M_1, \dots, M_n$ define the partition of the M_i.

$$(\sum_{i=1}^{i=n} M_i, \sum_{i=1}^{n} v_i) \rightarrow SJ_{dup,red}((M_1, v_1), (M_2, v_2), \dots, (M_n, v_n))$$

Split Equivalence

SJTODUP(SJ)
See Figure 4 for an example of this transformation.

$$SJ_{rr,any}(F_1, F_2, \dots, F_n) \rightarrow SJ_{dup,any}(PL(S_1, F_1), PL(S_2, F_2), \dots, PL(S_3, F_n))$$

where S_i is an $M \times N$ matrix where $M = rr[i]$ and $N = \sum_{l=1}^{n} rr[l]$
and the $S_i[j, k] = 1$ when $k - j = \sum_{l=1}^{i-1} rr[l]$ and $0 \leq j \leq M$, and is zero otherwise.

Join Equivalence

SJTOXOR(SJ)
See Figure 4 for an example of this transformation.

$$SJ_{any,rr}(F_1, F_2, \dots, F_n) \rightarrow SJ_{any,red}(PL(F_1, S_1), PL(F_2, S_2), \dots, PL(F_n, S_3))$$

where S_i is an $M \times N$ matrix with$M = \sum_{l=1}^{n} rr[l]$ and $N = rr[i]$
and the $S_1[j, k] = 1$ when $j - k = \sum_{l=1}^{i-1} rr[l]$ and $0 \leq k \leq N$, and is zero otherwise.

Figure 5. Rewrite rules on our StreamIt AST.

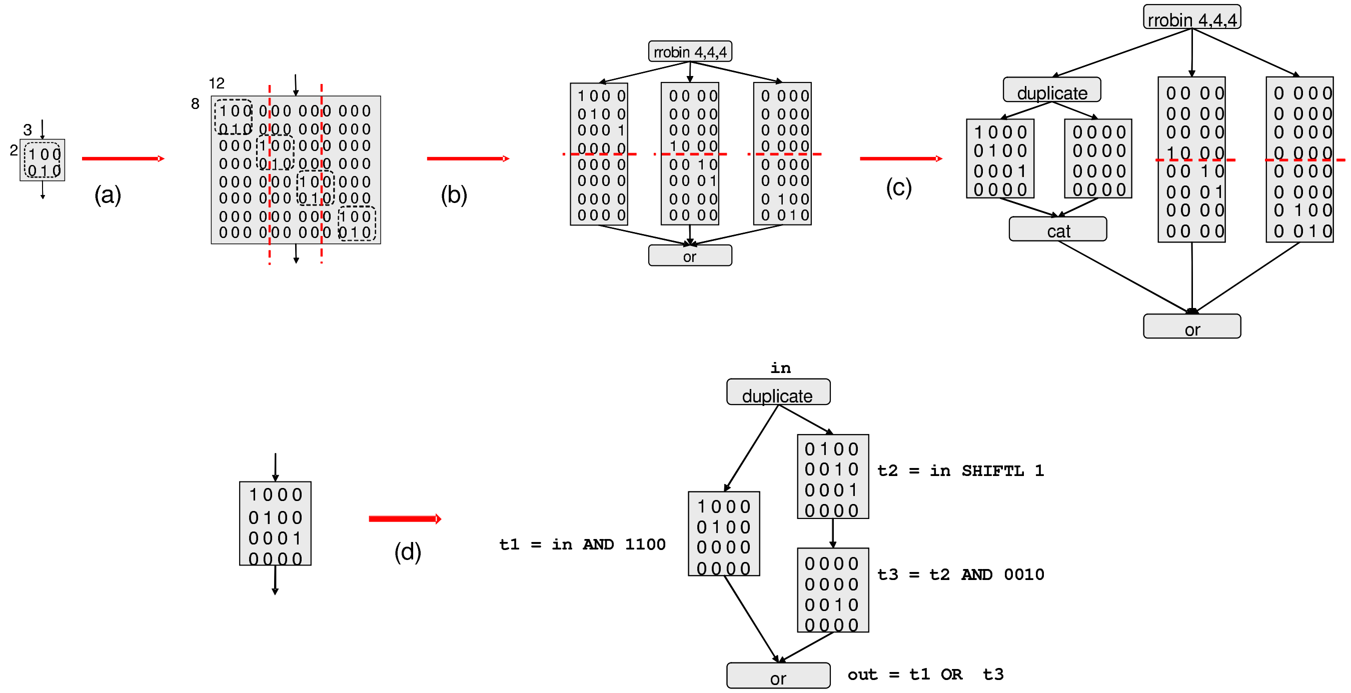
```
BASE-COMPILATION($program : AST$)
  $program \leftarrow$ SJ-RESIZE($program$)
  $program \leftarrow$ LEAF-SIZE-ADJ($program$)
  $program \leftarrow$ INSTRUCTION-DECOMPOSITION($program$)
  return $program$

SJ-RESIZE($program : AST$)
  while $\exists$ node $n \in program$ s.t $typeof(n) =$ SJ and $(split(n) =$ RR or $join(n) =$ RR$)$ and $\exists\, i$ s.t. $w \nmid rr[i]$
  ▷ $w \nmid rr[i]$ means $w$ doesn't divide the $i^{th}$ entry of the round-robin spliter/joiner
      do APPLY-SKETCH($program$)
         if $split(n) =$ RR and $\exists\, i$ s.t. $w \nmid rr[i]$ then $program \leftarrow program[n \backslash$SJTODUP$(n)]$; continue
         if $join(n) =$ RR and $\exists\, i$ s.t. $w \nmid rr[i]$ then $program \leftarrow program[n \backslash$SJTOXOR$(n)]$; continue
  return $program$

LEAF-SIZE-ADJ($program : AST$)
  while $\exists$ leaf-node $n \in program$ s.t $\neg(w =$ INSIZE$(n) =$ OUTSIZE$(n))$
      do APPLY-SKETCH($program$)
         if $w \nmid$ INSIZE$(n)$ then $program \leftarrow program[n \backslash$UNROLL$[w/gcd(w, insize(n))](n)]$; continue
         if $w \nmid$ OUTSIZE$(n)$ then $program \leftarrow program[n \backslash$UNROLL$[w/gcd(w, outsize(n))](n)]$ continue
         if insize$(n) \neq w$ then $program \leftarrow program[n \backslash$COLSPLIT$[w, \ldots, w](n)]$ continue
         if outsize$(n) \neq w$ then $program \leftarrow program[n \backslash$ROWSPLIT$[w, \ldots, w](n)]$ continue
  return $program$

INSTRUCTION-DECOMPOSITION($program : AST$)
  for each leaf-node $n \in program$
      do APPLY-SKETCH($program$)
         ▷ The IDECOMP transformation is defined in section 4
         $program \leftarrow program[n \backslash$IDECOMP$(n)]$
  return $program$
```

Figure 6. Pseudocode for the base compilation algorithm.



```
t1 = in AND 1100
t2 = in SHIFTL 1
t3 = t2 AND 0010
out = t1 OR  t3
```

Figure 7. An illustration of the base compilation algorithm for our running example, assuming a 4 bit machine: (a) involves unrolling the filter to take in and push out a multiple of w, the word size; (b) shows how a filter is broken into columns with a semantics preserving transformation; (c) shows the effect of a similar transformation for breaking into rows; (d) shows one of the filters from (c) as it is finally converted into LL-form filters. Steps (a) through (c) correspond to size adjustment. Step (d) corresponds to instruction decomposition.

define IDECOMP as

$$(M, v) = SJ_{dup,red}(PL(s_1, (a_1, v)), \ldots, PL(s_n, a_n))$$

As an example, to continue with our DropThird running example, we can take matrix $M_{1,1}$ and note that

$$\begin{aligned} M_{1,1} \quad &= Diag[1100] * s_0 + Diag[0010] * s_1 \\ &= PL(s_0, Diag[1100]) + PL(s_1, Diag[0010]) \\ &= SJ_{dup,or}(PL(s^0, Diag[1100]), PL(s^1, Diag[0010])). \end{aligned}$$

In this case, the first equality is simply a statement about matrices. The second one is an application of PEXPAND, and the third equality is an application of SEXPAND, and the composition of the two gives us IDECOMP.

This transformation is illustrated in Figure 7(d). Note that in the figure, $PL(s^0, Diag[1100])$ has been replaced by $Diag[1100]$ since s^0 corresponds to the identity matrix.

In the remainder of this section, we digress a little to show that this approach is actually quite general, and can allow us to build IDECOMP functions to produce code for more complicated machine models. For example, adding a rotation instruction to our machine model poses no major problem, since for positive i, $s_i = a_i^+ * r^{|i|}$, and for negative i, $s_i = a_i^- * r^{|i|}$, where a_i^+ and a_i^- are diagonal matrices that mask either the upper i bits or the lower i bits. Thus, we only need to replace s_i in the original expression for m, and collect terms to get a decomposition $m = \sum_{i=0}^{w-1} a_i r^i$ in terms of rotations r^i. With this new equation, the IDECOMP function can be produced trivially as a composition of SEXPAND and PEXPAND, just as with the original machine model.

In fact, the same strategy works even for more complicated shifts, like those found in some SIMD extensions, which have shift instructions to shift entire bytes at a time $t^i = s^{8*i}$, as well as packed byte shifts u^i which correspond to shifting several contiguous bytes independently but in parallel, with bits that move past the boundary of a byte being lost. In that case, $s^i = t^{\lfloor i/8 \rfloor} * u^{i \bmod 8} + t^{\lfloor i/8 \rfloor + 1} * u^{(i \bmod 8) - 8}$, and just like before, we can simply replace this expression for the s_i in the original formula and get a decomposition in terms of sequences of shifts of the form $t^i * u^j$.

The base algorithm has the advantage that because the transformation is based on local substitution rules, its results are easy to predict. In particular, the final number of operations is roughly proportional to the number of diagonals in the original matrix. It will generally produce suboptimal results because bits that shift by the same amount are shifted together all the way to their final position in a single operation. For example, in the case of DropThird, this strategy produces the naive implementation from Figure 1(a). In order to get the better implementation, we'll need to use sketching.

5. Sketching

We are now ready to describe the *sketching compiler*, which synthesizes an implementation that is functionally identical to the reference program while being structurally conformant to the sketched implementation.

We will explain synthesis with sketches in two steps. We will first assume that the sketch provided by the programmer is *complete*, i.e., that it does not omit any implementation detail. We will then explain how proper sketches, those which do omit details, can be *resolved* to complete sketches. In our running example, the dataflow program in Figure 2(d) is a complete sketch, while the program in Figure 2(c) is a proper sketch.[2]

The benefit of restricting ourselves initially to complete sketches is that we can focus on how to synthesize an implementation that conforms to a sketch, without distracting ourselves with how sketches are resolved. Sketch resolution, which synthesizes the details missing in a sketch, will be the focus of the second step.

Our sketching constructs provide support for several important implementation patterns: implementation of an affine filter, and in particular of permutations, as a sequence of steps; restructuring of pipelines and splitjoins; and implementation of filters with table lookups. We explain sketching on the problem of permutation decomposition. Section 5.1 then explains how we efficiently resolve sketches by means of combining search with constraint solving. In Sections 5.2–4, we discuss sketching for the remaining implementation patterns.

Complete sketches To motivate the definition of the sketch, it helps to recall that we view the synthesis of a bit-streaming implementation as a process of decomposing a dataflow program into Low-Level form; the base compiler performs one such decomposition. It is thus natural to view the sketch as a constraint on the shape of the decomposed program. We allow sketches to impose constraints at an arbitrary stage in the decomposition: constraining an early stage has the effect of sketching high-level steps of the implementation algorithm (e.g., that we want to pack bits within words first), while constraining later stages sketches finer details (e.g., how to manipulate bits within a word).

A key research question is what form these constraints on the decomposition should take so that the sketch is both concise and natural to express. To obtain conciseness, we rely on the base compiler to perform most of the decomposition. The base compiler performs a fixed decomposition sequence, which permits the programmer to anticipate the dataflow programs created throughout the decomposition, which in turn enables him to sketch these programs. The base compiler leads to conciseness because the programmer will control the compiler (with a sketch) only when its base algorithm would make a poor decoposition transformation.

The question now is how to override the base compiler with a sketch while making it natural for the programmer to express the sketch. Our solution is to express sketches as rewrite rules; these rewrite rules will extend the set of rules employed by the base compiler.

It may seem awkward to sketch the desired implementation with a rewrite rule, but to the programmer a sketch looks just like a program. Specifically, a complete sketch is a StreamIt program P that implements some permutation f. This program serves as a sketch in the following way: when the decomposition encounters a leaf filter with a permutation f, the filter is not decomposed using the base rewrite rules; instead, it is replaced with the program P. In effect, the leaf filter is rewritten with the AST of the program P. A compiler that consults the sketches in such a way is called *sketching compiler*.

As an example, consider the complete sketch in Figure 2(d). The StreamIt version of this complete sketch is below. The filter `Stage1` implements the first stage of the pipeline in the figure; the other two stages are analogous. To understand the code, recall that the first `pop()` reads the left-most bit in the pictured word. If the code looks too complicated, note that the programmer actually writes the simpler proper sketch shown below.

```
bit->bit pipeline LogShifter {
  add Stage1();
  add Stage2();
  add Stage3();
}
```

[2] Note that in Figure 2, the complete sketch (d) has been obtained automatically, by resolving the proper sketch (c), but the programmer is nonetheless free to develop a complete sketch manually. This is still easier than developing the complete implementation, and is beneficial when we desire an implementation that sketching does not support, e.g., when one wants to decompose a general affine filter in a way not expressible in our domain algebra.

```
bit->bit filter Stage1 {
  work push 16 pop 16 {
    // lines below either copy or drop a bit
    // or shift a group of bits by one position
    push(pop()); push(pop());
    pop(); push(pop()); push(pop()); push(0);
    push(pop()); push(pop()); push(pop());
    pop(); push(pop()); push(pop()); push(0);
    push(pop()); push(pop()); push(pop());
    pop(); push(pop()); push(0);
  }
}
...
```

Although this StreamIt program has structurally three pipeline stages, it implements the one-stage permutation shown in Figure 2(b); its equivalence is provable using the Product Equivalence rules in Figure 5. Figure 2 also shows how the sketching compiler uses the sketch: the reference program shown in Figure 2(a) is decomposed using base rules until one of the AST leaves is a filter with the function in Figure 2(b). At this point, this filter node is rewritten into the one shown in Figure 2(d). After the sketch is applied, the compiler will continue its base decomposition, breaking down each stage of the pipeline into machine instructions using the rule IDECOMP defined in Section 4. Note that in order take full advantage of the log-shifter, the implementation of DropThird should pack bits within a word before packing them across words. This is also expressed with a sketch, which we omit for lack of space.

Figure 6 shows that to extend the base compiler into a sketching compiler, it suffices to perform all applicable sketches before a base rewrite rule is applied; this is done in the function APPLY-SKETCH. A careful reader has by now observed that malformed sketches may prevent termination of the sketching compiler. Consider a sketch that decomposes a filter f into a pipeline $PL(f, identity)$; such a sketch can be applied indefinitely. Our current solution is to specify the position in the AST where each sketch should be applied. In the future, we plan to analyze the sketches.

Proper Sketches Informally, proper sketches differ from complete ones in that they omit some implementation details. Formally, we define proper sketches as non-deterministic StreamIt programs with the choice operator '*'. A non-deterministic program may compute one of several functions, potentially a different one for each of its executions. The process of resolving a sketch thus amounts to selecting the execution that computes the desired function f (if such an execution exists). We call this process *determinization to function* f. In other words, the deterministic portion of the sketch is what the implementation must adhere to; the rest is synthesized.

To make it work with proper sketches, we generalize the sketching compiler slightly: when the compiler is about to decompose a filter with a permutation f, it looks not for a complete sketch that implements f but instead for a proper sketch whose set of non-deterministic executions includes f; if such a sketch exists, it is determinized to f and applied as if it was a complete sketch.

As an example, consider the proper sketch in Figure 2(c). The non-deterministic StreamIt program equivalent to this sketch is a pipeline of three filters as follows.

```
bit->bit filter SketchedStage1 {
  work push 16 pop 16 {
    while(*) {
      switch (*) {
      case CopyOneBit:  push(pop()); break;
      case ShiftBits:
         pop(); while (*) push(pop()); push(0);
      }
    }
  }
}
```

Note that the complete sketch we gave above corresponds to one possible execution of this non-deterministic program. The set of all possible executions implement a generic version of the log-shifter that shifts, in the first stage, an unspecified subset of bits to the left by one position and copies the remaining bits; the remaining two stages are analogous.

The proper sketch is already more concise than the complete sketch, but the shifting pattern expressed in the non-deterministic program is so common that we developed a small sketching language to express it even more concisely. The proper sketch in this language is shown below. The meaning of the sketch is that a permutation filter is to be decomposed into a pipeline of three filters, where the first pipeline shifts a subset of bits from the 1:16 range by either zero bits or by one bit; similarly for the other pipeline stages. The language allows us to express only a restricted set of non-deterministic patterns, so it is not as expressive as the full non-deterministic StreamIt would be, but this is what will allow us to control the combinatorial explosion in the search space.

```
SketchDecomp[
  [shift(1:16 by 0 || 1)], // SketchedStage1
  [shift(1:16 by 0 || 2)], // SketchedStage2
  [shift(1:16 by 0 || 4)]  // SketchedStage3
];
```

Recall that resolving a sketch amounts to finding an execution of the non-deterministic program that implements the desired function. A straightforward way to resolve a sketch is a brute-force search over all possible executions, but there are 2^{3*16} of them for the non-deterministic pattern for the 16-bit log-shifter given above (2^{5*64} for a 64-bit version). To make the search feasible, we need to take advantage of the algebraic structure of the permutations to reduce the search space to a manageable size. To do this, we support a restricted class of sketches; in particular, we support sketches expressed in the sketching language described in the next subsection.

5.1 Sketching Decompositions of Permutations

We present next the algorithm for resolving sketches that decompose a permutation into a pipeline of (simpler) permutations. Our approach is to encode the desired pipeline as a vector of distances traveled by bits in a pipeline stage; these distances are our unknowns. We will express the sketch using two kinds of constraints on the decomposition vector: linear and non-linear; the former are solvable with linear algebra, the latter are not. We will first solve the linear constraints, leaving us with a reduced linear space of possible solutions. We then search for solutions in this space that also satisfy the non-linear constraints. The search is exhaustive, but we take advantage of the linear properties of the space to reduce it as much as possible. If the search space remains too large for a practical search, the user is asked to add more details to the sketch to further constrain the space (this was not necessary in our experiments).

Suppose a permutation consumes and produces a vector of bits numbered 1 to N. We express the permutation as a vector

$$\langle x_1, x_2, \ldots, x_N \rangle \text{ with } x_i = p_i^d - p_i^s$$

where p_i^d is the final position of bit i and p_i^s is the initial position of bit i. Since bits are labeled by their initial positions, we have $p_i^s = i$. A decomposition of this permutation into the desired k-stage pipeline defines the *decomposition vector* $\vec{Y}$:

$$\vec{Y} = \langle y_1^1, y_2^1, \ldots, y_N^1, y_1^2, y_2^2, \ldots, y_N^2, \ldots, y_1^k, y_2^k, \ldots, y_N^k \rangle$$

with $y_i^j = p_i^j - p_i^{j-1}$, where p_i^j is the position of bit i at the end of pipeline stage j; $p_i^0 = p_i^s$ and $p_i^k = p_i^d$. Once the constraints are generated, we want to solve for y_i^j.

The next step is to translate the sketch into constraints over the decomposition vector $\vec{Y}$. In our sketching language, a sketch can be built from four kinds of constraints. For each stage of the sketched pipeline, the programmer can specify any combination of the four types of constructs. For each sketching construct, we give below its translation into constraints over $\vec{Y}$.

1. `shift(`$b_1, \ldots, b_M$ `by j)`. Shifts by j positions all bits in the set $\{b_1, \ldots, b_M\}$. The set can be expressed as a range `a:b`. The bits are identified by their positions before stage 0 of the pipeline. Generated constraints: $y^k_{b_i} = j$, where k is the stage where the constraint appears and $i \in \{1, \ldots, M\}$.
2. `shift(`$b_1, \ldots, b_M$ `by ?)`. Shifts all bits in $\{b_1, \ldots, b_M\}$ by the same amount h, where h is unspecified in the sketch. Generated constraints: $y^k_{b_i} = y^k_{b_{i+1}}$, where $i \in \{1, \ldots, M-1\}$.
3. `pos(`b_i`, p)`. Requires that bit b_i will be in position p after stage k, i.e., $p^k_{b_i} = p$. Since $y^j_i = p^j_i - p^{j-1}_i$, this constraint translates to
$$p^0_{b_i} + \sum_{j=1}^{k} y^j_{b_i} = p.$$
4. `shift(`$b_1, \ldots, b_M$ `by a || b)`. Shifts each bit in $\{b_1, \ldots, b_M\}$ by either a or b positions. (Note that bits do not need to all move by the same amount.) Generated constraints: $y^k_{b_i} \in \{a, b\}$. This is a non-linear constraint.

To ensure that the decomposition is semantics preserving, we add two other constraints:

5. The final position of each bit must agree with the final position of the bit in the permutation being decomposed. This constraint is a special case of constraint (3), and is handled the same way.
6. No two bits can reside in the same position at the end of any stage (otherwise, they would overwrite each other). This constraint is non-linear.

The linear constraints will lead to a matrix equation of the form $S * \vec{Y} = \vec{T}$, where S is matrix and $\vec{T}$ is a vector representing generated linear constraints. This equation is solved in polynomial time using Gaussian elimination over the integers. The result will be a particular solution $\vec{Z}$ and a set of decomposition vectors of $\vec{V}_1, \ldots, \vec{V}_m$ such that the decomposition vector $\vec{Y}$ can be obtained as a linear combination of $\vec{Z}$ and $\vec{V}_i$:

$$\vec{Y} = \vec{Z} + \sum_{i=1}^{m} \alpha_i * \vec{V}_i \tag{1}$$

Any such $\vec{Y}$ satisfies our linear constraints. The goal now is to find a set of α_i that makes $\vec{Y}$ satisfy the non-linear constraints as well. We will show in detail how this is done for the constraints of type 4, and then outline handling of constraints of type 6. The key idea is to view Equation 1 as a matrix equation by letting V be the matrix whose columns are vectors $\vec{V}_i$. Then, we have $V * \vec{\alpha} = \vec{Y} - \vec{Z}$.

This equation has two unknowns, $\vec{\alpha}$ and $\vec{Y}$. We can choose vector $\vec{\alpha}$ arbitrarily, but some of the entries of $\vec{Y}$ are limited by constraints of type 4. Now, because at this stage we are only interested in solving the non-linear constraints of type 4, we eliminate from the above equation rows corresponding to y^i_j on which we don't have constraints of type 4. We call the new matrix V' and the new vectors $\vec{Y}'$ and $\vec{Z}'$.

$$V' * \vec{\alpha} = \vec{Y}' - \vec{Z}' \tag{2}$$

Once we have this equation, we have two choices. One alternative is to search the space of $\vec{\alpha}$'s until we find an $\vec{\alpha}$ that makes the resulting $\vec{Y}$ satisfy all the non-linear constraints, both of type 4 and 6.

Another alternative is to try the different alternatives $\vec{Y}'$ permitted by the type 4 constraints until we find $\vec{Y}'$ that is in the column span of V'. In this case, we know from basic linear algebra that $\vec{Y}'$ is in the column span of V' as long as $(V' * V'^{+} - I) * (\vec{Y}' - \vec{Z}') = 0$, where V'^{+} is the pseudoinverse $V'^{+} = (V'^{T} * V')^{-1} * V'^{T}$. This implies

$$A * \vec{Y}' = \vec{B} \tag{3}$$

where $A = (V' * V'^{+} - I)$ and $\vec{B} = A * \vec{Z}'$.

For each $y^{k_i}_{b_j}$ in $\vec{Y}'$ we have a set of choices of the form $y^{k_i}_{b_j} = (a \text{ or } b \text{ or } \ldots)$ (remember we eliminated those entries of $\vec{Y}$ for which we didn't have constraints of type 4). Thus, we need to search through all these choices until we find a set of choices that satisfy Equation 3. In principle, we may have to explore all possible combinations of the possible values for each of the $y^{k_i}_{b_j}$, but in practice, StreamBit first puts A in reduced row echelon form (*rref*). In most cases this allows they $y^{k_i}_{b_j}$ to be isolated into small clusters that can be searched independently, reducing the exponential blowup. For example, in the case of the sketch for DropThird, the *rref* reduction means that instead of having to search a 2^{3*16} search space, we have to do 16 searches on spaces of size 2^3.

Once the search is done, we have a $\vec{Y}'$ vector that we can use to find a set of solutions $\vec{\alpha}$ to Equation 2. Now, given an $\vec{\alpha}$ that satisfies Equation 2 for a $\vec{Y}'$ found through this process will make $V * \vec{\alpha} + \vec{Z}$ satisfy all the constraints of type 1 through 5, so now we have to pick one of these $\vec{\alpha}$'s that also satisfies constraint 6. Note, however, that at this point we are only searching among those decomposition vectors that have satisfied constraints 1 through 5.

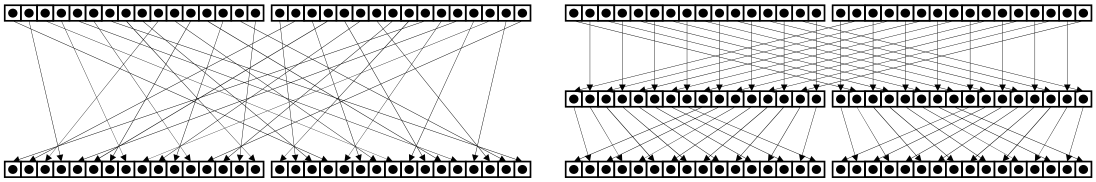
5.2 Restructuring

Restructuring transformations replace a sub-tree in the AST with an equivalent but structurally different sub-tree. For example, restructuring may reorder filters in a pipeline, hoist filters out of splitjoins, sink them into splitjoins, or coalesce several leaf filters into a single leaf filter. Restructuring transformations are typically enabling transformations that lead to dramatic code improvements through subsequent permutation decomposition. Sketching helps in performing restructuring by avoiding the need to specify the values of matrices for filters composing the new filter structure. For example, in our implementation of DES, we moved a filter across a joiner, and sketching automatically computed the "compensating" filter that had to be inserted into the other input of the joiner.

5.3 Sketching Decompositions of Affine Functions

It turns out that (sketches of) permutations are useful when implementing the more general affine filters. In this setting, permutations are often used to efficiently pack bits into words with the goal of fully exploiting word-level parallelism. Consider a `xor` filter with the matrix [1 1]. This filter takes two consecutive bits from the input stream and xors them to produce a single output bit. To implement this filter efficiently on a machine with a w-bit word, we want to permute $2 * w$ consecutive bits of the input stream such that all odd bits are in the first word and all even bits are in the second; after this transformation, the two words can be xor-ed with full word-level parallelism.

This permutation can be achieved in three steps: first, using restructuring, insert a $2 * w$-bit identity filter in front of the $[1\ 1]$ filter. Next, the following sketch shuffles bits as desired using log-shifting; specifically, the bits will be placed as desired at the entry to the last stage, which is unspecified in the sketch. After sketching, this stage will shuffle bits back into the original position so that the whole sketched pipeline remains an identity. In the last step, use restructuring to merge the last stage of the pipeline with the



(a) (b)

Figure 8. (a) A truncated version of the DES *IP* permutation. (b) Same permutation decomposed through sketching into a pipeline exposing two identical permutations in the second stage, which can now be implemented using a single table, reducing table storage four-times.

[1 1] filter. This step will modify the [1 1] filter to operate on the transformed stream.

```
SketchDecomp[
  [shift(1:2*w by 0 ||  1 ||  -1)],
  [shift(1:2*w by 0 ||  2 ||  -2)],
  [shift(1:2*w by 0 ||  4 ||  -4)],
  [shift(1:2*w by 0 ||  8 ||  -8)],
  [shift(1:2*w by 0 || 16 || -16),
  pos(1:2:2*w to 1:w), pos(2:2:2*w to w+1:2*w)],
  []
];
```

5.4 Table Conversion

Tables play an important role in efficient cipher implementations. In particular, a filter with n input bits and m output bits can be implemented as a lookup in a table with 2^n entries of m-bit values. To reduce the table size, the filter can be implemented using k tables with $2^{n/k}$ entries each using transformations based on the column equivalence from Figure 5.

When filter properties permit, table conversion can be further optimized, with great help from sketching. Consider the *IP* permutation from the DES cipher shown in Figure 8(a). With sketching, this permutation can be decomposed into a two-stage pipeline, shown in Figure 8(b), which the programmer obtained as follows: after examining the *IP* permutation, he observed certain regularity in how *IP* shifts bits, which lead him to suspect that *IP* is a composition of two narrower permutations. Guided by the observed regularity, he sketched the first pipeline stage and the compiler produced the second stage. The result of the sketch is an efficient implementation: the first stage is very efficient; the second stage contains two table lookups that are not only narrower but also identical. As a result, the total table size was reduced four-fold, producing a speedup of over 65% on an IA64 machine.

6. Evaluation

In this section, we quantify the productivity and performance rewards of sketching as compared to manually tuning programs written in C. We also show that the performance of a sketched implementation is competitive with that of heavily optimized implementations for a couple of well known ciphers.

6.1 User Study

We held a *user study* to evaluate the productivity and performance rewards attributed to sketching in general and StreamBit in particular. Specifically, we were interested in two questions.

- *Time to first solution.* How quickly can a reference program be developed and debugged by a programmer unfamiliar with the StreamIt dataflow programming language? In summary, we found that novice StreamIt programmers develop the first working solution faster than C programmers do (in C).
- *Performance of base compiler.* How good is the code generated from the reference program by the base compiler? In other words, can the base compiler compete with rapidly developed and tuned C code? In summary, we found that the base-compiled StreamBit code runs at least twice as fast as the C code, which took nearly twice as long to tune.

6.1.1 Methodology

The study invited participants to implement a non-trivial cipher based on Feistel rounds. The cipher consisted of an initial permutation of the bits in the input message, followed by three rounds of key mixing operations and a non-linear transformation. The cipher is formally described by the following equations:

$$\begin{aligned} X_0 &= \mathit{IP}(M) \\ X^f_{i+1} &= X^b_i \oplus F(X^f_i, K_i) \\ X^b_{i+1} &= X^f_i \\ F(X_i, K_i) &= L(K_i \oplus P_i(X_i)) \end{aligned}$$

where M is the input message, *IP* is the application of an initial bit permutation, F performs the key mixing subject to some permutation P, and L applies a non-linear transformation. Each X_i is split into a front half X^f_i and a back half X^b_i.

We recruited two sets of users: one set implemented the cipher in C and the other group implemented the cipher in StreamBit[3]. In all, there were six C participants of which five finished the study and submitted working ciphers. On the StreamBit side, there were seven participants of which four submitted working ciphers[4]. The study participants were all well-versed in C but none had any experience with StreamBit prior to the study; although they were provided with a short tutorial on StreamBit on the day of the study.

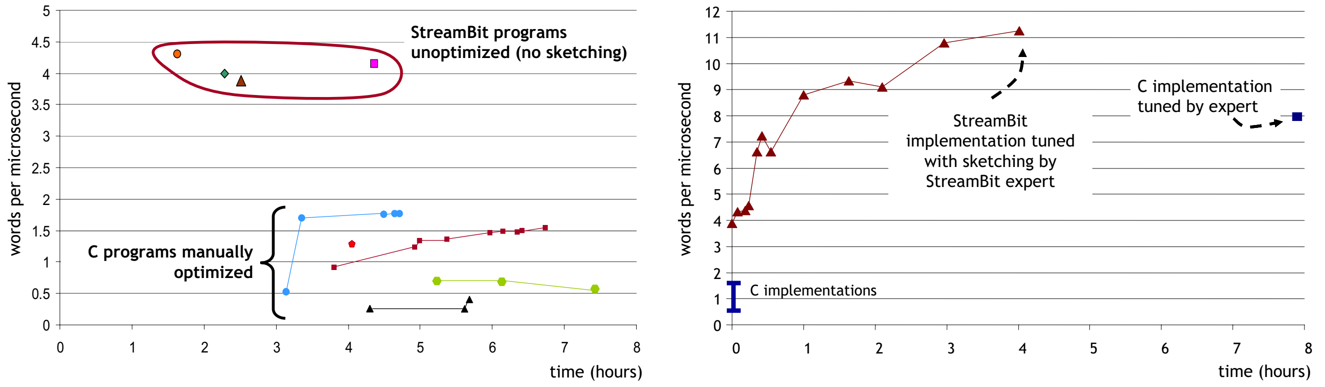
C participants were also provided with a short tutorial, this one on well known (bit-level) optimizations, and we encouraged them to apply any other optimization ideas they could think of. In the case of C participants, we did not restrict the number of submissions that a participant could make, and instead encouraged performance tuning of the initial solution.

6.1.2 Results

In Figure 9(a), we report the results from our user study. The x-axis represents development time (in hours), and the y-axis represents the performance of the ciphers in units of encrypted-words per microseconds. Each point on the graph represents the development time and performance of a single participant; the C users have multiple points per participant connected by solid lines showing their progress over time. It is readily apparent that the StreamBit partic-

[3] The programming language used in StreamBit is StreamIt with a few extensions as detailed in Section 2. Solely for the sake of clarity will we refer to the language as StreamBit.

[4] Users who left the study early chose to do so due to personal constraints.



(a) (b)

Figure 9. StreamBit vs. C : performance as a function of development time. (a) Comparison of first solutions for StreamBit and C implementations. (b) Performance improvement through sketching.

ipants spent between two and four hours implementing the cipher and achieved better performance compared to the C participants.

We also note that all but one of the C participants tried to tune the performance of their ciphers. The C participants spent between one and three hours optimizing their implementations, and while some improved their performance by 50% or more, the StreamBit ciphers were still two and a half times faster. The results highlight the complexity of tuning bit-level applications, and demonstrate the challenge in understanding which optimizations pay off.

The data also suggest that the scope of optimizations a programmer might attempt are tied to the implementation decisions made much earlier in the design process. Namely, if an optimization requires major modifications to the source code, a programmer is less likely to try it, especially if the rewards are not guaranteed.

The C implementations were compiled with gcc version 3.3 and optimization level -O3 -unroll-all-loops. The base StreamBit compiler produced low-level C code. This C code was subsequently compiled with gcc and the same optimization flags. All resulting executables were run on an Itanium2 processor.

Our sample of users is relatively small, so it's hard to draw very definitive judgments from it, but in terms of the two questions we wanted to answer, we can see that it *is* possible for someone who has never used StreamIt to produce a working solution in less time than it would take an experienced programmer working in C. We can also see that the performance of the base compilation algorithm is very good compared with the performance of handwritten C code, even after it has been tuned for several hours.

6.2 Benefits of Sketching

The user study showed that the StreamBit system can be a good choice when developing prototypes of ciphers, because it allows for the code to be developed faster, and the code is actually of much better quality than code produced by hand in a comparable amount of time. Next, we wanted to evaluate a sketched implementation with a heavily optimized MiniCipher from the user study and also with widely-used cipher implementations.

6.2.1 Optimizing the MiniCipher

The user study also provided us with an opportunity to evaluate the separation of concerns afforded by sketching, and the potential for performance improvement in a benchmark more realistic than DropThird.

First, we assigned a *performance expert* (one of the authors) to select one of the StreamBit reference programs written in the user study, and sketch for it a high-performance implementation. The sketching expert managed to iterate through ten different implementations in four hours, tripling the performance of the base-compiled code, which is a huge improvement considering the base StreamBit implementation was already twice as good as the C implementations from the user study. It is worth noting that this sketching was done with code produced by another developer who had no contact with the performance expert. The performance expert did have a list of implementation ideas to try in his sketches; the same list was available to the user study participants.

As a point of comparison, we asked another member of our research group to serve as a C performance expert and tune an already working C implementation. He was done in just under eight hours, and achieved a performance of eight encrypted-words per microsecond. It must be said that of those eight hours, about 3/4 of an hour were spent understanding the reference implementation and the specification. The results from this exercise are reported in Figure 9(b).

The sketching methodology thus affords programmers the ability to prototype and evaluate ideas quickly, as they are not concerned with low-level details and can rest assured that the compiler will verify the soundness of their transformations. This is in contrast to the C performance expert who must pay close attention to tedious implementation details lest they introduce errors. As an added advantage, programming with sketches does not alter the original StreamBit code which therefore remains clean and much easier to maintain and port compared to the manually tuned C implementation.

6.2.2 Implementation of Real Ciphers

LibDES from OpenSSL We compare StreamBit generated code with a libDES, a widely used publicly available implementation of DES that is considered one of the fastest portable implementations of DES [21]. LibDES combines extensive high-level transformations that take advantage of boolean algebra with careful low-level coding to achieve very high performance on many platforms. Table 1 compares libDES across different platforms with DES implementations produced by StreamBit.

We were able to implement most of the high-level optimizations that DES uses, and even a few more that were not present in libDES, including the one described in Section 5.4. Our code was missing some of the low-level optimizations present in libDES. For example, our code uses lots of variables, which places heavy strains on the register allocation done by the compiler, and it assumes the compiler will do a good job with constant folding, constant propagation and loop unrolling.

Even with these handicaps, we were able to outperform libDES on at least one platform. The 17% degradation on Pentium III is mainly due to not implementing a libDES trick that our sketching currently does not support. Finally, it is worth noting that whereas the libDES code is extremely hard to read and understand, the StreamBit reference program reads very much like the standard [9].

processor	P-IV	P-III	IA64	Solaris	IBMSP
performance	0.90	0.83	1.06	0.91	1.07

Table 1. Comparison of sketched DES with libDES on five processors. Performance is given as ratio of throughputs; sketched DES was faster on IA64 and IBM SP.

Serpent Serpent is considered the most secure of the AES finalists[5]. Serpent is particularly interesting from the point of view of sketching because of the way it was designed. The cipher is defined in terms of bit operations including permutations, as well as linear and non-linear functions. However, all of these functions were defined in such a way that a particular transformation known as bit-slicing, together with some additional algebraic manipulation, would produce a very efficient implementation of the cipher for regular 32-bit machines [2].

As part of their submission, the Serpent team developed both a reference implementation using the bit-level definition as well as an optimized version of the cipher.

Table 2 shows our results for Serpent. It is worth pointing out that the level of abstraction of the StreamBit code is comparable to that of the bit-level reference implementation, yet the base compilation algorithm produced code that was an order of magnitude faster than the C reference implementation. This is not a small matter, considering that the reference implementations are used to generate correct outputs for large sets of inputs. The process of generating reference outputs can take several hours, and is on the critical path for testing optimized implementations.

By simply sketching the bit-slicing transformation, we were able to achieve half of the performance of the heavily optimized implementations. We believe that the other half of performance can be had by implementing in our compiler a smarter conversion from an arbitrary boolean function defined as a table-lookup into one defined as a boolean expression, a transformation used as part of bit-slicing. Furthermore, on the IA64, we were able to get 90% of the performance of the hand-optimized implementation by using 64-bit-specific optimizations. Granted, the optimized AES Serpent did not use 64-bit-specific optimizations, but it speaks of the flexibility of our approach that we were able to modify our sketch to produce 64-bit-specific optimizations by changing only a single line in the sketch.

processor	reference	base StreamBit	bit-sliced StreamBit	64-bit StreamBit
IA64	0.003	0.13	0.478	0.90
P-IV	0.003	0.08	0.62	N/A

Table 2. Several Serpent implementations compared to the official optimized Serpent. Performance is given as ratio of throughputs; sketched Serpent achieved 62% of "official" performance despite infrastructure limitations.

7. Related Work

Our presentation of Sketching was inspired by some of the work on partial programming in the AI community. For example, ALisp [3], developed by Andre and Russell to program Reinforcement Learning Agents is a form of lisp extended with non-deterministic constructs. In ALisp, the behavior of the non-deterministic branches is defined through learning. Sketches in StreamBit can also be thought of as non-deterministic descriptions of an algorithm, but their behavior is determined not through learning, but by matching them with the reference program.

The sketch resolution problem is a constraint satisfaction problem similar to those studied by the constraint programming community [11].

The idea of separating the task description from the implementation specification has been explored before. Ennals, Shart and Mycroft [7], for example, have built a system for programming network processors that allows the user to describe the task in a high-level domain-specific language and then specify an implementation by applying a series of semantics preserving transformations. Their transformations, however, involve no sketching, and have to be fully specified.

Aspect-Oriented Programming (AOP) aims at supporting the programmer in "cleanly separating concerns and aspects from each other, by providing mechanisms that make it possible to abstract and compose them to produce the overall system" [14]. Our approach can be understood as a form of AOP, where the algorithm specification and the performance improving transformations are the two aspects we are dealing with. There have been other efforts at applying AOP to restricted application domains, for example Irwin et. al. demonstrate the use of AOP in the domain of sparse matrix computations [12].

Efficiently compiling domain-specific languages to achieve high-quality implementations without user intervention has been an area of active research for a number of years. Padua et. al. have been very successful in producing high-quality code from MATLAB through the use of aggressive type and maximum matrix size inference [1].

Kennedy et. al. have worked on an approach called telescoping languages. The idea is that in many scripting languages, domain-specific abstractions are provided through libraries. The telescoping language approach is to preprocess those libraries to create specialized versions of them to allow for aggressive optimization of the programs that use them without having to incur the cost of extensive interprocedural analysis at compile time [5, 13].

One approach that has recently gained popularity in the high-performance community has been the use of search to find optimal implementations for important kernels. The idea is to search through a suitably restricted implementation space by actually generating code for the possible implementations and either run them directly on the machine, or do very detailed simulation. For example, FFTW [10] uses a planner to try many different execution plans for an FFT at run-time and find the best one. SPIRAL [18] is another example of the search based approach; it generates high-performance DSP kernels by searching the space of possible implementations, taking advantage of the structure of the algorithm and implementation space to speed up the search. Demmel et. al. [6] also use search based methods to generate dense and sparse linear algebra kernels. Wu, Weaver and Austin [20] developed an architecture aimed at executing ciphers, and they also used a search based method to derive optimal implementations of the ciphers for their architecture. However, the search was simply to find an optimal way to map a dataflow graph for the already optimized ciphers into machine instructions.

We believe sketching complements the search based methods, because instead of having to restrict the space of implementations a priori to make the search tractable, the sketch encodes knowledge the user has about the best way to implement an algorithm, allowing the search to proceed faster.

[5] In 2000, a competition was held to define the new Advanced Encryption Standard, to replace DES. Serpent was one of the finalists, and was widely considered the most secure of the candidates.

Finally, our work builds very heavily on StreamIt [15, 19]. The StreamIt compiler automatically identifies linear filters, and performs many optimizations targeted towards DSP applications. The StreamIt language itself, traces its roots to the work on Synchronous Data Flow by Edward Lee and his group since the 80's [16, 4].

8. Conclusion

In programming by sketching, the developer outlines the implementation strategy and the compiler fills in the missing detail by ensuring that the completed sketch implements the reference program, which serves as a full behavioral specification.

In this paper, sketching is explored in the context of bitstreaming. We believe that sketching is also applicable in other domains where a domain algebra for semantics-preserving restructuring of programs is available. One atractive candidate is sparse matrix computation.

Depending on the development setting (and the point of view), benefits of sketching can take different forms. First, sketching allows collaboration between a domain expert (e.g., a crypto expert) and a performance expert who understands the processor and a little about the domain algebra, such as bit permutations. While the former specifies the crypto algorithm and keeps modifying it, the latter provides an efficient implementation by preparing a separate sketch. Via separation of concerns, sketching thus allows collaboration of programmer roles.

An alternative view is that sketching is a method for rapid prototyping of an effective domain-specific compiler. Rather than implementing an analysis and a transformation, which may not be economical for a domain-specific compiler, we give a sketch that is filled in according to the provided reference program. While a sketch is not as generally applicable as an optimization, it is easy to port to another reference program; it is also tailored to a given reference program, which gives it unique power.

Acknowledgments

This work is supported in part by the National Science Foundation, with grants CCF-0085949, CCR-0105721, CCR-0243657, CNS-0225610, CCR-0326577, an award from University of California MICRO program, the Okawa Research Award, as well as donations from IBM and Intel. This work has also been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCHC020056. The views expressed herein are not necessarily those of DARPA or IBM.

The StreamIt project is supported by DARPA grants PCA-F29601-03-2-0065 and HPCA/PERCS-W0133890; NSF awards CNS-0305453 and EIA-0071841; and the MIT Oxygen Alliance.

We would like to thank Vivek Sarkar for his invaluable help in initiating this project and organizing the user study. Our gratitude also goes to the many user study subjects who suffered through the glitches of our infrastructure.

We would also like to thank Brian Fields, Dave Mandelin, Bill Thies and Renju Thomas for helping with preparations for the user study.

References

[1] G. Almasi and D. Padua. Majic: Compiling matlab for speed and responsiveness. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.

[2] R. Anderson, E. Biham, and L. Knudsen. Serpent: A proposal for the advanced encryption standard. The implementation we tested can be found at http://www.cl.cam.ac.uk/ rja14/serpent.html.

[3] D. Andre and S. Russell. Programmable reinforcement learning agents. *Advances in Neural Information Processing Systems*, 13, 2001. MIT Press.

[4] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *Int. Journal of Computer Simulation*, 4:155–182, April 1994. special issue on "Simulation Software Development".

[5] A. Chauhan, C. McCosh, and K. Kennedy. Automatic type-driven library generation for telescoping languages. In *Proceedings of SC: High-performance Computing and Networking Conference*, Nov. 2003.

[6] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, C. Whaley, and K. Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2), 2005.

[7] R. Ennals, R. Sharp, and A. Mycroft. Task partitioning for multi-core network processors. In *Compiler Construction*, Edinburgh, Scotland, April 2005.

[8] N. Ferguson and B. Schneier. *Practical Cryptography*. Wiley Publishing Inc, 2003.

[9] Data encryption standard (des). U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology, December 1993. http://www.itl.nist.gov/fipspubs/fip46-2.htm.

[10] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft. In *ICASSP conference proceedings*, volume 3, pages 1381–1384, 1998.

[11] P. V. Hentenryck and V. Saraswat. Strategic directions in constraint programming. *ACM Comput. Surv.*, 28(4):701–726, 1996.

[12] J. Irwin, J.-M. Loingtier, J. R. Gilbert, G. Kiczales, J. Lamping, A. Mendhekar, and T. Shpeisman. Aspect-oriented programming of sparse matrix code. In *Proceedings International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, number 1343 in LNCS, Marina del Rey, CA, 1997. Springer-Verlag.

[13] K. Kennedy, B. Broom, A. Chauhan, R. Fowler, J. Garvin, C. Koelbel, C. McCosh, and J. Mellor-Crummey. Telescoping languages: A system for automatic generation of domain languages. *Proceedings of the IEEE*, 93(2), 2005.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in LNCS. Springer-Verlag, June 1997.

[15] A. A. Lamb, W. Thies, and S. Amarasinghe. Linear analysis and optimization of stream programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003.

[16] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, September 1987.

[17] M. Morgan. http://www.schneier.com/blowfish-bug.txt.

[18] M. Püschel, B. Singer, J. Xiong, J. Moura, J. Johnson, D. Padua, M. Veloso, and R. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Journal of High Performance Computing and Applications*, accepted for publication.

[19] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.

[20] L. Wu, C. Weaver, and T. Austin. Cryptomaniac: A fast flexible architecture for secure communication. In *28th Annual International Symposium on Computer Architecture (28th ISCA 2001)*, Goteborg, Sweden, June-July 2001. ACM SIGARCH / IEEE.

[21] E. Young. http://www.openssl.org. libDES is now part of OpenSSL.