

Programming by Sketching for Bit-streaming Programs

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodik, and Kemal Ebcioglu

{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. With sketching, programmers write clean and portable reference code, and then obtain a high-quality implementation by simply *sketching* the outlines of the desired implementation. The compiler then fills in the missing detail and ensures that the completed sketch implements the reference code. In this paper, we develop sketching 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 has several benefits to programmer 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 orchestrating low-level details. *Third*, the sketching compiler rejects a “buggy” sketch, thus providing correctness by construction and also allowing the programmer to quickly evaluate sophisticated implementation ideas.

We have evaluated our programming methodology from a productivity and a performance perspective in a user study, where a group of novice StreamBit programmers competed with a group of experienced C programmers on implementing a cipher. We learnt that, given the same time budget, the cipher developed in StreamBit ran $3 - 5\times$ faster than ciphers coded in C. Furthermore, our sketching-based implementation of the DES cipher, developed and tuned in just four hours, improved performance 4-fold and on some machines matched the performance of libDES, the best publicly available DES implementation.

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 implementation are often daunting. Word-level implementations are essential because they can deliver an order of

magnitude speedups, which is important for servers where security-related processing can consume up to 95% of processing capacity [16]. At the same time, the characteristics of bit-streaming codes render vectorizing compilers largely ineffective. In fact, widely used cipher implementations, almost all hand-written, often achieve performance thanks to algebraic insights into the algorithms 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 [12]. 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; the problem is that many ciphers, especially those based on Feistel rounds, correctly 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 ensures correctness, and 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, we first introduce a running example that we will use throughout the paper. The example also illustrate the complexities in implementing bit-streaming programs.

Our running example is a bit manipulation problem we refer to as DropThird. It is concisely described as follows: “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 in ciphers.

To illustrate why this is a non-trivial example, consider a general-purpose processor with the usual suite of bitwise operations: *and*, *or*, *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. Furthermore, even after the bits are prearranged into words, various sophisticated intra-word bit processing can lead to significant differences in performance. This is illustrated in Figure 1. The naive scheme shown on the left needs $O(N_w)$ steps, while the smarter scheme (on the right)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

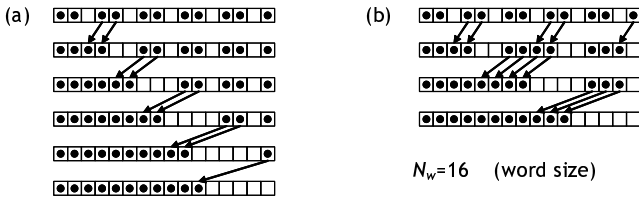


Figure 1: Two algorithms for dropping every third bit from a word: (a) naive implementation with a $O(N_w)$ running time, referred to as SLOW throughout the paper. (b) log-shifter implementation with a $O(\log(N_w))$ running time, referred to as FAST throughout the paper.

requires $O(\log(N_w))$ steps; $N_w = 16$ 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, inexplicably, it is $34\times$ 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.* The low-level code of the implementation is typically disproportionate in size and complexity to its simple specification. One log-shifter implementation requires 140 lines of FORTRAN code to implement a slight modification of `DROPThird`. Part of the problem is that in `DROPThird`, one must use different bit masks for each stage of the log-shifter (because any three consecutive words in the input stream each drop differently positioned bits).
- *Malleability and portability.* The low-level code is hard to modify given even simple changes to the bit-stream manipulation program, such as changing the specification to drop the second bit (instead of the third) 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 (its shifts are data independent). To find a suitable implementation, the programmer typically implements and measures each.

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 [15], 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. We compile the reference program to assembly code with a simple compiler that we call a *naive compiler*. The naive compiler exploits word-level parallelism but only using a simple greedy strategy.

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 completed, it implements the reference program; that is, the completed sketch is behaviorally equivalent to the reference program. The implementation details are obtained by decomposing the reference program into high-level algorithmic steps (e.g., packing bits into words) which are in turn decomposed into lower-level steps (e.g., shifting bits within words).

The problem with sketching an outline of the implementation is that the programmer may want to sketch the implementation at arbitrary levels of the hierarchy, ideally without referring to uninteresting levels. Our solution to make sketching convenient is to (i) make the naive compiler follow a simple and fixed translation sequence which (ii) may be replaced at any step by the user-provided sketch. Effectively, the input program automatically undergoes one or more transformations as dictated by the sketch, the result of which is an implementation that is faithful to what the performance expert desires. After the sketch is processed, the naive compiler resumes its role and completes the compilation. The idea is that the naive compiler will perform an optimal translation if the difficult global decisions are supplied via sketches—this is especially important when the complexity of an optimization or transformation are well beyond the scope of what a compiler can autonomously discover.

As mentioned above, sketches are transformations that guide the naive compiler. What makes them sketches is that they need to be specified only partially. A simple example is when the user wants to factor a bit permutation P into a sequence (i.e., a pipeline) of two permutations P_1 and P_2 , which are less expensive to implement than P . In this case, he may specify P_1 , and the compiler will determine P_2 from P_1 and the reference permutation P . Alternatively, P_1 and P_2 can be described by the programmer only partially, with a set of constraints that are satisfied by more than one permutation. Our compiler then selects P_1 and P_2 such that together they implement P . Figure 2 illustrates the key idea behind sketching. On the very left is the reference program in visual form. This bit manipulation task is translated by the naive compiler into a task that is equivalent, but also word-aligned. Now, if the naive compiler was allowed to continue, it would produce 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, etc. The sketch doesn't specify which bits must shift in each step, and 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. If the sketch is not satisfiable under the given specification, the sketch 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.

An added benefit of the lack of low level details is the robustness of the sketch. For example, note that even though the implementation of the log-shifter is slightly different for different words,

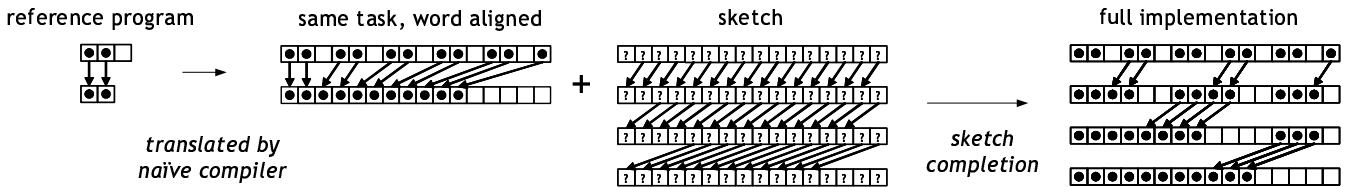


Figure 2: An illustration of programming with sketches. The naive compiler first unrolls the reference program to operate on complete words of bits. Then, the compiler combines this word-aligned task (a full specification of the bit-stream manipulation function) with the sketch (a partial specification of the implementation) to derive the full implementation that performs the desired functionality.

the same sketch applies to all of them because the general strategy which the sketch encodes 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 change the reference program; the original sketch remains applicable. This gives the sketch a level of reusability 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. To compile sketches, we develop a constraint solver which uses the reference program and the sketch to derive a complete implementation.
- We develop an Transformation Specification Language (TSL) that allows programmers to specify a large number of implementations. The TSL is a high-order language: it manipulates the program written in StreamIt by the domain expert.
- We performed a user study comparing the productivity of programming in StreamBit versus C. We observed that in the same amount of time, StreamBit programmers produced code that ran $3\times$ faster than C implementations.
- We used StreamBit to generate an implementation for DES which is early as to read as the NIST specification, and with an additional 25 lines of TSL specification, we improved performance 4-fold and on some machines matched the performance of libDES, the best publicly available DES implementation.

2. STREAMIT

Our programming methodology uses StreamIt as the language for writing the reference programs. StreamIt is also used as an intermediate representation during naive compilation and sketching. StreamIt [15] embraces a synchronous dataflow model of computation. Its main unit of abstraction is the *filter* which reads data from a FIFO input stream and writes data into a FIFO output stream. Additionally, as a consequence of being a synchronous dataflow language, all filters must statically specify their input and output rate (which is not a restriction for our domain). This enables the compiler to statically schedule filter execution.

Filters may be composed into *pipelines* or *splitjoins*. A pipeline is simply a sequence of filters that are linked together such that the output of filter i is the input of filter $i + 1$. Several splitjoins are illustrated in Figure 5 (a figure designed mainly to illustrate some other points). A splitjoin contains a *splitter* which distributes its input stream among its N children, and a *joiner* which collects and merges the output of all N children. We currently support two different types of splitters: *duplicate* splitters and *roundrobin*

splitters. A duplicate splitter, as the name implies, 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. 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 (this joiner effectively concatenates its inputs).

The StreamBit compiler is concerned with bit level manipulations, and hence it concentrates on filters that have input and output streams of type *bit*. The compiler attempts to convert filters into Boolean expressions; for that, it requires that all loops trip counts are fixed at compile time. Filters may otherwise include more arbitrary code, although sketching is restricted to those filters that satisfy the static properties just mentioned. While the static constraints may appear restrictive, we note that they do not hinder the implementation of public key ciphers—since in general, data dependent computation renders ciphers vulnerable to timing attacks and are therefore largely undesirable. The focus on bit operations is not restrictive either. In fact, we implemented the AES cipher (which is not generally considered to be a bit-streaming filter) using our bit-streaming compiler.

Furthermore, in addition to the StreamIt joiners, we support a small set of Boolean joiners for programming convenience. Namely, we allow for AND, OR and XOR joiners that merge input stream bit-wise using these Boolean operators.

The use of StreamIt provides several advantages compared to a general-purpose programming language. For example, from the point of view of the programmer, StreamIt combines modularity with implicit scheduling and buffer management. Consider a filter A that produces 5 bits at a time, and filter B that consumes blocks of 6 bits at a time. In StreamIt, the user can simply compose the two filters into a pipeline, without having to worry about how to interleave the executions of the filters, or how much buffer space to allocate to store the results of A before B can read them.

3. PROGRAM REPRESENTATION

The synchronous dataflow programming model of the StreamIt language is also used as an intermediate representation, employed both by sketching and the naive compiler. The key invariant is that both the reference program and the machine-level implementation is expressed as a dataflow program.

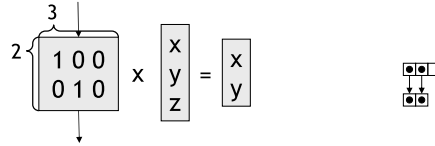
Internally, we represent the dataflow program as an abstract syntax tree (AST) that represents the hierarchical structure of the StreamIt program. The leafs of the AST correspond to filters, which are Boolean functions that translate N_{in} bits of input into N_{out} bits of output. The intermediate nodes correspond to the potentially nested splitjoins and pipelines; they represent the communication network that defines how values flow across filters.

```

filter DropThird{
  Work push 2 pop 3{
    for(int i=0; i<3; ++i){
      x = peek(i)
      if(i<2) push(x);
      pop();
    }
  }
}

```

(a)



consumes a 3-bit chunk of input;
produces a 2-bit chunk of output.

(b)

Figure 3: Our running example is a single a bit-stream filter that generates its output stream by dropping every third bit in its input stream. (a) The reference program in the StreamIt language. This code does not specify a particular implementation of the filter. (b) Two alternative representations of the same filter. Because the filter is linear, it can be represented by a matrix. Each three bits popped from the input stream form a vector. This vector multiplied by the matrix produces the corresponding part of the output stream.

Filters (AST leaves) are represented as matrix together with an offset vector. As shown in Figure 3, the filter’s boolean function corresponds to a multiplication of the input bit block with the matrix, followed by the addition of the offset vector. The matrix representation of filters gives us a simple yet powerful algebra for manipulations of dataflow programs.¹ The algebra will be used both during naive compilation and during sketching.

4. THE NAIVE COMPILER

The goal of the naive compiler is to translate the reference program into machine instructions. This process boils down to a transformation that convert the reference program into an equivalent StreamIt program where all the filters correspond to instructions available in hardware, like shifts and masks. This target program will be considered to be in *Low Level Form (LL-form)*. Since LL-form programs consist of basic machine operations along with a communication network that defines how values flow among different operations, they can be readily mapped to C code; the only optimizations left to do on them are register allocation, code scheduling, and the like, which modern compilers can do quite well.

LL-form programs are defined formally as programs with two properties: (1) Their filters (AST leaves) correspond to one of the basic instructions in the target our machine; and (2) Their communication network (AST internal nodes) transfers data in word-size units.

Let us first deal with the second property. In the case of pipelines, this property follows directly from the first property, because the links in the pipeline will transfer data in the same units that it are produced by the components of the pipeline. The only case where property two is an issue is in the case of roundrobin splitters and joiners that distribute bits in units different from a word-size. However, roundrobin splitters and joiners can be replaced replacing with duplication splitters and aggregation joiners after adding the appropriate filters on each branch; these filters select the appropriate bits. This transformation is illustrated in Figure 4 After this transformation, we need to worry only about Property 1.

The naive compilation algorithm exploits the fact that property one pertains exclusively to the leaf filters. It works by going directly to the leaf filters, and growing them, breaking them and decomposing them until they satisfy property one for LL-form. Because it acts independently on each leaf filter, it misses optimization

¹The matrix multiplication algebra gives us all the expressiveness we need, because an arbitrary Boolean expression can be represented as a pipeline of affine transformations provided we have some flexibility in defining the addition and multiplication operators.

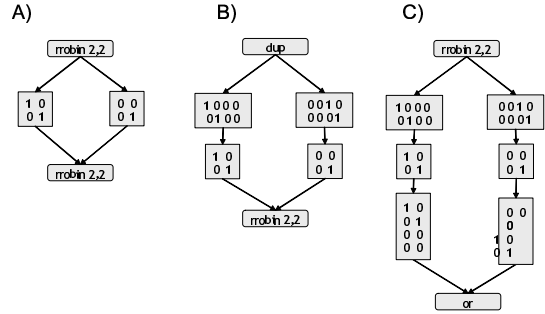


Figure 4: The splitjoin transformations. (a) The original filter. (b) The result of applying SRRT-ODUP on the splitjoin in (a). (c) The result of applying JRRT-ORXOR to (b). Note that the two transformations eliminated the round robin splitters and joiners, which is useful when the size of the round robin splitters and joiners is not a multiple of the word size.

opportunities that could arise from considering groups of filters together. Furthermore, the algorithm greedily tries to reduce the number of shifts that are done on a given bit, missing some opportunities to use word-level parallelism.

The Naive algorithm can be understood as working in two different stages: i) Size adjustment stage, and ii) instruction decomposition stage.

The role of the size adjustment stage will be to take a leaf filter in the AST of the program and replace it with a pipeline or splitjoin of filters which is equivalent to the original leaf filter, but such that all its leaf filters take in W bits and produce W bits, where W is the word size. Equivalence in this case must be defined generously to take into account the fact that a filter that reads N_{in} bits of input and produces N_{out} bits of output can be equivalent to a filter that takes $k * N_{in}$ bits of input and produces $k * N_{out}$ bits of output, as long as one execution of the latter corresponds to k executions of the former. It is important to note that because of the implicit scheduling and buffer management offered by StreamIt, it is possible to replace a filter with an equivalent filter, without having to worry about the effect that the potentially different input and output sizes will have on the rest of the program.

The size adjustment will be achieved through a combination of filter unrolling (to make the input and output sizes multiples of the word-size), and filter decompositions to make the filters produce word-size input and output. Figure 5 shows the complete sequence of unrolling and decomposition used for the dropThird filter.

The instruction decomposition stage will take each of the word-

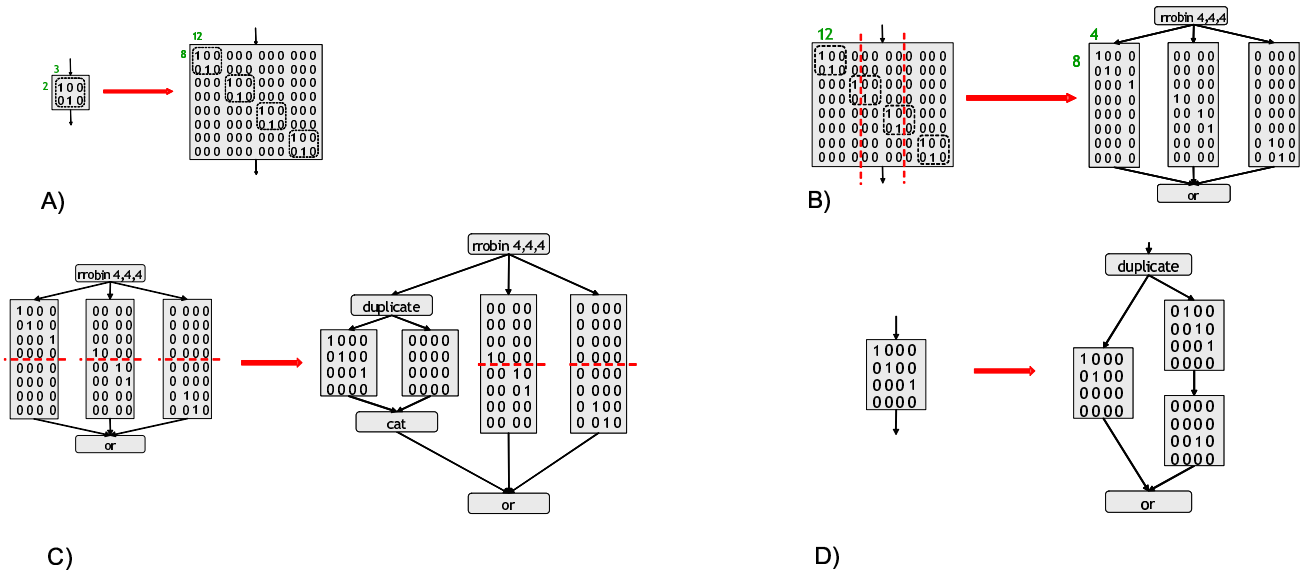


Figure 5: An illustration of the naive transformation algorithm for our running example, assuming a 4 bit machine. A) involves unrolling the filter to take in and push out a multiple of N_w , the word size. B) shows how `ColSplit[4]` breaks the filter into columns, each taking N_w bits from its input. C) shows the effect of `RowSplit[4]`. D) shows one of the filters from C) as it is finally converted into LL-form filters.

sized leaf filters produced by the size adjustment phase and decompose them into splitjoins and pipelines with all their leaf filters corresponding to individual bit operations. These bit operations we use are left and right logical shifts as well as bitwise AND, OR and XOR on words of size W .

The key idea of this phase is 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 bitmask 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 . This decomposition can be expressed in StreamIt because sums and products of matrices actually correspond to splitjoins and pipelines of filters respectively (see Figure 5 (D)).

This approach is actually quite general. For example, note that the approach is applicable for arbitrary word sizes, since both the diagonal matrices and the shifts behave the same way regardless of the word-size. Furthermore, it is easy to generalize for other types of instructions besides simple shifts and masks. For example, rotation instructions pose no major problem, since for positive i , $s_i = a_{pi} * r^{|i|}$ and for negative i , $s_i = a_{mi} * r^{|i|}$, where a_{pi} and a_{mi} 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 .

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 which correspond to shifting several contiguous bytes independently but in parallel, with bits that move from one byte to the other being lost. In that case, we have that $s^i = t^{i/8} * u^{i \bmod 8} + t^{i/8+1} * u^{8(i \bmod 8)}$, 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 $t^i * u^j$.

Regardless of the choice of instructions, the Naive algorithm has several interesting properties. First, the decompositions are unique,

and their results are easy to predict. In particular, the final number of operations will be roughly proportional to the number of diagonals in the original matrix. Additionally, the decompositions will generally be very efficient when compared with hand coded implementations of the filter because they will take some advantage of the parallelism available in the word, both by shifting together bits that will have to shift by the same amount and by ORing and masking complete words at a time. Finally, these decompositions in general will not be optimal. The problem with the decompositions is that they try to shift bits to their final position in one step. The example of `dropThird` shown earlier illustrates why this can be inefficient. The SLOW implementation actually corresponds to what the Naive algorithm will get you. In particular, bits that shift by the same amount are shifted together in the same shift, and all bits are shifted to their final position in a single movement. By contrast, the FAST approach involves moving bits partially along the way, in order to make better use of the shift in a subsequent step.

5. SKETCHING

As we have seen so far, there are two fundamental problems with the Naive algorithm. The first one is its local nature; by doing size adjustment and instruction decomposition independently one filter at a time, it misses optimization opportunities that can arise by looking at complete pipelines or splitjoins. The second problem comes from the direct nature of the instruction decomposition; by requiring bits to be shifted in one step to their final position, it misses on some of the potential word-level parallelism, as in the case of the `dropThird` example.

The traditional approach to handling these problems would be to add an optimization phase that applies some predefined set of transformations to the program based on some heuristics about what transformation sequences are likely to improve performance, and on some analysis to decide when the transformations are legal and when they are not.

Unfortunately, a couple of features of our domain that make this approach unsuitable. The most important one is the sheer size of the search space. As an example, consider a pipeline with N steps,

```

WSIZE=16;
Unroll[WSIZE](DropThird);
SketchDecomp[ [ shift(1:2 by 0),
                shift(17:18 by 0),
                shift(33:34 by 0) ],
              [ shift(1:16 by ?),
                shift(17:32 by ?),
                shift(33:48 by ?) ]
              ] ( DropThird );

DiagSplit[WSIZE](DropThird.DropThird_1);

for(i=0; i<3; ++i)
  SketchDecomp[ [shift(1:16 by 0 || 1)],
                [shift(1:16 by 0 || 2)],
                [shift(1:16 by 0 || 4)]
              ] ( DropThird.DropThird_1.filter(i) );

```

Figure 6: A sketch, specifying the log-shifting algorithm. The first `SketchDecomp` instructs the system to perform the `DropThird` task in 2 steps. The second step must shift all the bits within a word together, so the bits must be packed within a word by then. The first step is not allowed to move the first 2 bits in every word, so the bits will be packed towards the beginning of the word. The second `SketchDecomp` contains the sketch illustrated in figure 2. On each step it specifies the amounts by which each bit in the word is allowed to move.

each corresponding to a matrix. The first transformation that one may want to do is to restructure the pipeline by collapsing contiguous steps in the pipeline together in order to get fewer matrices and hopefully a smaller number of operations. It is easy to see that there are actually 2^N possible pipelines that one can derive by selecting how one collapses contiguous steps. But that's not all; we also need to be able to evaluate the cost of each of the 2^N possibilities in order to pick the best one. In order to do this, we need to know the cost of implementing each matrix in each of the pipelines. One option is to use the number of shifts required by the Naive algorithm, which is easy to measure simply by counting non-zero diagonals. However, remember that the Naive algorithm is not optimal, so to get a true sense of which restructuring is better we need to find an optimal decomposition for each of the matrices into steps, and then look at the cost of each step. The problem is that the number of such possible factorings for a matrix also grows exponentially with the size of the matrix. If we add to that the difficulty of estimating the performance on a real machine, once you factor in cache effects, register usage, dependencies among instructions, etc. One can see that searching for an optimal solution is not an ideal approach.

With sufficient engineering effort, it is quite likely that one could find a set of suitable heuristics and evaluation metrics that could guide the transformation and deliver reasonably good results on several machines in a reasonable amount of time. However, the domain is probably too narrow to justify the necessary investment on such a complex infrastructure.

Instead, our approach is to provide the programmer with the tools to specify these transformations at a high level, omitting low level details that the machine can figure out on its own, but confident that the system will not permit changes that affect the semantics of the program. This allows the developer to try different implementation strategies quickly and safely, and to take advantage of domain knowledge about the structure of the program, or of clever tricks that other programmers may have discovered.

There are two classes of transformations that StreamBit allows the user to sketch: Restructuring and Decomposition. Each ad-

resses one of the problems with the Naive compilation. The restructuring allows you to reorganize complete pipelines and splitjoins, merging filters, moving filters inside splitjoins, or hoisting them out, and merging pipelines and splitjoins together. The decomposition on the other hand, allows you to break individual filters into steps, each corresponding to a step in the algorithm you want to use to implement the filter.

5.1 Sketching Decompositions

As we saw in the example of `dropThird`, there are cases when the Naive algorithm fails to deliver an optimal result because of its insistence in shifting bits directly to their final position in one step. In many cases, the user may know of a better algorithm to implement the filter in terms of shifts and masks, so we want the user to be able to guide the Naive compilation by specifying the high-level steps of the better algorithm. Figure ?? shows the sketch for the log-shifting for `dropThird` as entered by the user. As mentioned before, we want the user to have correctness guarantees, so she can feel confident to try several different implementations without worrying about introducing bugs. At the same time, we don't want the user to have to provide too many low level details of each step, since in many cases we can determine those from the original DSL program, and the absence of such details will not only make the decompositions easier to specify, but it will provide the sketch with a level of reusability across closely related algorithms.

To see how this works, we now restrict ourselves to the case of permutations, since they are the most expensive part of these bit manipulations. In fact, any bit matrix operation on a word can be thought of as a sequence of operations of the form $P_1 * X * P_2 * X \dots P_n * X$, where the P_i are permutations and the X simply take two words and apply the operator to them (either OR, XOR or AND). Out of these, the most expensive part, and the part where there will be the most potential for optimization will be the permutations.

5.1.1 Permutations

We use the term permutation, in a slightly more general way than it is commonly understood, to refer to leaf filters where the N_{out} bits they produce are simply a reordering of the N_{in} bits they take in, with some bits possibly missing, but no bits appearing more than once. More formally, in terms of Boolean matrices, a permutation will be a filter whose matrix has at most one non-zero entry per column and at most one non-zero entry per row.

The goal of the decomposition will be to transform the leaf filter into a pipeline of leaf filters, each of which corresponds to a step in the implementation. The user will provide the steps, and some constraints on what the system can or can not do with the bits on each step. The system must then search through the space of possible decompositions to find one that satisfies all the constraints.

This is more efficient than searching the space for an optimal decomposition, because we will be able to use the algebraic properties of the constraints to rule out large portions of the search space with a polynomial amount of work. Depending on the constraints, the space may still be too large for the system to search, but in that case, the user can be asked to add more details to the sketch in order to make the search space tractable.

To explain how this works, we introduce a bit of notation. Suppose that we are given N bits numbered 1 to N , and a specification of a reordering of the bits. Such reordering, or permutation, can be expressed as a vector of the form $\langle x_1, x_2, \dots, x_N \rangle$ where $x_i = P_i^d - P_i^s$ where P_i^d is the final position of bit i and P_i^s is the initial position of bit i . In this case, since the bits were labeled based on their initial positions, we have that $P_i^s = i$. Now, a de-

composition of this permutation into K steps can be expressed as a *decomposition vector* of the form

$$Y = \langle y_1^1, y_2^1, \dots, y_N^1, y_1^2, y_2^2, \dots, y_N^2, \dots, y_1^k, y_2^k, \dots, y_N^k \rangle$$

where $y_i^j = P_i^j - P_i^{j-1}$, where P_i^j is the position of bit i after step j . This notation allows us to treat the space of possible decompositions as a vector space, and this is what allow us to use the constraints to efficiently rule out entire subspaces from the vector space.

Now, for a given step k , the user can specify any combination of the following four different types of constraints.

1. `shift(b1, ... bM by j)` Shifts all the bits $\{b_1, \dots, b_M\}$ by j . It translates into M separate constraints on the decomposition vector. For each b_i , the constraint requires that $y_{b_i}^k = j$, where k is the step where the constraint appears.
2. `shift(b1, ... bM by ?)` Shifts all the bits $\{b_1, \dots, b_M\}$ by the same amount, but gives the system freedom to select the amount. This will translate into $M - 1$ linear constraints of the form $y_{b_i}^k = y_{b_{i+1}}^k$.
3. `pos(bi, p)` Guarantees that bit b_i will be in position p at step k , or in other words, that $P_{b_i}^k = p$. Since $y_i^j = P_i^j - P_i^{j-1}$, this constraint translates to

$$P_{b_i}^0 + \sum_{j=1}^k y_{b_i}^j = p.$$

4. `shift(b1, ... bM by a || b ...)` Shifts b_1, \dots, b_M by either a or b, or some other amount listed in the option list. They don't have to all move by the same amount. In this case, we have M constraints of the form $y_{b_i}^k \in \{a, b, \dots\}$. This is a non-linear constraint so it will be handled in a different way from all the previous ones.

To the constraints specified by the user, we add two constraints that are implicit from the fact that the decomposition has to be semantics preserving.

1. The final position of all bits must coincide with the final position of the bits from the original permutation. This constraint is just a special case of constraint 3 above, and is handled in the same way.
2. Two bits can not be in the same position in the same step. This is also a non-linear constraint, and will be handled together with constraint 4 above.

The constraints identified above as linear can be encoded as a matrix, and they can be solved in polynomial time using Gaussian elimination over the integers. The result will be a particular solution Z and a set of decomposition vectors of v_1, \dots, v_m such that any vector of the form

$$V = Z + \sum_{i=1}^m \alpha_i * v_i \quad (1)$$

satisfies the linear constraints. At this point, the challenge will be to find a set of α_i that make V satisfy the non-linear constraints as well.

Now, for the non-linear constraints, the strategy will be to create a set of linear constraints on the α_i in terms of symbolic values, and then to derive constraints for those symbolic values that are easy to check and that will guarantee that all the constraints, linear

and non-linear, are satisfied. I will now show in detail how this is done for the type 4 constraints, and then mention how the same approach is used also for the other set of constraints. If we consider V to be a fixed vector, then equation (1) can be thought of as a set of linear constraints on the α_i . Now, we don't know what V is, that's what we are trying to find out, but we do have some information on the entries of V given by the constraints of type 4. For example, if on step k we have a constraint of the form `shift(b1 by a || b)`, then we know $V_{b_1}^k = (a \text{ or } b)$. Now, putting those constraints in matrix form, and using only those constraints that include components of V for which we know something, we get a matrix equation like this:

$$\begin{bmatrix} (v_1)_{b_1}^{k_1} & (v_2)_{b_1}^{k_1} & \dots & (v_m)_{b_1}^{k_1} \\ (v_1)_{b_2}^{k_2} & \vdots & \vdots & (v_m)_{b_2}^{k_2} \\ \vdots & \vdots & \vdots & \vdots \\ (v_1)_{b_l}^{k_l} & (v_2)_{b_l}^{k_l} & \dots & (v_m)_{b_l}^{k_l} \end{bmatrix} * \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_m \end{bmatrix} = \begin{bmatrix} V_{b_1}^{k_1} - Z_{b_1}^{k_1} \\ V_{b_2}^{k_2} - Z_{b_2}^{k_2} \\ \vdots \\ V_{b_l}^{k_l} - Z_{b_l}^{k_l} \end{bmatrix} \quad (2)$$

After running Gaussian elimination on this system of equations, keeping the $V_{b_i}^{k_i}$ as symbolic variables, we will get a set of expressions of the form $\alpha_i = a_1 * V_{b_1}^{k_1} + \dots + a_l * V_{b_l}^{k_l}$, and an additional set of expressions of the form $c_1 * V_{b_1}^{k_1} + \dots + c_l * V_{b_l}^{k_l} = 0$. At this point, what we can do is search for values for the $V_{b_i}^{k_i}$, that satisfy the second type of expression, and then plug those values into the expressions of the first kind to find the value of α_i .

To handle the constraint that no two bits can be in the same position at the same time, our symbolic values will be $P_{b_i}^{k_i}$, the position of bit b_i at step k_i , which can also be expressed in terms of the α , and our resulting expressions after Gaussian elimination will look like $\alpha_i = a_1 * V_{b_1}^{k_1} + \dots + a_l * V_{b_l}^{k_l} + d_1 * P_{b_1}^{k_1} + \dots + d_m * P_{b_m}^{k_m}$, and $c_1 * V_{b_1}^{k_1} + \dots + c_l * V_{b_l}^{k_l} + e_1 * P_{b_1}^{k_1} + \dots + e_m * P_{b_m}^{k_m} = 0$, but in both cases, the search strategy will be the same.

5.2 Restructuring

Compared to the decompositions, the restructurings are quite simple. The user simply selects a pipeline or a splitjoin, and specifies a new structure for the pipeline. In particular, the user can reorder filters, hoist filters into our out of splitjoins, and coalesce many leaf filters into a single filter. Most of the algorithms to do the aforementioned transformations come directly from StreamIt. The only thing that is new in this case, is that we give the user control over how to do the reorganizations, instead of leaving it entirely up to the system.

6. EVALUATION

In this paper we have described StreamBit as a new methodology for productively implementing high-performance bit-streaming applications. In this section, we provide concrete results that quantify the productivity impact of programming in StreamBit, as well as the rewards on investment when users try to optimize the performance of their programs. Furthermore, we quantify the productivity versus performance rewards of sketching, compared to manually tuning programs written in C.

6.1 Methodology

We held a *user study* to evaluate the productivity and performance rewards attributed to StreamBit and sketching. The study invited participant 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

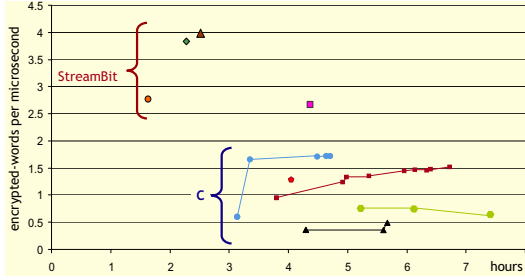


Figure 7: Performance and development time analysis from our user study. The four points in the top half of the graph represent the StreamBit users. The remaining points on the graph represent the data points collected from the C users.

operations and a non-linear transformation. The cipher is formally described by the following equations:

$$\begin{aligned} X_0 &= IP(M) \\ X_{i+1} &= X_i \oplus F(X_i, K_i) \\ F(X_i, K_i) &= L(K_i \oplus P_i(X_i)) \end{aligned}$$

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.

We recruited two groups of users: one set of users implemented the cipher in C and the other group implemented the cipher in StreamBit². 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 ciphers³. 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.

In Figure 7, 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 microsecond; better performance means more encrypted-words per microsecond. Each point on the graph represents the development time and performance of a single participant; the C users were allowed to optimize their implementations and therefore there are multiple points per participant that are connected by solid lines showing their progress over time. The graph is an intuitive representation of productivity versus performance, and it is readily apparent that the StreamBit participants spent between two and four hours implementing the cipher and achieved better performance compared to the C participants. The StreamBit ciphers were three times faster on average, with slightly faster development times compared to their C counterparts.

The C cipher implementations were compiled with gcc version 3.3 and optimization level `-O3 -unroll-all-loops`. The StreamBit compiler applied several transformations to the StreamBit code submitted by the user study participants, and generated low level C code that was subsequently also compiled with gcc and the same optimization flags. All of the resulting executables were run on an

²The 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.

³C and StreamBit users who did not complete the study chose to leave due to personal constraints.

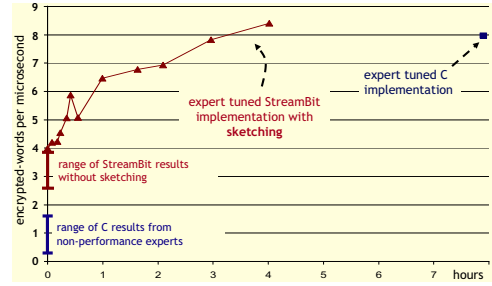


Figure 8: Performance and development time analysis from our user study. The four points in the top half of the graph represent the StreamBit users. The remaining points on the graph represent the data points collected from the C users.

Itanium2 processor.

In addition to the first-solution comparisons, we noted earlier that the C participants also manually tuned their implementations. They were given a short tutorial on well known (bit-level) optimizations, and allowed to apply any other optimization ideas they could think of. We did not restrict the number of submissions that a participant could make, and encouraged as many submissions as possible. The results from this phases of the study appear as connected data points in Figure 7 (one series per participant). All but one of the C participants tried to tune the performance of their ciphers. From the results, it is interesting to observe that at least one optimization lowered performance, quite a few did not have a measurable impact, and only one provided more than a modest reward. The results highlight the complexity of tuning bit-level applications, and demonstrate the challenge in understanding which optimizations pay off. The C participants spent between one and three hours optimizing their implementations, and while some improved their performance by 50% or more, they did not achieve the performance of the StreamBit implementations. 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 or at least poorly understood.

We went further in our user study, and assigned a *performance expert* to each of the two user groups. A C performance expert tuned a reference C implementation and a sketching expert began to sketch partial high-performance implementations for the cipher. The results from this phase of the user study are reported in Figure 8. The expert tuned C implementations was completed in just under eight hours, and achieved a performance of eight encrypted-words per microsecond. In contrast, the sketching expert evaluated over ten different implementations in four hours, and achieved slightly superior performance compared to the manually tuned C code. 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 Implementation of DES

The user study showed that given a comparable amount of time, it is easier to achieve high performance by using the StreamBit system, even for users who have never seen the language before. Just as important, however, is to compare how StreamBit generated code compares with heavily optimized ciphers in the public domain. In order to do this, we compare StreamBit generated code with a libDES, a widely used publicly available implementation of DES that is considered to be one of the fastest portable implementations of DES. LibDES combines extensive high level transformations that take strong advantage of Boolean algebra with careful low level coding to achieve very high performance on most platforms. Table 1 compares libDES across different platforms with three different implementations produced by StreamBit having different tradeoffs of memory vs. operations. There are several things worth noting. First, 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. 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. This was particularly noticeable in the PentiumIII architecture, but even then, we were able to beat libDES on at least one platform. Additionally, it is worth noticing that whereas the libDES code is extremely hard to read and understand, matches the FISP standard very closely. Finally, note that on all platforms, the transformations produce major performance improvements compared to what the naive algorithm delivers.

7. RELATED WORK

The issue of productivity has been a theme to varying degrees in much of the high performance literature. Optimizing compilers, for example, allow programmers to write code in a high level but general purpose language while the compiler takes care of transforming the program to run efficiently in the target machine. Modern compilers can perform a number of transformations aimed at eliminating redundant or unnecessary operations, improving memory access locality, and for exploiting particular machine features such as vector processors (see [3] for a very complete survey). Our approach, complements traditional compiler technology by allowing the users to specify transformations that are too specific to a particular problem to be worth including in a compiler. At the same time, by compiling into a mature language like C, we benefit from the optimizations performed by high end C compilers.

Domain specific languages have also been widely studied for their potential to improve productivity. There have been a number of recent efforts to improve the performance of domain specific languages. Padua et.al., for example, have made significant progress towards making the performance of MATLAB comparable to that of handwritten code through the use of aggressive type and maximum matrix size inference[1] [5].

Kennedy et. al. have worked to improve the performance of domain specific languages through the use of an approach called telescoping languages. The idea is to build libraries to provide component operations accessible from the domain specific language, and precompile several specialized versions of them tailored for different sets of conditions that may hold when the routine is invoked, therefore avoiding the expense of having to perform extensive interprocedural optimization at the time the program in the domain specific language is compiled, while still achieving good performance [8] [4].

The StreamIt language on which our abstraction is based, builds upon a large body of work on Synchronous Data Flow programming languages (see [11] for examples of this work). It's compiler automatically identifies linear filters, and performs many optimizations targeted towards DSP applications.

Aspect Oriented Programming 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" [9]. Our approach can be understood as a form of Aspect Oriented Programming, where the algorithm specification and the performance improving transformations are the two aspects we are dealing with. There have been other efforts at applying aspect oriented programming to restricted application domains, for example Irwin et. al. demonstrate the use of Aspect Oriented Programming in the domain of sparse matrix computations[7].

Finally, one of the most widely used methods for improving productivity in high performance computing is the use of libraries. Successful domain specific libraries in widespread use include BLAS [10] and LAPAC [2]. Code that uses the high performance libraries can be clean and fast, and composed relatively quickly, but within the library, the problems described above sometimes even become amplified, since the library must be performance tuned for lots of different platforms. More recently, a lot of work has been devoted to the development of self tuning libraries like FFTW [6], or SPIRAL [14], as well as runtime adaptive libraries like STAPL [13]. Our approach could provide an alternative for these approaches for cases when the narrow applicability of a library does not warrant the high development effort of a self tuning or runtime adaptive system.

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 bit-streaming. We believe that sketching is also applicable in other domains where a domain algebra for semantics-preserving restructuring of programs is available. Two attractive candidate domains are FFT computations and sparsifications of matrix computations.

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.

9. REFERENCES

- [1] G. Almasi and D. Padua. Majic: Compiling matlab for speed and responsiveness. In *Proceedings of ACM SIGPLAN*

	libDES	SBit A	SBit B	SBit C	NoOpt
375 MHz IBMSP	2.45	1.61 (65%)	1.96 (80%)	2.06 (84%)	0.5921(20%)
1.3GHz Itanium 2	6.41	5.92 (92%)	7.05 (110%)	4.35 (67%)	2.22(34%)
900MHz Sparc	3.94	2.4 (60%)	2.9 (73%)	2.59 (65%)	0.65 (16%)
699 MHz Pentium III	3.1	1.13 (36%)	1.05 (33%)	1.17 (37%)	0.3048 (9%)

Table 1: Comparison of 3 different implementations produced by StreamBit with libDES. Note that there is no *BEST* implementation, but StreamBit allowed us to try many different implementations and try them on different machine over the course of half a day.

- Conference on Programming Language Design and Implementation*, pages 294–303, June 2002.
- [2] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Proceedings, Supercomputing '90*, pages 2–11. IEEE Computer Society Press, 1990.
- [3] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.
- [4] 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.
- [5] L. DeRose and D. Padua. A matlab to fortran 90 translator and its effectiveness. In *Proceedings of the 10th ACM International Conference on Supercomputing - ICS'96, Philadelphia, PA*, pages 309–316, May 1996.
- [6] M. Frigo and S. Johnson. Fftw: An adaptive software architecture for the fft. In *ICASSP conference proceedings*, volume 3, pages 1381–1384, 1998.
- [7] 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.
- [8] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, December 2001.
- [9] 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.
- [10] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, Sept. 1979.
- [11] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, September 1987.
- [12] M. Morgan. <http://www.schneier.com/blowfish-bug.txt>.
- [13] A. Ping, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In *Int. Workshop on Languages and Compilers for Parallel Computing*, August 2001.
- [14] 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.
- [15] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, Apr. 2002.
- [16] 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) Computer Architecture News*, Goteborg, Sweden, June-July 2001. ACM SIGARCH / IEEE. Published as 28th Annual International Symposium on Computer Architecture (28th ISCA 2001) Computer Architecture News, volume 29, number 7.

APPENDIX

Due to space limitations, we removed from the submission the definition of the sketching language (which is used in Figure 6), and a complete example of how sketching interacts with the naive compilation to produce the desired implementation. These will be provided in the full paper, along with more details about our experimental evaluation.