Bit-Packing Optimization for StreamIt

Kunal Agrawal
Singapore MIT Alliance

Saman Amarasinghe
Singapore MIT Alliance and
Laboratory of Computer Science
Massachusetts Institute of Technology

Wong Weng Fai
Singapore MIT Alliance and
Department of Computer Science
National University of Singapore

Abstract— **StreamIt is a language specifically designed for modern streaming applications. A certain important class of these applications operates on streams of bits. This paper presents the motivation for a bit-packing optimization to be implemented in the StreamIt compiler for the RAW Architecture. This technique aims to pack bits into integers so that operations can be performed on multiple bits at once thus increasing the performance of these applications considerably. This paper gives some simple example applications to illustrate the various conditions where this technique can be applied and also analyses some of its limitations.**

***Index Terms*— Bit-packing, optimization, Streaming Applications, StreamIt.**

I. INTRODUCTION

Streaming applications are a new and distinct domain of programs that are becoming increasingly important and widespread. StreamIt is a language and compiler designed to make programming in this domain easy and efficient [3], [4]. In addition the StreamIt compiler aims to use the structure of the programs to generate efficient code for this class if applications. Streaming applications include the domains digital signal processing, embedded applications and many others. A subset of these applications like programs for digital circuit design and simulation operate on streams of bits. For this purpose StreaMIT has a data type defined as bits. But typically microprocessors do not operate on individual bits and so these bits are represented as bytes/integers in the hardware. Thus operations on a single bit are as expensive as operations on an entire integer which makes this implementation very inefficient.

But an important observation is that in streaming applications typically same operations are performed repeatedly on all the data in the stream. So in most cases it might be possible to operate on multiple bits at the same time. Taking advantage of this observation we propose and motivate an optimization technique wherein we put multiple bits in one integer and perform operations on all these bits at once. This could significantly increase the performance in many cases.

This paper is organized as follows. In section 2 we give some background on the StreamIt language and syntax. In section 3 we give the motivation for this optimization. We present the various ways of bit packing in section 4 with examples. Section 5 contains the conditions when this optimization cannot be applied and Section 6 has the conclusions.

Manuscript received November, 2002 This work was supported by Singapore MIT Alliance.
Kunal Agrawal(email: kunal@comp.nus.edu.sg)
Saman Amarasinghe (email: saman@lcs.mit.edu)
Wong Weng Fai(email: wongwf@comp.nus.edu.sg)

II. BACKGROUND

A. StreamIt Language Overview

The StreamIt language provides novel high-level representations to improve programmer productivity and program robustness within the streaming domain. The programmer constructs a stream graph containing blocks with single input and single output and describes the function of atomic blocks and structure of composite blocks. The compiler generates code for each block and applies optimizations to the stream graph to produce efficient code for the target architecture.

Computation in StreamIt is performed within stream objects. Every stream object has an input type and an output data type; the object is connected to ``tapes" of a hypothetically infinite number of homogeneous data objects.

The various stream objects are filters, pipelines, splitjoins and feedbackloops. Filters are atomic and have an initialization code and a steady state work code. Pipelines, feedbackloops and splitjoins are all composite structures that include some number of stream objects as children. Each structure specifies a pre-defined way of connecting streams into a single-input, single-output block.

1) Filters: The basic unit of computation in StreamIt is the Filter. All computation in StreamIt takes place within filters. The central aspect of a filter is the work function, which describes the filter's most fine grained execution step in the steady state. Within the work function, a filter can communicate with neighboring blocks using the tapes described above. These high-volume channels support the three intuitive operations: a) pop() removes an item from the

end of the channel and returns its value, b) peek(i) returns the value of the item i spaces from the end of the channel without removing it, and c) push(x) writes x to the front of the channel.

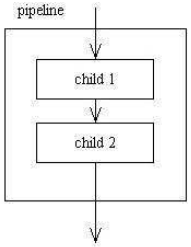
Filters must explicitly declare their initialization code if present, but also have the option of omitting it. A filter may have a single anonymous work function, or multiple named work functions. Each work function must declare its I/O rates, the number of items one call to the function removes, examines, or places on its tape. A basic filter declaration looks like the code shown in **Figure 1.**

```
int->int filter IntAvgFilter {
     init {
          // empty
     }
     work pop 1 peek 2 push 1 {
          push(peek(0) + peek(1));
          pop();
     }
}
```

Figure 1 : Filter Code

One important point of note is that StreamIt requires filters to have static input and output rates. That is, the number of items peeked, popped, and pushed by each filter must be constant from one invocation of the work function to the next.

2) Pipeline: The simplest composite filter is a pipeline. A pipeline contains a number of child streams; the output of the first stream is connected to the input of the second, whose output is connected to the input of the third, and so on as shown in **Figure 2:** Pipeline. All the streams are added to the pipeline using the add statement. Note that output type of the first stream should be the same as the input type of the second stream and so on. In addition the input of the first stream should be the input type of the pipeline and the output type of the last stream should be the output type of the pipeline.

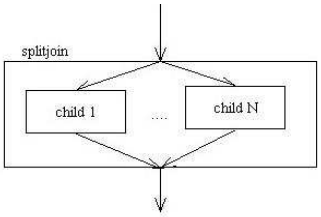



```
float->complex pipeline FloatToComplexAdd{
     add ReImToComplex();
     add ComplexPairAdd();
}
```

Figure 2: Pipeline

The body of the pipeline declaration is the initialization code; no internal function declarations or message handlers are allowed.

3)Split-Joins:A split-join allows computation to be run in parallel, possibly using different parts of the input stream. It is so named because incoming data passes through a splitter, is distributed to the child streams for processing, and then is fed through a joiner to be combined into a single output stream as shown in **Figure 3:** SplitJoin**.**



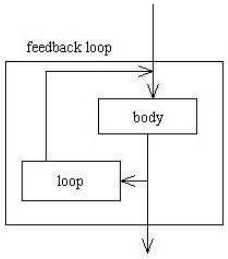

```
int->int splitjoin AddAndSub {
     split duplicate;
     add IntAdder();
     add IntSubtractor();
     join roundrobin;
}
```

Figure 3: SplitJoin

The splitter specifies how items from the input of the SplitJoin are distributed to the parallel components. The type of splitters allowed are: a) Duplicate, which replicates each data item and sends a copy to each parallel stream, b) RoundRobin(i1, i2, : : :, ik), which sends the first i1 data items to the stream that was added first, the next i2 data items to the stream that was added second, and so on, If the weights are ommitted from a RoundRobin, then they are assumed to be equal to one for each stream. Note that RoundRobin can function as an exclusive selector if one or more of the weights are zero.

Likewise, the joiner is used to indicate how the outputs of the parallel streams should be interleaved on the output channel of the SplitJoin. The only joiner allowed is RoundRobin, whose function is analogous to a RoundRobin splitter.

3)Feedback Loops: The FeedbackLoop construct provides a way to create cycles in the stream graph. A feedback loop has a body stream. Its output passes through a splitter; one branch of the splitter leaves the loop, and the other goes to the loop stream as shown in **Figure 4:** Feedback Loop. The output of the loop stream and the loop input then go through a joiner to the body's input.




```
float->float feedbackloop Feedback(){
       join roundrobin;
       body FloatAdder();
       loop FloatScaler();
       split duplicate;
       enqueue 0.0;
}
```

Figure 4: Feedback Loop

Enqueue statement takes a value and places it in FIFO order on the output tape from the loop stream. We need to enqueue enough items that the input joiner will be able to fire once; enqueuing more items causes a delay in the feedback loop.

B. StreamIt Compiler for RAW

Though StreamIt is architecture independent, it's structure is especially suitable for compilation to grid based architedture

with software exposed communication [1], [2]. The StreamIt compiler being developed at MIT Laboratory of Computer Science targets the RAW Architecture [5]. RAW stands for raw Architecture Workstation. It is composed of interconnected tiles each tile comprising of ALU, FPU, registers, data memory, programmable switch etc. The communication between these tiles is fully exposed to the software. As we have seen, StreamIt is composed of individual filters with fixed communication patterns. Thus these filters can be assigned to different tiles which perform communication in parallel. Since StreamIt syntax exposes the complete structure of the stream graph to the compiler, the compiler can work out the best layout at compile time, thus improving the performance.

III. MOTIVATION

A certain class of streaming applications has bits as the data on the stream tapes. There is a bit datatype defined in the streamit language definition. But in general, a microprocessor computes on bytes or words of data and not individual bits. Thus these bits are represented as integers after compilation. Eg. bit c = 1&0 is converted to int c = 00…000 & 00…001

This seems to be a big waste of computation resources. Also the StreamIt compiler targets the RAW architecture which, as explained above has interconnected tiles. So to send an entire integer for one bit is a waste of communication bandwidth.

Thus we propose this bit packing optimization. The idea is very simple. We try to put many bits into an integer instead of just one and then perform computation on this integer once effectively performing that operation on all those bits. Thus it is obviously necessary that the same operation is required to be performed on all these bits. This idea is especially feasible for StreamIt because typically the filter performs the same computation on all the data items on the incoming tape before putting them on the outgoing tape. This also addresses the communication problem since if we manage to pack an entire application or even many communicating filters then we can send the bit packed integers instead of individual bits thus decreasing the communication requirements.

IV. BIT PACKING

As explained above the idea of the optimization is to put multiple bits into an integer and perform operations on them as if they were one unit. Thus the programmer writes the program in terms of filters and other constructs that operate on bits. The compiler should automatically analyze where packing is possible and convert the program accordingly. There are two aspects to this conversion.

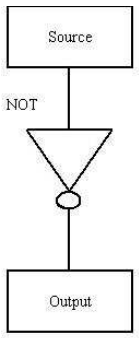
- Converting the filter so that it operates on integers instead of bits. For this we first need to analyze the filter to see if it can be packed (the conditions when packing is impossible are described in section 5). And if so how it should be packed.
- Packing and unpacking the bits. Since the input to the filter is originally bits we need to put a certain number of bits into a single integer. This effectively means adding another filter that will do the packing. Also it might not be possible to pack the next filter in the stream. Thus we have to again unpack the integer and send a stream of bits to the next filter.

One of the important decisions to be made is how many bits should be packed into an integer. If we put more bits into an integer, we should theoretically get a higher performance advantage but the opportunities for packing may be fewer. Also trying to pack too many bits could cause some filters to remain idle while they are waiting for enough bits to arrive for packing. This may cause a drop in performance. Currently we have chosen to pack 8 bits in one integer. This strategy may be re-evaluated later.

In the following sections we will use some simple example programs to illustrate some of the issues involved in bit packing.

A. A single NOT Gate

The simplest candidate filter for packing would be a `1->1` filter. That's a filter that pops one bit from the input tape and pushes one bit onto its output tape. Such filter can be easily and intuitively packed (assuming it satisfies all other conditions described in Section 5.). This is illustrated in the example application of NOT gate in **Figure 5:** Example 1.

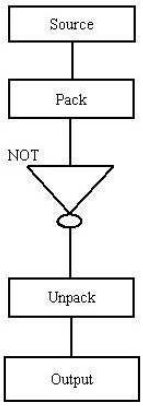



```
void ->void pipeline Gate(){
      add Source();
      add NOT();
      add Result();
}

void -> bit filter Source(){
      //to read in the input data
}

bit ->bit filter NOT(){
      work pop 1 push 1{
            push(pop()^1);
      }
}

bit -> void filter Output(){
      //to take the results and output it.
}
```

Figure 5: Example 1, Unpacked NOT

Note that top level stream in StreamIt has to be void->void type, and the source of input data and the sink of output data also have to be filters.

It is fairly easy to pack this application. The compiler has to just insert a packing filter before the NOT filter and make changes to the NOT filter so it will work correctly on integers. Also since the user probably wants the output also in terms of bits, we unpack before the output. The changes are illustrated in **Figure 6:** Example 1, Packed NOT.




```
bit -> int filter Pack(){
  work pop 8 push 1{
       int out =0;
       int mult = 1;
       for (int i=0;i<8;i++){
            out = out + pop()*mult;
            mult=mult*2;
       }
   push(out);

   }
}
bit ->bit filter NOT(){
     work pop 1 push 1{
          push(pop()^256);
     }
}
int -> bit filter unPack(){
   work pop 1 push 8{
        int in=pop();
        for (int i=0;i<8;i++){
             push(in%2);
             in = in/2;
        }
   }
}
```

Figure 6: Example 1, Packed NOT

B. *A One Bit Adder*:

From the previous example we see that it is fairly simple to pack a 1->1 filter. But that will not give performance advantages in most applications. But filters with different pop/push rates can also be packed. Consider the circuit for a OneBitAdder shown in **Figure 7: One Bit Adder Circuit Diagram.**

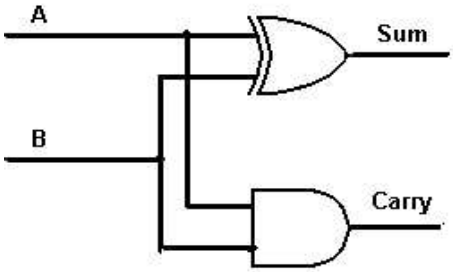
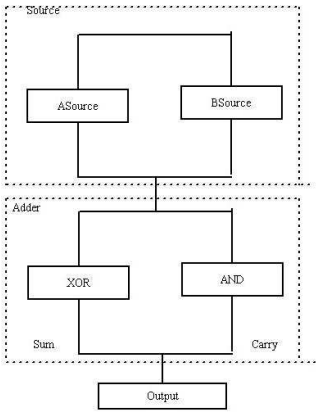


Figure 7: One Bit Adder Circuit Diagram

It can be represented in StreamIt format in **Figure 7: One Bit Adder Circuit Diagram**8. Note that the output bits are alternate sum and carry bits. If we want to saperate them, we have to add another roundrobin split join in the output filter.

This application is not as straight forward as the previous one. We can not just pack consecutive bits on the tape going into the filter, since the filter operates on two bits at once. But if we pop 16 bits and push 2 integers packing alternate bits on the two integers, we still get a good packing and then the AND and XOR gate filters can be transformed very easily. This packing is shown in **Figure 8: OneBitAdder in StreamIt.**




```
void -> void pipeline OneBitAdder(){
        add Source();
        add Adder();
        add Output();
}

void -> bit splitjoin Source(){
        split duplicate;
        add ASource();
        add BSource();
        join roundrobin();
}

bit -> bit filter XOR(){
        work push 1 pop 2{
                bit in1 = pop();
                bit in2 = pop();
                bit out = in1^in2;
                push(out);
        }
}

bit -> bit filter AND(){
        work push 1 pop 2{
                bit in1 = pop();
                bit in2 = pop();
                bit out = in1&in2;
                push(out);
        }
}

bit -> bit splitjoin Adder(){
        split duplicate;
        add XOR();
        add AND();
        join roundrobin();
}
```

Figure 8: OneBitAdder in StreamIt

Notice that we can pack before the split join because both AND and XOR behave in the same way as regards to their input output rates. If this were not the case, then it would be necessary to insert separate packing filters on each branch of the split join.

C. *A Full Adder:*

This final example illustrates one of the limitations of this technique and shows a condition where this technique cannot be used.

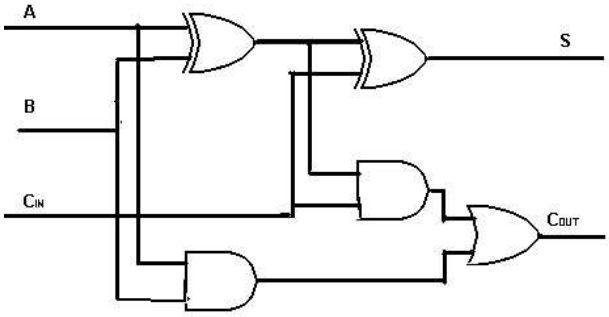


Figure 9: Full Adder Circuit Diagram

Figure 9: Full Adder Circuit Diagram shows the logic diagram of a full Adder circuit. The Cout can be fedback to the Cin so as to add multiple bits. Notice that the first part of the circuit is the same as the OneBitAdder. The StreamIt representation is shown in **Figure 10:** Full Adder and the streamIt code is in **Figure 11:** FullAdder in StreamIt

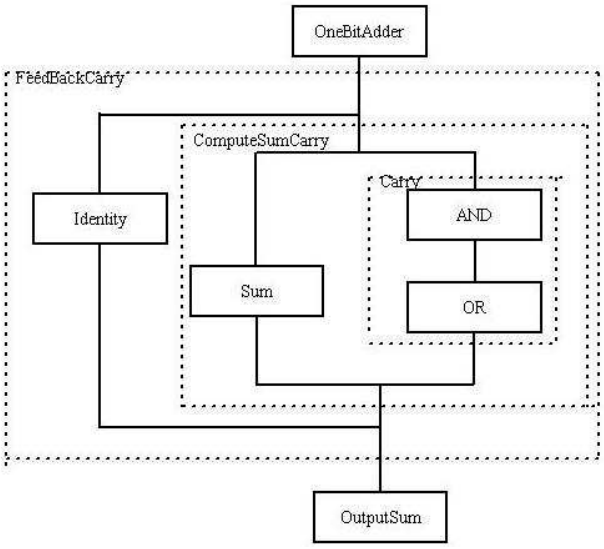


Figure 10: Full Adder

Here we can use the packed or unpacked OneBitAdder. But the rest of the application, the CarryFeedBack and the filters inside it cannot be packed. This is because every input depends on the previous output and thus we do not have enough inputs at the beginning of the filter available for packing. This is one of the conditions when this technique of bit packing is not possible. The other deterrents to bit packing are explained in the next section.

V. ISSUES WITH BIT-PACKING

There are certain types of filters and arrangements of filters for which it is not possible to perform this kind of bit packing. These are discussed briefly in this section.

A. Part of feedbackloop

As we already illustrated in the previous section, the filters which are part of feedbackloop cannot be packed as there are not enough inputs in advance to pack. There are exceptions to this case when there is an 8-bit slack in the feedbackloop.

B. Control flow changes from one input to another

If there are control flow structures in the filter work function like if-then-else, the filter might behave differently from one input to another. This might be in two circumstances

- There is data-dependent control flow, how the filter behaves depends on the incoming bit.
- There is local state in the filter which determines the control flow.

In both these cases, it is not possible to use this optimization since the basic premise is that the same operation is performed on all the packed bits.

```
void -> void pipeline FullAdder(){
     add OneBitAdder; //as above without the Output
     add FeedBackCarry();
     add OutputSum();
}

bit  -> void feedbackloop FeedBackCarry(){
     join roundrobin(2,1);
     body ComputeSumCarry();
     split roundrobin(1,1);
     loop Identity<bit>;
     enqueue (0);
}

bit -> bit splitjoin ComputeSumCarry(){
     split duplicate();
     add Sum();
     add Carry();
     join roundrobin(1,1);
}

bit -> bit pipeline Carry(){
     add AND();
     add OR();
}

bit -> bit filter Sum(){
     work pop 3 push 1{
          bit in1;
          bit in2;
          bit in3;
          in1 = pop();
          in2 = pop();
          in3 = pop();
          push (in1 ^ in3);
     }
}

bit -> bit filter AND(){
     work pop 3 push 2{
          bit in1;
          bit in2;
          bit in3;
          in1 = pop();
          in2 = pop();
          in3= pop();
          push (in1 & in3);
          push (in2);
     }
}

bit -> bit filter OR(){
     work pop 2 push 1{
          bit in1;
          bit in2;
          in1 = pop();
          in2 = pop();
          push (in1 | in2);
     }
}
```

Figure 11: FullAdder in StreamIt

C. More than one work function

If there is more than one work function in the filter and they have different push/pop rates, it is difficult to determine how to pack the bits so all of them can operate on the packed bits.

VI. CONCLUSIONS

We have presented the motivation and the idea for a bit-packing optimization technique which might be very useful for certain types of streaming applications. We believe that it is possible to considerably increase the performance of these applications by using this technique. We have started implementing this optimization in the StreamIt compiler. The compiler support for packing for 1->1 filter is almost complete. The other cases require detailed analysis of the filters for automatic packing and the implementation for this will be started in near future.

REFERENCES

[1] M. Gordon, W. Thies, M. Karczmarek, J. Lin, A. Meli, A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, **A Stream Compiler for Communication-Exposed Architectures**, *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October, 2002.

[2] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, **A Common Machine Language for Grid-Based Architectures**, In *ACM SIGARCH Computer Architecture News*, June, 2002

[3] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffmann, M. Brown, and S. Amarasinghe, **StreamIt: A Compiler for Streaming Applications**, *MIT LCS Technical Memo LCS-TM-622*, Cambridge, MA, December, 2001

[4] W. Thies, M. Karczmarek, and S. Amarasinghe, **StreamIt: A Language for Streaming Applications**, *2002 International Conference on Compiler Construction, Grenoble, France*. To appear in the Springer-Verlag Lecture Notes on Computer Science

[5] E. Waingold, M. Taylor, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, S. Devabhaktuni, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, Baring it all to Software: The Raw Machine, MIT/LCS Technical Report TR-709, March 1997.