Abstract

New parallel architectures are emerging to meet the increased computational demands of streaming applications. This creates a need for high-level, architecture-independent languages. One such language is StreamIt, designed around the notions of streams and stream transformers, which allows efficient mapping to a variety of architectures.

This paper presents our approach of compiling StreamIt applications to the XPP reconfigurable array architecture. We focus mainly on the compiler back end. Although StreamIt exposes the parallelism in the stream program, still a thorough analysis is needed to adapt it to the target architecture. A code generator has been designed for the XPP. It has been demonstrated that by applying optimizations, performance comparable to the low level NML implementation can be achieved. Moreover, the construction of the compiler makes it possible to port StreamIt applications to multiprocessor architectures by doing some architecture specific modifications in the back end.

1 Introduction and Motivation

Media applications, such as voice and video processing as in HDTV or baseband processing in cell phone base stations, are becoming increasingly complex and consume more and more computing power.

Programming these streaming applications faces the difficulty of making full use of the computing power available by new increasingly complex parallel architectures. New grid-based hardware technologies are emerging, which conventional programming languages are not well suited for. The compilers for conventional programming languages such as C or Java generate trees that are difficult to map onto parallel architectures. The programming language StreamIt is designed for applications with chains of data to be processed in different ways. The compiler for StreamIt produces stream graphs [1] that can easily be mapped onto parallel architectures such as grid-based architectures.

The goal of the StreamIt project at MIT is to provide the language and compilation infrastructure required by the Reconfigurable Architecture Workstation (RAW) project. RAW is a scalable processor architecture suitable for applications in which the compiler can extract and statically schedule fine-grain parallelism. It consists of 16 tile processors arranged in a 2D-mesh, each with a programmable switch for communication between tiles [2].

The eXtreme Processing Platform (XPP) from PACT XPP Technologies, with its packet oriented communication mechanism is very suitable for compute intensive streaming applications. The PACT XPP64A-1 reconfigurable processor core consists of an 8 x 8 array of ALU-PAEs (Processing Array Elements) with 2 rows of RAM-PAEs at the edges [3]. The software development tools for XPP consist of Vectorizing-C compiler for compiling C code to Native Mapping Language (NML) code, that can be compiled by XMAP mapper to generate binaries for XPP core or for simulation by XSIM simulator [4].

In this paper we present a method to compile applications developed in StreamIt to XPP. The primary focus is laid on issues related to the compiler back end and its interface to the preprocessor. It is the back end of the compiler that translates the graph representation of the
parallel program produced by the compiler front end into machine code.

The rest of the paper is organized as follows; Section 2 presents salient features of parallelizing compilers. Section 3 provides an overview of the StreamIt language. Section 4 describes the StreamIt compiler and its generated code. Section 5 examines the proposed StreamIt compilation steps for XPP. The results of FIR and FFT implementations are discussed in Section 6, and the conclusions of the study are presented in Section 7.

2 Parallelizing Compilers

For parallel architectures the compilers must be able to detect and utilize parallelism. Mapping computations onto various processing array elements and optimization of different distributions are dealt with by the compiler back end tool chain. Compared to traditional compilers, a significant feature of parallelizing compilers is instruction scheduling. The phase of mapping a parallel program to a parallel architecture starts without explicit communication and produces a program with explicit communication operations. The communication scheduling starts with data distribution and generates a code for whatever communication is required to adhere to the chosen distribution.

3 StreamIt Language

The StreamIt language is an implementation of the stream-programming model, which constructs a stream graph, containing stream transformers with a single input and a single output to enable efficient development of stream programs.

StreamIt [5] has five data types and four primitives for building stream transformers. The StreamIt primitives, i.e., filter, pipeline, split-join and feedback loop, have one input and one output stream. Filters are used for building atomic stream transformers while pipelines, split-joins and feedback loops are used for combining stream transformers as shown in Figure 1.

StreamIt, which is a portable language for compute-intensive signal processing applications, allows the compiler to map a single source program on a variety of grid-based architectures by taking away the variations and retaining the similarities [6]. The ability of the compiler to reconstruct the stream graphs enables it to combine adjacent stream transformers or split computationally intensive transformers into multiple parts to have a more parallel computation. The StreamIt compiler optimizes the stream graph to produce an efficient code, which allows the implementation of high-performance applications for parallel systems without detailed knowledge of the target architecture.

![Figure 1 StreamIt combinator](image)

*Figure 1* StreamIt combinator: (a) Pipeline, (b) Split-join, (c) Feedback loop.
4 StreamIt Compiler

The StreamIt compiler front end converts the application code to legal Java code, which is then used to produce Kopi intermediate representation (IR). The Kopi IR is translated to StreamIt high IR (SIR) consisting of a stream graph of filters, pipelines, split-joins and feedback loops [2]. The structured stream graph produced by the compiler front end is converted into an unstructured flat graph, consisting of nodes representing filters, splitters and joiners. In order to generate the SIR code, each node in the flat graph is visited in data-flow order. The entire flat graph of the application is visited twice, first for the initialization stage and then for the steady-state stage. The scheduler in the back end simulates the execution of the stream graph and determines steady-state execution timings of the filters.

4.1 StreamIt Code generation

Before generating the filter’s code, there are certain optimizations such as loop unrolling, destroying field-arrays and constant propagation performed by the compiler. The generated code of the entire application is nested inside the main function, only the helper functions declared by the filters and the code for the initialization stage appear outside main. The resulting code is organized as follows:

- Definitions of the variables by each of the filters in the application.
- Definitions of the buffers and buffer index variables that are used to pass the data between different filters.
- The code for the steady-state schedule of the complete application is enclosed in a while loop.
- A block containing init function calls for each of the filter.
- The code for the initialization schedule of the stream graph.

In each stage, the nodes are placed in the order based on the data-flow dependencies of the stream graph. The nodes communicate with their neighbors by accessing buffers. All the push, pop and peek expressions are translated to these buffer accesses. Now the code generation of StreamIt primitives is described.

Filter

The filter translation generates separate code for the initialization stage and for the steady-state stage of the filter. The code for each of these stages is enclosed in a loop, which runs a number of times that is equal to the multiplicity of the current stage. The filter translation also includes the conversion of all the pipeline communication expressions to buffer accesses. The number of items added to the outgoing tape by the filter is equal to the multiplicity of the current stage multiplied by the push rate. The size of the incoming buffer is calculated as the maximum of the initialization or steady-state stage multiplicity added to the number of items remaining on the input tape after the initialization stage of the filter.

Pipeline

Pipelines, which are used to connect StreamIt stream transformers in a chain fashion, do not have an init or work function, because a pipeline itself does not perform any computation, thus pipelines are translated simply to buffers.

Duplicate Splitter

The code generated for the duplicate splitter consists of a loop that runs $M$ times, where $M$ is the multiplicity of the splitter, and duplicates the data elements from incoming buffer to $n$ outgoing buffers. The number of pushed items on each of the outgoing tapes equals $M$.

Round-robin Splitter

The code generated for the round-robin splitter consists of a loop that runs $M$ times and in addition there are loops for each of the splitter branches which copy $W_i$ data elements from the incoming buffer to $n$ outgoing buffers, where $W_i$ is the weight of each outgoing buffer.

Joiner

The code generated for the joiner consists of a loop that runs $M$ times, and there are loops for each of the terminating branches which copy $W_i$
data elements from \( n \) incoming buffers to the outgoing buffer, where \( W_i \) is the weight of each incoming buffer. The number of pushed items on each of the outgoing tapes is equal to the product of \( M \) and \( W_i \). The number of pushed items on the single outgoing tape equals to the product of \( M \) and \( W_{tot} \), where \( W_{tot} \) is the sum of all the weights of the joiner.

**Feedback loop**
Feedback loops do not appear in the unstructured flat graph. Each feedback loop is broken down into a joiner, a body stream, a loop stream and a splitter and then the translation of these individual objects is performed. The use of `enqueue` statement in the initialization function of the feedback loop causes the specified items to be placed in the buffer representing the delay path before the execution of the initialization stage [5]. These items are taken into account when calculating the buffer sizes.

### 4.2 Optimizations
The StreamIt compiler also includes a number of high-level optimizations that can be applied to improve the performance of the produced code. The unroll loop optimization is used to unroll the for-loops generated during the translation of different StreamIt primitives. The compiler also takes a parameter to determine the unroll factor. The higher the unroll factor is, the more aggressive are the optimizations performed.

The destroy field arrays optimization is applied together with the unroll loops optimization to decompose the buffers created during the translation of filters, pipelines, feedback loops and split-joins into local variables.

The StreamIt compiler also provides an interface to the programmer to manually guide the optimizations performed by the compiler on a particular application. This is done by obtaining the numbers assigned to the different StreamIt primitives used in the application by looking at the stream dot graph produced during compilation. These are used by another pass to provide optimization specifications to the compiler.

### 5 StreamIt Compilation Methodology for XPP
The complete compilation process is divided into three steps, starting from the StreamIt code and resulting into the binaries for configuration of the XPP core as shown in Figure 2.
5.1 Code generation from StreamIt compiler back end

The compilation process starts with reading an application source program written in StreamIt as shown in Figure 2. The source file is then parsed by Kopi front end for syntax and semantic analysis. Then the Kopi IR is translated to SIR code. Operations of constant propagation and loop unrolling have been performed on SIR code to convert the parameterized structure of IR into a static graph. The SIR code is then passed through the RAW back end with the stand-alone option to generate the C-code consisting of a single file that does not depend on a runtime library. The back end converts the object-oriented style of StreamIt filters into procedural code and schedules the execution of filters. In other words, the entire stream execution is inlined into a single function. The optimizations described in Section 4.2 are also applied in this step to optimize the generated code.

The objective for producing an executable C-code of the application at this stage of compilation is that this functional code can then be used for range estimation of the floating point numbers used in the application using real inputs. The results of the range estimation are used to determine the fixed-point representation for the floating-point numbers. The need for fixed-point representation arises, as floating-point numbers are not supported in XPP.

5.2 C-Code Transformations for XPP

The generated code is transformed to make it compatible with the XPP Vectorizing-C compiler during this step in the compilation. The following transformations are performed:

- Inclusion of XPP header files.
- Removal of pointer references.
- Conversion of floating-point constants to fixed-point representation.
- Unrolling of StreamIt filter's `init` function code to the `main` function.
- Insertion of XPP port functions for I/O operations instead of source and print statements.
- Insertion of NML module instantiation instructions for integer division operator and fixed-point operators for multiplication and division.

A program has been implemented to perform these transformations automatically. The application consists of a number of passes as shown in Figure 3.

![Figure 3 Block Diagram of C-Code Transformation.](image-url)

The code transformation application inserts the XPP header files needed for compilation in the first pass. Then the declarations of variables and arrays used in the application to be compiled are visited and a linked list is created. The conversion of pointer references in the code is also done in this pass. For each of the declared variables, the linked list consists of the type, scope, name and initialization value of the variable. The type of the variable identifies whether the variable is integer, float, integer-array or float-array. The scope of the variable differentiates between local and global variables. The name identifies the variable itself and the initialization value initializes the variable. For float type variables, the initialization values are converted into the equivalent fixed-point numbers.
After collecting the information about all the variables, the arithmetic operators used in the code are searched in the next step. In order to gather the operator’s information, the complete code is parsed expression by expression, and then, based on that information, the NML module declaration pass inserts the relevant module declarations. Then the code for the init functions is unrolled to the main function and finally the application code to be compiled is visited and instantiation instructions for NML modules are inserted. The I/O operation instructions are also inserted in the same pass.

**Fixed-point representation**

The StreamIt language supports floating-point numbers, which are not supported by XPP. Thus the floating-point numbers are represented as fixed-point numbers. The maximum word length available for fixed-point representation in XPP is 32-bits. Based on the dynamic range of the floating-point variables, the Q16.16 format is used to represent fixed-point numbers for the applications presented in this paper. Thus, the maximum and minimum ranges of the integer part of a variable are 32767 and –32768 respectively and the resolution of the fractional part is 0.00001. The supported fixed-point operations are add, subtract, assignment, multiply and divide.

**5.3 Generation of XPP configurations**

The transformed code is then compiled with the XPP-VC compiler to generate the NML code of the application. The generated NML code together with the library of NML modules is used by the XMAP tool to generate binaries for configuring the XPP core. These binaries can also be used for simulation and debugging purposes.

### 6 Implementation Results

**6.1 Performance Comparison of two Integer FIR implementations**

In this sub-section, the performance results of two integer FIR implementations on XPP are presented. The fine-grained 8-tap FIR filter implementation consists of a series of filters, one per tap, to expose more parallelism to the compiler, which generates a number of for-loops and arrays depending upon the number of filters.

<table>
<thead>
<tr>
<th>Event REGs</th>
<th>Data REGs</th>
<th>ALUs</th>
<th>BREG ALUs</th>
<th>FREG ALUs</th>
<th>Configuration Cycles</th>
<th>Simulation Cycles 128 samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-Tap FIR (StreamIt)</td>
<td>153</td>
<td>0</td>
<td>54</td>
<td>19</td>
<td>106</td>
<td>-</td>
</tr>
<tr>
<td>10-Tap FIR (StreamIt)</td>
<td>49</td>
<td>0</td>
<td>16</td>
<td>8</td>
<td>17</td>
<td>3586</td>
</tr>
<tr>
<td>8-Tap FIR Optimized (StreamIt)</td>
<td>0</td>
<td>7</td>
<td>11</td>
<td>4</td>
<td>0</td>
<td>1955</td>
</tr>
<tr>
<td>10-Tap FIR Optimized (StreamIt)</td>
<td>64</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>22</td>
<td>3735</td>
</tr>
<tr>
<td>8-Tap FIR (NML)</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>1950</td>
</tr>
</tbody>
</table>

These arrays have been used in the generated code to buffer inputs and outputs of each filter. These multiple array accesses generate lots of extra overhead control, i.e. event-registers and forward-registers (FREG) in NML, as evident from Table 1. The event, forward and back-registers (BREG) are used to control the data flow. Thus, the implementation exceeds the resources available on the XPP64A-1 device.

The coarse-grained 10-tap FIR implementation consists of a DelayMany filter that pushes zeroes depending upon the filter taps on the tape
and the FilterKernel that computes the FIR output. The FilterKernel operates in a windowing fashion and moves the coefficient window over the pipeline, multiplies the coefficients and accumulates the result. The accumulated result is finally pushed on the tape. The coarse-grained FIR implementation consists of fewer filters, thus less array accesses are required, and consequently the implementation requires fewer resources when mapped to XPP.

In order to minimize the resource usage, the for-loops and arrays produced by the StreamIt code generator in C code should be avoided. All the array accesses are eliminated in favor of local variables by using destroyfieldarray and unroll optimizations. The destroyfieldarray option attempts to decompose buffers into individual local variables and the unroll option performs unrolling of the for-loops. The optimized code generated by the StreamIt compiler still contains dead code, which is then eliminated by the XPP-VC compiler. In fact, the optimized code results in as efficient use of resources as the same application implemented directly in NML.

From the results showing the number of simulation cycles, it can be deduced that both the fine-grained FIR and FIR in NML extract more optimization; whereas there is no significant improvement achieved in coarse-grained FIR implementation, because there is one array buffer declared within the DelayMany filter which cannot be decomposed by the compiler. The sequential iteration of this buffer causes a delay of 16 cycles per output sample as compared to a single cycle per output sample for the fine-grained FIR and the FIR implementation in NML.

6.2 Compilation results of FFT implementations

The serialized FFT implementation consists of a single radix-2 butterfly that is iterated repeatedly to perform the complete FFT. The data for the butterfly in different FFT stages are provided by locally buffering the computed result. The compilation is performed without using the destroyfieldarray and unroll options and the results are presented in Table 2.

<table>
<thead>
<tr>
<th>Event REGs</th>
<th>ALUs</th>
<th>BREG ALUs</th>
<th>FREG ALUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialized FFT</td>
<td>86</td>
<td>39</td>
<td>16</td>
</tr>
<tr>
<td>Parallelized FFT</td>
<td>31</td>
<td>163</td>
<td>141</td>
</tr>
</tbody>
</table>

The local buffers are mapped to the internal RAM-PAEs of XPP and thus the largest size of FFT that can be implemented is 256 without adding any extra resources. For larger FFTs, the local buffers can be mapped to external RAM, for which additional control logic will be required.

The parallelized FFT implementation consists of five stages of computations and it is compiled by using the unroll 256 and destroyfieldarray optimizations and the results are presented for a complete configuration of 32-point FFT. The resource usage will grow for larger FFTs and in that case temporal partitioning [3] can be used to divide it into a number of configurations. In order to obtain the performance results of the FFT implementations extensive testing is yet to be done.

7 Conclusions

A study has been performed by using a stream language to program a reconfigurable array processor. A few example implementations have been developed and a code generator has been designed to map applications developed in StreamIt to XPP.

From the study of the StreamIt language, it is concluded that the simple structures of the language make the program flow visualization quite easy, which facilitates the debugging process and enhances the reusability of components. The study also reveals that it can be adapted to a variety of parallel array based architectures by implementing some architecture specific modifications in the compiler back end.
Moreover, it is evident from the performance results that fine-grained filter structures are mapped efficiently to XPP. It is also observed that the compiler is unable to decompose the arrays declared within the filter body in the StreamIt code, which results in generation of extra event control logic and makes it difficult for the XPP mapper to route the application. Another observation is that XPP is suitable for arithmetic operations but the conditional constructs in the StreamIt code are not mapped efficiently to XPP.

As a future work we propose that a partitioner can be developed to partition the StreamIt generated code of the application between the host processor and the XPP, where XPP acts as a co-processor with a host processor performing the control of the application flow and assigning computational tasks to the XPP. Another approach for compilation for XPP is to generate NML code directly from the SIR representation.

Finally we would like to thank the StreamIt group at MIT and PACT XPP Technologies for their support and give special thanks to Dr. Verónica Gaspes of Halmstad University for her guidance and suggestions throughout the project.

8 References


