Engineering of Reconfigurable Systems and Algorithms (ERSA'05), Las Vegas, NV, USA, June 27-30, 2005.

Compiling Stream-Language Applications to a Reconfigurable Array Processor

Zain-ul-Abdin and Bertil Svensson
Centre for Research on Embedded Systems (CERES),
Halmstad University, Halmstad, Sweden.

1 Introduction and Motivation

Programming of media streaming applications such as HDTV voice and video processing or baseband processing 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 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 [2].

We present 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, 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.

2 StreamIt Compilation Methodology for XPP

2.1 Code generation

The source file is parsed by Kopi front end for syntax and semantic analysis. Then the Kopi IR is translated to SIR code consisting of a stream graph of filters, pipelines, split-joins and feedback loops [3]. 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 run time library. The back end converts the object-oriented style of StreamIt filters into procedural code and schedules the execution of filters.

2.2 C-code transformations for XPP

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 records 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. 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. The supported fixed-point operations are add, subtract, assignment, multiply and divide.

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 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 inlined 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.

2.3 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 XPP core [4]. These binaries can also be used for simulation and debugging purposes.

3 Implementation Results

3.1 Performance comparison of two integer FIR implementations

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. 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.

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 requires less array accesses due to fewer filters, and consequently the implementation requires fewer resources when mapped to XPP.

In order to minimize the resource usage, all the array accesses are eliminated in favor of local variables by using `destroyfieldarray` and `unroll` optimizations. The dead code in the optimized code generated is eliminated by the XPP-VC compiler. The results in simulation cycles reveal that both the fine-grained FIR and FIR in NML produce one output sample per cycle. The coarse-grained FIR implementation has a less efficient throughput because it contains data array constructs which can not be decomposed efficiently by the compiler. The execution of this filter produces one output sample per 16 cycles.

3.2 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 local buffers are mapped to the internal RAM-PAEs of XPP and thus the largest size of FFT that can be implemented in XPP64A-1 without adding any extra resources is 256. For larger FFTs, the local buffers are to be mapped to external RAM, for which additional control logic will be required.

The parallelized FFT implementation consists of five stages of computations. The resource usage will grow for larger FFTs and in that case temporal partitioning [2] can be used to divide it into a number of configurations to fit it into XPP64A-1. In order to obtain the performance results of the FFT implementations extensive testing is yet to be done.

4 Conclusions

It has been observed that fine-grained filter structures are mapped efficiently to XPP but the compiler is unable to decompose the arrays declared within the filter body in 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.

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

References

[1] Karczmarek, M.; et al. “Phased scheduling of stream programs”, *Proc. 2003 Conference on Languages, Compilers, and Tools for Embedded Systems*, San Diego, CA, 2003.

[2] PACT Software Design System M64 Reference Manual version 4.0 (PSDS-M64_Reference_Manual.pdf), Mar. 2004.

[3] Gordon, M.; et al. “A stream compiler for communication-exposed architectures”, *MIT Tech. Memo TM-627, Cambridge, MA*, 2002.

[4] Baumgarte, V.; May, F.; Nuckel, A.; Vorbach, M. and Weinhardt, M. “PACT XPP – A self-reconfigurable data processing architecture”, *Proc. 1st Int’l Conference of Engineering of Reconfigurable Systems and Algorithms (ERSA’01)*, Las Vegas, NV, June 2001.