XPP Technologies

Programming XPP-III Processors

White Paper

For further information and questions, please contact support@pactxpp.com.

Version 2.0.1

July 13, 2006

1 Introduction

This White Paper describes the programming methods and tools for XPP-III processors. First, the profiling and partitioning of a given application is discussed. Next, different methods of programming a XPP-III core are presented, separately for the sequential Function-PAEs (FNC-PAEs) and the XPP dataflow array (ALU-PAEs and RAM-PAEs). Finally, PACT's PSDS tools are described. For basic information on XPP-III processors, please refer to the White Paper *XPP-III Processor Overview*.

2 XPP-III Software Development Flow

Figure 1 gives an overview of the application development process: First, a standard C/C++ implementation of an application is profiled on a FNC-PAE. Next, the application is partitioned into multiple parallel threads in order to exploit the XPP-III processor's task-level parallelism (cf. section 2.1). The threads run either on the dataflow array or on FNC-PAEs. Then, if required, the profiling and partitioning process is repeated until the XPP resources are exploited optimally. Optionally, the application can be further optimized by using NML code (for programming the dataflow array, cf. section 3.3) and FNC assembler code (for the FNC-PAEs, cf. section 4.3). All C, NML and FNC assembler codes are processed with PACT's PSDS tools, cf. section 5.

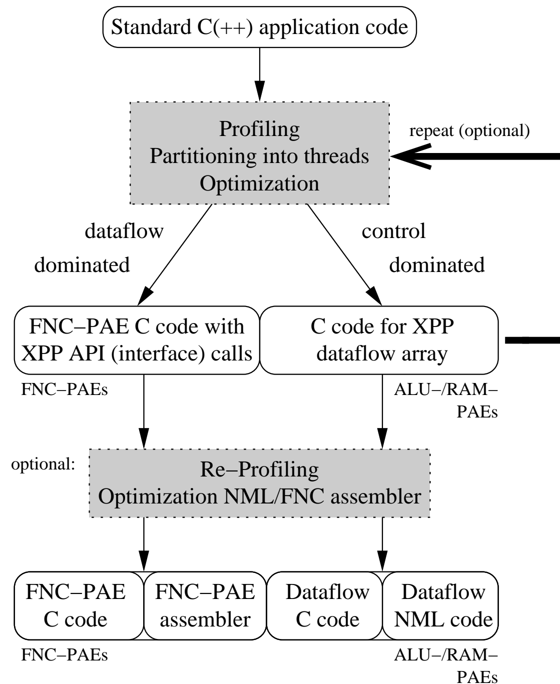


Figure 1: XPP-III Software Development Overview

2.1 Application Profiling and Partitioning

Any application can be directly compiled to a FNC-PAE and run on it. However, in order to achieve the full XPP-III performance, partitioning of the application code into multiple threads running on several FNC-PAEs and on the dataflow array (ALU- and RAM-PAEs) is required.

For a good partitioning, the sequential code is first profiled by the *XPP Profiler*. Based on the profiling results, the most time-consuming function calls and inner program loops are identified. These code sections are likely candidates for acceleration on the XPP dataflow array, especially if they are *regular*, i. e. if the same computations are performed on many data items. They are called *dataflow sections* in this White Paper since the computations can be performed by data streaming through dataflow graphs. In the C/C++ code, these sections are typically represented as loops with high iteration counts, but with few conditional branches, function calls or pointer accesses. These program parts exhibit a high degree of loop-level parallelism. However, note that the ALU- and RAM-PAEs are not restricted to processing pure dataflow graphs. They can handle nested loops and nested conditions as well.

The dataflow sections are extracted into own threads mapped to the XPP dataflow array. Several threads can be combined in one configuration, depending on the array size of the XPP-III processor.

For the dataflow threads, several options for generating XPP dataflow configurations exist (cf. section 3): (a) the XPP-VC compiler is used to directly convert the C code to a XPP dataflow configuration, (b) an optimized configuration is selected from a Module Library, or (c) an optimized configuration is programmed in PACT's proprietary, assembler-level *Native Mapping Language* (NML).

If time-consuming *irregular* code exists in the application, a coarse-grain parallelization into several FNC-PAE threads is very useful. This allows to even run irregular, control-dominated code in parallel on serveral FNC-PAEs.

Application Example: In an MPEG decoder, the following computationally intensive dataflow sections can be identified for the dataflow array: inverse quantization and DCT, motion compensation and picture reconstruction, deblocking filter and color space conversion. On the other hand, entropy decoding is an irregular, sequential algorithm suitable for FNC-PAEs. ■

The threads (whether allocated to the dataflow array or to FNC-PAEs) all access shared system memory. They synchronize and communicate via *XPP API* calls (XPP I/O functions) which are mapped to the XPP communication network. The hardware implementation of a ready/acknowledge protocol in this network enables very fast and efficient thread communication. Nevertheless, unnecessary synchronization points should be avoided to enable as much concurrent processing as possible. Special API calls are used for configuring, starting and clearing the dataflow array. In order to minimize the reconfiguration overhead, the XPP dataflow array should be reconfigured as infrequently as possible. A sufficiently large processing phase must execute between reconfigurations. In some cases the application needs to be optimized to achieve this

goal. For more details on reconfiguration trade-offs, please refer to the White Paper *Reconfiguration on XPP-III Processors*.

Application Example: In an MPEG decoder, the XPP dataflow array must be reconfigured between several configurations, e. g. IQ/IDCT, motion compensation/picture reconstruction, and deblocking/color conversion. In order to reduce the reconfiguration overhead, each configuration should process many macro-blocks before the next configuration is loaded. ■

3 XPP Dataflow Array Programming

This section describes the options for generating XPP dataflow array (i. e. ALU-/RAM-PAE) configurations for the dataflow threads.

3.1 C Programming with XPP-VC

PACT's *XPP Vectorizing C Compiler* (XPP-VC) provides the fastest way to generate XPP configurations. It directly translates standard C functions to XPP configurations. The original application code can be reused but may require some adaptations since XPP-VC cannot handle C++ constructs, pointers, and floating-point operations. Furthermore, specific XPP I/O functions (corresponding to the XPP API calls on the FNC-PAEs) must be used for synchronization and for data transfers.

The XPP-VC compiler uses vectorization techniques to execute suitable program loops in a pipelined fashion, i. e. data streams taken from memory or from I/O ports flow through operator networks. In this way many ALUs are continuously and concurrently active, exploiting the XPP dataflow array's high performance potential. Profiling the generated XPP configurations (through simulation or execution on XPP hardware) determines how to reduce bottlenecks such as memory accesses and how to optimize the code. In some cases, the C code has to be split into two or more configurations if the dataflow array size of the given XPP-III processor is insufficient.

Application Example: In the MPEG decoder, the picture reconstuction or color space conversion are examples for algorihms well suited for processing with XPP-VC. ■

Code Example: The C code in Figure 2(a) is a small `for` loop with a conditional assignment and a XPP I/O function for a port output. The XPP functions are defined in file `XPP.h` which must therefore be included. Figure 2(b) shows the dataflow graph for this program generated by XPP-VC. While the counter `COUNT` controls the loop execution, the comparator `LT` and the multiplexer `SWAP` select the result being forwarded to the output port `DOUT0`. The dotted arrow from `LT` to `SWAP` is an event (i. e. one-bit control) connection. All other connections are data connections. ■

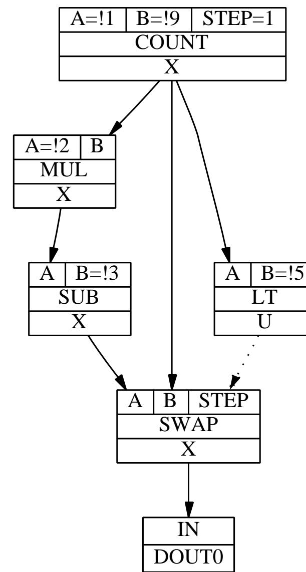
```
#include "XPP.h"

#define N 10

main() {
  int i, res;

  for (i = 0; i<N; i++) {
    if (i < 5)
      res = i;
    else
      res = 2 * i - 3;
    XPP_putstream(1, 0, res);
  }
}
```

(a)



(b)

Figure 2: XPP-VC Loop Example

3.2 NML Library Modules

For commonly used high performance function, optimized implementations from a NML Module Library (provided by PACT or third parties) can be used. They are optimal in terms of performance, area efficiency and power dissipation. There are two module types: complete configurations which only need to be configured and controlled by the FNC-PAE program, and modules which can be included in larger configurations. The latter can be included in a XPP-VC program. Hence it is possible to mix optimized NML modules for the most critical application kernels and C code. XPP-VC will create a NML configuration which contains one or more instantiations of the optimized modules along with automatically generated NML code.

Application Example: In the MPEG decoder, an optimized NML Library module can be used for the computation-intensive inverse DCT. It can be combined in one configuration with the C implementation of the inverse quantization algorithm. ■

3.3 NML Programming

If a highly optimized implementation for a dataflow section is required, but no suitable NML Library module available, the code can be directly implemented in NML. The effort is comparable to assembler programming but much simpler than HDL (e. g. Verilog or VHDL) design. In contrast to HDL design, only the functionality of the dataflow section needs to be described in NML, but no timing issues arise.

In NML, arithmetic expressions are described like C expressions and automatically converted to operator trees. Counters, memories, dataflow operators (stream multiplexers, demultiplexers, mergers, etc.), accumulators, and event operators (for program control) are explicitly allocated and connected to other operators or expressions. They are used to implement conditional computations (branches) and iterative computations (loops). Instead of using arithmetic expressions, all operators can also be explicitly allocated. This allows to place them manually on the XPP dataflow array. Hierarchical modules allow component reuse, especially for repetitive layouts.

```
XPP(XPP-III, 10,8, 8,6;
    DATA_BIT_WIDTH = 24  CONFIG_BIT_WIDTH = 24
    FREG_DATA_PORTS = 4  BREG_DATA_PORTS = 4
    FREG_EVENT_PORTS = 4 BREG_EVENT_PORTS = 4
    IRAM_ADR_WIDTH = 9)

MODULE forexample {
  OBJ ctr: COUNT {     // loop counter
    A =! 1
    B =! 9
    STEP = 1
  }
  OBJ cond: LT {       // loop condition
    A = ctr.X
    B =! 5
  }
  // arithmetic expression for else branch:
  SIG DATA elsebranch  // signal definition
  elsebranch = EXPR(2 * ctr.X - 3)

  OBJ mux: SWAP {      // result multiplexer
    A = elsebranch
    B = ctr.X
    STEP = cond.U
  }
  OBJ outport: DOUT0 { // output port
    IN = mux.X
  }
  // pipeline delay balancing command:
  DELAY_BALANCE(ctr -> outport)
}
```

Figure 3: NML Loop Example

Code Example: The NML code in Figure 3 is a direct NML implementation of the dataflow graph in Figure 2(b). Here, the else branch of the condition is described as an arithmetic expression, and all other operators are explicitly allocated. Note that the `XPP` statement at the top of the file defines the XPP core parameters, and the `DELAY_BALANCE` command is used for pipeline balancing to optimize the throughput of the loop body. ■

As with XPP-VC, it is possible to instantiate NML Library modules in a manually designed NML configuration to achieve maximum performance with reduced programming effort.

4 FNC-PAE Programming

This section describes the options for programming FNC-PAEs.

4.1 C Programming with FNC-CC

PACT's *FNC-PAE C Compiler* (FNC-CC) compiles ANSI C programs to FNC-PAE assembler. All C language features are supported, including floating-point operations which are emulated by the integer ALUs. XPP API functions are used to configure the XPP dataflow array, to communicate with the dataflow array, or to communicate and synchronize with other FNC-PAE threads.

FNC-CC is similar to a conventional RISC compiler, but uses some features of VLIW compilers (e. g. merging basic blocks to superblocks) to take advantage of the code's intrinsic instruction-level parallelism. It maps the graph representation of the superblocks to the FNC-PAE's 8-ALU matrix (via graph matching), thereby utilizing as many ALUs as possible per opcode.[1]

Profiling the FNC-PAE threads determines how to reduce bottlenecks such as memory accesses and how to optimze the code.

4.2 FNC Library Modules

If very high performance for a specific FNC-PAE function is required, optimized assembler functions from a FNC Module Library (provided by PACT or third parties) can be used. The optimized assembler functions are simply called by the C program.

Application Example: In the MPEG decoder, an optimized FNC Library function for the computation-intensive entropy codecs can be used. As an additional advantage, the assembler function benefits from special I/O instructions provided by the FNC-PAE. ■

4.3 FNC Assembler Programming

If a highly optimized FNC-PAE implementation is required, but no suitable FNC Library module available, the code can be directly implemented in FNC assembler. It supports all FNC-PAE hardware features.

The assembler uses three-address code for most instructions, as in this example:

```
SUB target, source1, source2
```

Multiple ALU instructions are merged into one FNC opcode as follows: The instructions for the left and right ALU columns are separated by a horizontal bar (|), and the

[1]Advanced features like software pipelining and predicated instruction execution will be supported in future releases.

ALU rows (at most four) are just described one by one. A FNC opcode is terminated by the keyword `NEXT`.

Code Example: The FNC assembler code in Figure 4 sequentially multiplies two 8-bit numbers (in registers `r0` and `r1`, with the 16-bit result in `r2`. Note that this example was only chosen for demonstration purposes. In a real application, the built-in single-cycle MUL instruction would be used instead. The first opcode initializes the registers, including the loop counter `r7`. The second opcode (after the label `loop`) contains all loop computations, including counter decrement, test and jump. The predicates before the instructions have the following meanings: `CY` indicates that `ADD` is only executed if the shift `SHRU` above it had a carry-out value one, and `ACT` means that `SUB` is executed (activated) in any case. `ZE NOP ! HPC loop` instructs the FNC-PAE to perform a single-cycle jump to label `loop` (*high-performance continue* = HPC) if the `SUB` instruction above it did not set the zero flag (`ZE`). This means that every loop iteration requires only one cycle. If `r7` is zero, i. e. the `ZE` flag set, the program execution continues after the loop. ■

```
; initialize parameters for test
      MOV r0, #10  ; operand 0
      MOV r1, #6   ; operand 1
      MOV r2, #0   ; clear result register
      MOV r7, #8   ; loop counter init
      NEXT

loop:

      SHRU r0, r0, #1 | SHL r1, r1, #1
  CY  ADD r2, r2, r1
  ACT SUB r7, r7, #1
  ZE  NOP ! HPC loop
      NEXT

...
```

Figure 4: FNC Assembler Loop Example

5 PACT Software Design System (PSDS)

This section describes the design flow and the tools contained in the PACT Software Design System (PSDS).

5.1 Design Entry

Figure 5 shows the PSDS design entry tools. As explained above, XPP-VC compiles C to NML code, and FNC-CC compiles C to FNC assembler. All NML files (whether

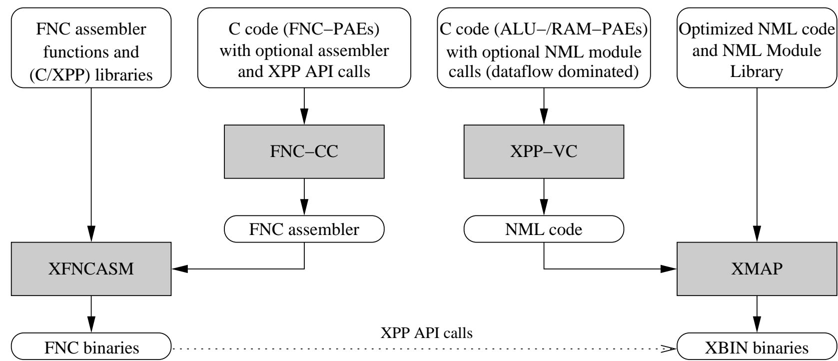


Figure 5: PSDS Design Entry Tools

generated by XPP-VC, manually designed, or from the NML Module Library) are processed by the XPP mapper *XMAP*. It compiles NML source files, automatically places and routes the configurations, and generates XBIN binary files. All FNC assembler files are processed by the FNC-PAE assembler *XFNCASM*.

Code Example: Figure 6 shows the placement and routing of the NML code in Figure 3, as determined and displayed by XMAP. ■

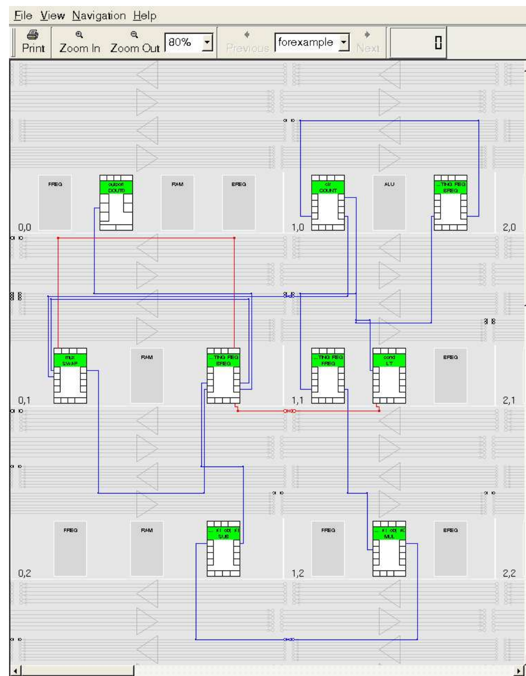


Figure 6: XMAP Screenshot for Loop Example

5.2 Execution/Simulation and Debugging on XPP-III Processors

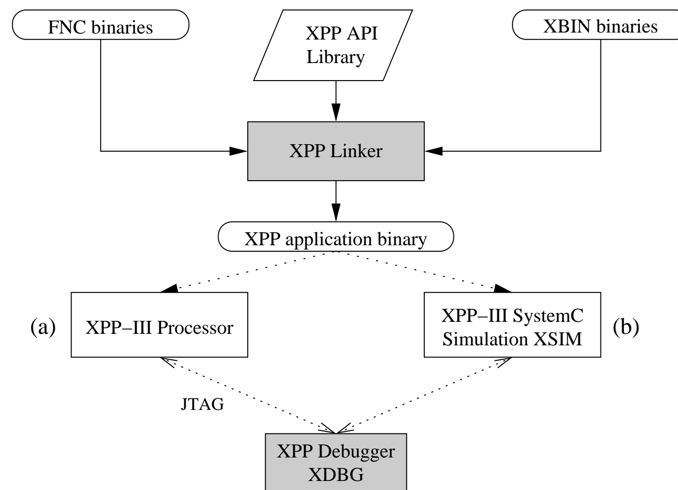


Figure 7: Execution/Simulation and Debugging on XPP-III Processor

Figure 7 shows the remaining part of the PSDS tool chain. The FNC and XBIN binaries and the XPP API Library are linked to an XPP application binary. This binary is (a) either loaded to the XPP-III processor and executed on it, or (b) simulated on the XPP-III simulator *XSIM*, cf. Figure 7. XSIM is implemented as a SystemC library which can be integrated in any C/C++ or SystemC simulation environment or used in a stand-alone simulator.

In both cases, the application can be debugged by the XPP debugger *XDBG*. This tool visualizes the data being processed on the XPP-III processor cycle by cycle. It also ollects the input data for the XPP Profiler (not shown in the figure).

6 Summary

This White Paper first discussed the steps involved in porting a C or C++ application to a XPP-III processor: profiling, partitioning into parallel threads and optimization.

Next, the PACT Software Design System PSDS was presented. The tool chain covers the entire design flow: design entry (C, NML or FNC assembler), compilation, code generation and linking. Additionally, a complete SystemC-based, cycle-accurate processor simulator and a sophisticated debugger with advanced visualization features are provided.