The Journal of Supercomputing, 26, 167–184, 2003
© 2003 Kluwer Academic Publishers. Manufactured in The Netherlands.

PACT XPP—A Self-Reconfigurable Data Processing Architecture

V. BAUMGARTE
PACT Informationstechnologie AG, Muthmannstr. 1, D-80939 München, Germany

G. EHLERS
PACT Informationstechnologie AG, Muthmannstr. 1, D-80939 München, Germany

F. MAY
PACT Informationstechnologie AG, Muthmannstr. 1, D-80939 München, Germany

A. NÜCKEL
PACT Informationstechnologie AG, Muthmannstr. 1, D-80939 München, Germany

M. VORBACH
PACT Informationstechnologie AG, Muthmannstr. 1, D-80939 München, Germany

M. WEINHARDT
PACT Informationstechnologie AG, Muthmannstr. 1, D-80939 München, Germany

Abstract. The eXtreme Processing Platform (XPP^{TM}) is a new runtime-reconfigurable data processing architecture. It is based on a hierarchical array of coarsegrain, adaptive computing elements, and a packet-oriented communication network. The strength of the XPP^{TM} technology originates from the combination of array processing with unique, powerful run-time reconfiguration mechanisms. Parts of the array can be configured rapidly in parallel while neighboring computing elements are processing data. Reconfiguration is triggered externally or even by special event signals originating within the array, enabling self-reconfiguring designs. The XPP^{TM} architecture is designed to support different types of parallelism: pipelining, instruction level, data flow, and task level parallelism. Therefore this technology is well suited for applications in multimedia, telecommunications, simulation, signal processing (DSP), graphics, and similar stream-based application domains. The anticipated peak performance of the first commercial device running at 150 MHz is estimated to be 57.6 GigaOps/sec, with a peak I/O bandwidth of several GByte/sec. Simulated applications achieve up to 43.5 GigaOps/sec (32-bit fixed point).

Keywords: reconfigurable processor, adaptive computing, run-time reconfiguration, partial reconfiguration, XPP

1. Introduction

The limitations of conventional processors are becoming more and more evident. The growing importance of stream-based applications makes reconfigurable architectures an attractive alternative [17]. They combine the performance of ASICs with the flexibility of processors.

This paper presents the eXtreme Processing Platform (XPP^{TM}), a new family of runtime- and self-reconfigurable IP cores (XPP^{TM} Cores) and processors [1, 15]. A prototype device, the XPU128-ES, has been produced in silicon by PACT. The XPP^{TM} architecture is designed to support different types of parallelism: pipelining, instruction level, data flow, and task level parallelism. Therefore this technology is well suited for applications in multimedia, telecommunications, simulation, signal processing (DSP), graphics, and similar stream-based application domains.

1.1. The XPP idea

This section gives a brief introduction to XPP^{TM} data processing. The main idea is to combine data-stream processing in an array configuration with sophisticated run-time reconfiguration mechanisms. Configurations are parallel computation modules derived from a data-flow graph of an algorithm. Nodes of the data-flow graph are mapped to fundamental machine operations such as multiplication, addition etc. The operations are implemented by configurable ALUs. The ALUs communicate via an automatically synchronizing, packet-oriented communication network. Figure 1 shows a small configuration performing a parallel matrix-matrix multiplication. In this algorithm, data packets continuously stream through a single configuration. During the computation, the graph remains static, i.e., no operators or connections are changed. No reconfiguration or instruction decoding is required, and all operators and input and output ports are active in each cycle, resulting in the optimal performance [9].

Several of these configurations can be executed sequentially on an XPP^{TM} Core. Since each configuration processes long data streams, the reconfiguration overhead is amortized over many parallel operations. Results of computations are stored in distributed memories or FIFOs for use by subsequent configurations [10]. We call this programming paradigm configuration flow, as opposed to the instruction flow in a classical Von-Neumann architecture. The difference is illustrated in Figure 2. This programming paradigm is highly suited to computation-intensive applications since many of them can be separated into smaller, inherently parallel phases.

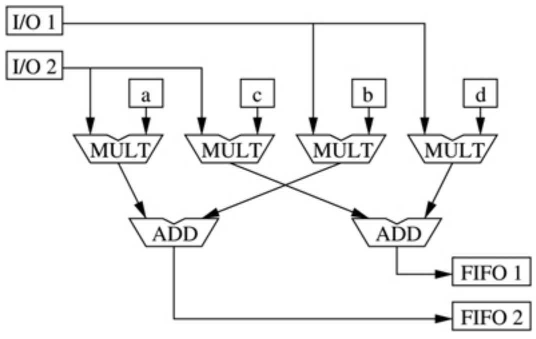


Figure 1. A small configuration: matrix-matrix multiplication.

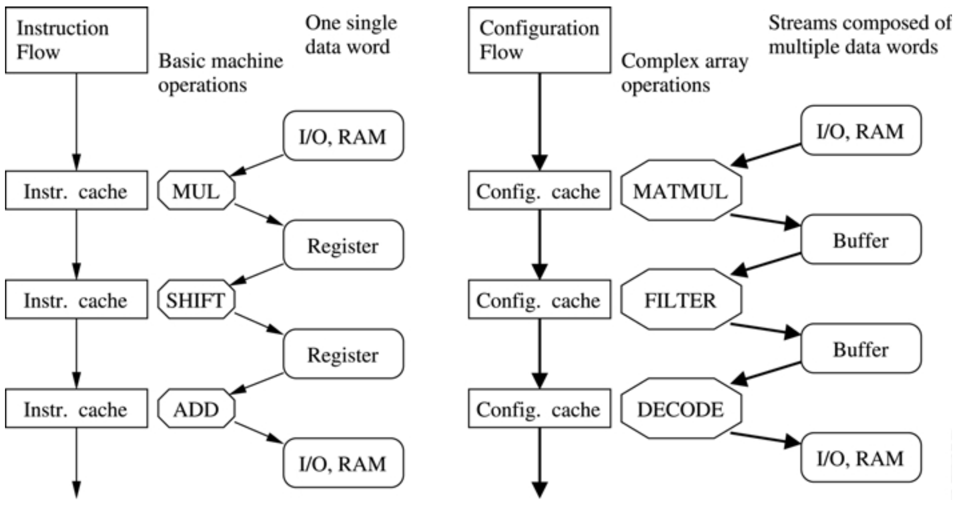


Figure 2. From instruction flow to configuration flow.

The reconfiguration overhead is reduced to a minimum by caching configurations. Therefore, subsequent configurations are available instantly. The flow of configurations automatically synchronizes with the data streams.

1.2. Paper organization

The remainder of this paper is organized as follows: the next section compares our work with alternative approaches. Then, Sections 3 and 4 introduce the XPP^{TM} architecture as well as its configuration methods. Section 5 outlines application mapping for XPP^{TM}, Section 6 gives some applications and performance results, and Section 7 concludes the paper.

2. Related work

This section defines several classes of reconfigurable computing architectures and compares them with the XPP^{TM} architecture.

Field programmable gate arrays (FPGAs) are the root of all reconfigurable computing devices. They use fine grained cells and operate at the gate level. Devices like the Xilinx XC4000 series have been used for the first reconfigurable computing hardware and software experiments. Since they only allow complete configuration and cannot hold internal data during reconfiguration, the resulting performance was only acceptable for algorithms very well suited to the FPGA architecture. Therefore other architectures were invented.

Partially reconfigurable FPGAs were the first devices available on the market which were more suitable for reconfigurable computing since only the required resources need to be configured. Well known families of partially reconfigurable FPGAs are Xilinx XC6200 and Atmel AT6000. However, they were not very successful in the market. The new Xilinx Virtex-II family seems to be a promising architecture of this kind.

Multi-context PLDs are very similar to FPGAs since they typically use a fine grained architecture. The main difference is that they contain multiple planes of context memory. If there is enough time to load a configuration to an inactive "shadow plane", the configuration can be changed on the fly by switching the plane. It is doubtful whether these devices overcome technical and commercial problems and will be actually available on the market.

Combinations of microcontrollers and FPGAs are the first step towards a complete, programmable reconfigurable system. The microcontroller manages and executes the configuration and reconfiguration of a state of the art FPGA. However, this approach is no real solution for reconfiguration performance and synchronization issues. The GARP architecture [2, 6] was a basic architecture for the software and compiler development of reconfigurable technologies. Meanwhile there are several products available on the market, e.g., the Atmel AT94K, the Altera EPXA series, and the Triscend Reconfigurable System-on-Chip series.[1]

Reconfigurable processors are the most advanced class of reconfigurable architectures. The PACT XPP™ architecture belongs to this class. Reconfigurable processors have a coarse-grained architecture and work at the operator level. Configuration files are shorter than for FPGAs and therefore reconfiguration is faster. These architectures support higher abstraction levels for the programmer and advanced reconfiguration and synchronization methodologies. Early members of this class were the KressArray [5], RaPiD [4], and Raw Machines [18] which were specially designed for streaming algorithms. Morphosys [7] and REMARC [8] contain programmable ALUs with a reconfigurable interconnect and are controled by a microprocessor. Chameleon System's CS2000 family [16] is similar to these architectures. Configuration memory defines the operations and connections of an array of ALUs for a certain period of time. Then the memory content is changed by a processor or memory planes are switched (similar to multi-context PLD) to define a new configuration. The main differences between XPP™ and these architectures are XPP™'s automatic packet-handling mechanisms and its sophisticated hierarchical configuration protocols for full or partial runtime- and self-reconfiguration which lead to a high level of software abstraction and applicative performance.

3. Architecture

The XPP™ architecture is based on a hierarchical array of coarsegrain, adaptive computing elements called processing array elements (PAEs), cf. Section 3.1, and a packet-oriented communication network, cf. Section 3.2. Regular streaming applications can be implemented efficiently on this array. Distributed event signals within the array add additional flexibility for less regular applications since events can be used to control the data streams. However, the real strength of the XPP^{TM} technology originates from the combination of array processing with unique, powerful run-time reconfiguration mechanisms. Since configuration control is distributed over several configuration managers (CMs) embedded in the array, PAEs can be configured rapidly in parallel while neighboring PAEs are processing data. Entire applications can be configured and run independently on different parts of the array. Reconfiguration is triggered externally or even by special event signals originating within the array, enabling self-reconfiguring designs, cf. Section 4. By utilizing protocols implemented in hardware, data and event packets are used to process, generate, decompose and merge streams of data.

3.1. Array structure

An XPP^{TM} device contains one or several processing array clusters (PACs), i.e., rectangular blocks of PAEs. Figure 3 shows the structure of a typical XPP^{TM} device. It contains four PACs (a). Each PAC is attached to a CM responsible for writing configuration data into the configurable objects of the PAC. Multi-PAC devices contain additional CMs for concurrent configuration data handling, forming a hierarchical tree of CMs. The root CM is called the supervising CM or SCM. It has an external interface (dotted arrow originating from the SCM in Figure 3) which usually connects the SCM to an external configuration memory. The interface consists of an address bus, a data bus, and control signals.

The XPP^{TM} architecture is also designed for cascading multiple devices in a multi-chip setup. In this arrangement, SCMs act like ordinary, subordinate CMs [12].

A CM consists of a state machine and internal RAM for configuration caching, cf. Figure 3(b). The PAC itself contains a configuration bus which connects the CM with PAEs and other configurable objects. On the lowest level, the objects have small individual configuration caches. Horizontal busses carry data and events. They can be segmented by configurable switch-objects, and connected to PAEs and special I/O objects at the periphery of the device.

A PAE is a collection of PAE objects, cf. Figure 4. The typical PAE shown in Figure 4(b) contains a BREG object (back registers) and an FREG object (forward registers) which are used for vertical routing, as well as an ALU object which performs the actual computations. The ALU object's internal structure is shown in Figure 4(a). The ALU implemented in the XPU128-ES prototype performs common fixed-point arithmetical and logical operations as well as several special three-input opcodes such as multiply-add, sort, and counters. Events generated by ALU objects depend on ALU results or exceptions, very similar to the state flags of a classical microprocessor. A counter, e.g., generates a special event only after it has terminated. The next section explains how these events are used.

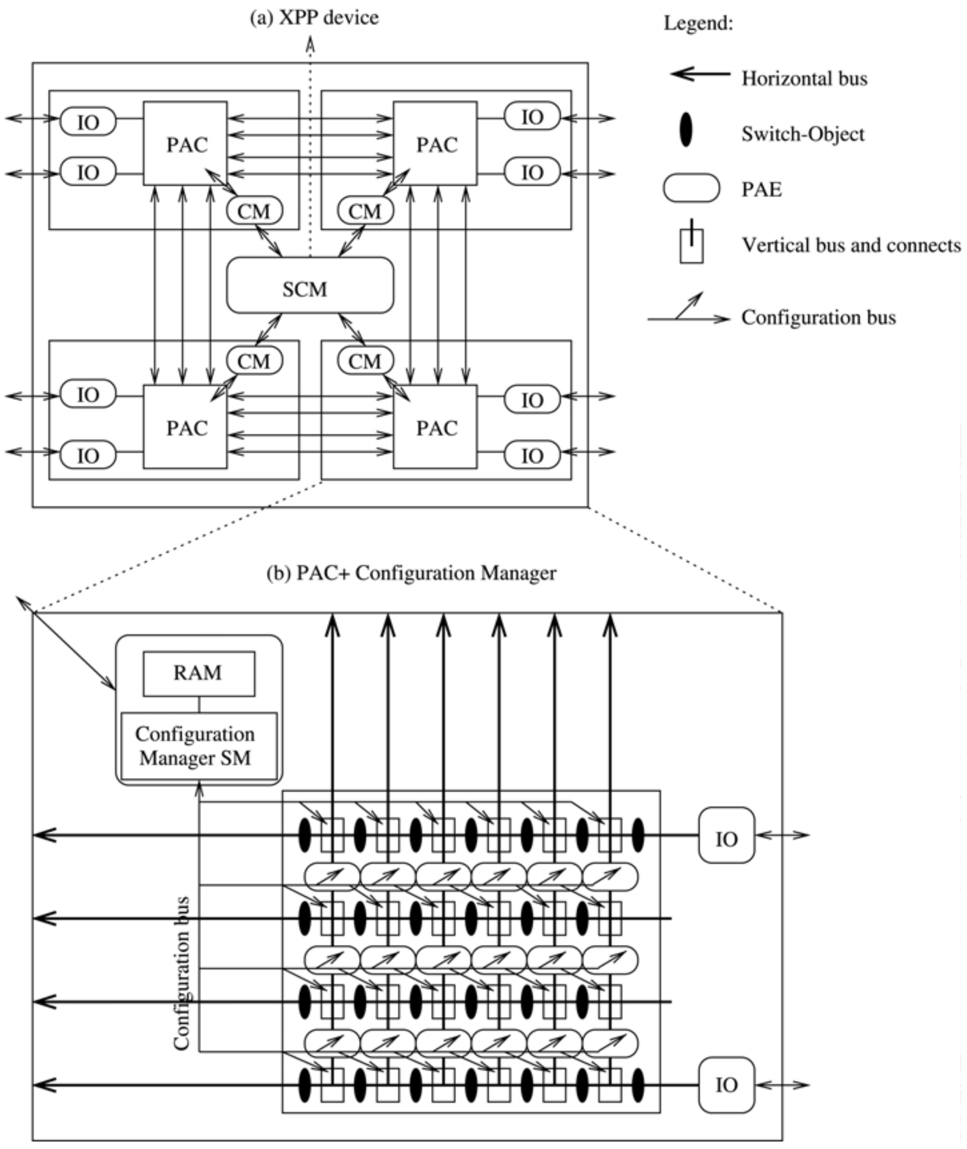


Figure 3. Structure of an XPP device.

Another PAE object implemented in the prototype is a memory object which can be used in FIFO mode or as RAM for lookup tables, intermediate results etc. However, any PAE object functionality can be included in the XPP^{TM} architecture.

The XPU128-ES prototype contains two PACs; each consisting of a square of 64 32-bit ALU-based PAEs, 16 memory-based PAEs (one KByte each), and 4 32-bit I/O units (with two channels each). Additionally, an adder/subtracter is integrated in each BREG object since these are particularly useful for optimal performance on DSP algorithms.

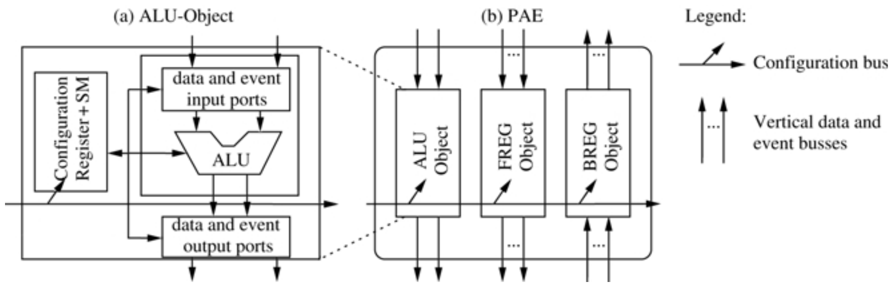

(a) ALU-Object
(b) PAE
Legend:
Configuration Register + SM
data and event input ports
ALU
data and event output ports
ALU Object
FREG Object
BREG Object
Configuration bus
Vertical data and event busses

Figure 4. PAE and ALU structure.

3.2. Packet handling and synchronization

PAE objects as defined above communicate via a packet-oriented network. Two types of packets are sent through the array: data packets and event packets. Data packets have a uniform bit width specific to the device type. As mentioned above, the prototype device uses 32-bit data.

In normal operation mode, PAE objects are self-synchronizing. An operation is performed as soon as all necessary data input packets are available. The results are forwarded as soon as they are available, provided the previous results have been consumed. Thus it is possible to map a signal-flow graph directly to ALU objects, and to pipeline input data streams through it. The communication system is designed to transmit one packet per cycle. Hardware protocols ensure that no packets are lost, even in the case of pipeline stalls or during the configuration process. This simplifies application development considerably. No explicit scheduling of operations is required.

In contrast to data packets, event packets are just one or a few bits wide. They transmit state information which controls ALU execution and packet generation [11]. For instance, they can be used to control the merging of data-streams or to deliberately discard data packets. Thus conditional computations depending on the results of earlier ALU operations are feasible. Events can even trigger a self-reconfiguration of the device. Details are provided in the next section. Furthermore, events can be combined. Depending on the configuration, an output event is either created when one of the input events occurs, or it is delayed until all input events have occurred.

4. Configuration

The configuration and reconfiguration methods employed in a reconfigurable processor are crucial to its performance. Processors with slow configuration are only useful for very regular data-flow applications with a high computation to configuration ratio. On the other hand, processors which handle configurations fast and flexibly are also beneficial for less regular applications. This is especially true if configuration and computation can occur concurrently on the same device. Therefore very sophisticated, unique configuration techniques were devised for the XPP^{TM} architecture.

4.1. *Parallel and User-Transparent Configuration*

As mentioned in Section 1.1, an XPP^{TM} application normally consists of several phases which are implemented by configurations. After a phase has been executed, the XPP^{TM} Core needs to be reconfigured.

For rapid reconfiguration, the configuration managers in the CM tree operate independently, and therefore are able to configure their respective parts of the array in parallel. An entire word of configuration information is sent to the PAC in each clock cycle through a dedicated configuration bus, cf. Section 3.1. To relieve the user of the burden of explicitly synchronizing a sequence of successive configurations, every PAE locally stores its current configuration state, i.e., if it is part of a configuration or not (states "configured" or "free"). If a configuration is requested by the supervising CM, the configuration data traverses the hierarchical CM tree to the low-level or leaf CMs which load the configurations onto the array.

The leaf CM locally synchronizes with the PAEs in the PAC it configures. Once a PAE is configured, it changes its state to "configured". This prevents the respective CM from reconfiguring a PAE which is still used by another application. The CM caches the configuration data in its internal RAM and in the PAEs until the required PAEs become available. Hence the caches and the distributed configuration state in the array enable the leaf CMs to configure their respective PACs independently. No global synchronization is necessary. The highest configuration performance can be achieved if configurations are used several times. Since the caches can hold several configurations, reloaded configurations are already resident in the PAC and can be loaded instantly.

Figure 5 shows a situation where several configurations are already cached in the leaf CMs. First, configuration *Conf1* which uses both PACs is loaded on both PACs in parallel. Next, *Conf2* is processed in the same way. Finally, *Conf3* and *Conf4* as well as *Conf5* and *Conf6* are independently loaded on PAC a and PAC b, respectively.

4.2. *Computation and configuration*

While loading a configuration, all PAEs start to compute their part of the application as soon as they are in state "configured". The hardware protocols mentioned in Section 3.2 ensure that partially configured applications are able to process data without loss of packets. This concurrency of configuration and computation hides configuration latency. Additionally, a pre-fetching mechanism is used. After a configuration is loaded onto the array, the next configuration may already be requested and cached in the low-level CMs' internal RAM and in the PAEs. Thus it need not be requested all the way from the SCM down to the array when PAEs become available.

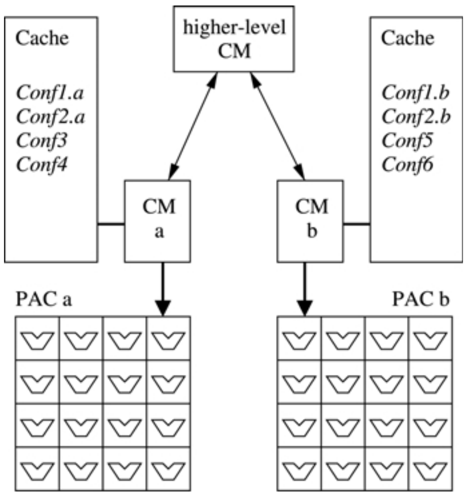


Figure 5. Parallel and independent configuration of multiple CMs.

To exploit the full configuration bandwidth, it is not sufficient to delay loading subsequent configurations until the previous one has terminated. Since the sets of used PAEs only partially overlap for many configurations, those PAEs not used in the previous configuration can be configured earlier, while those which are used must actually wait until the previous configuration terminates and is removed, cf. Section 4.3. However, this strategy causes deadlock situations if multiple partially loaded configurations have mutually exclusive PAE allocation requirements. Therefore a safe extension of the simple "first-in first-out" strategy was developed and implemented in the XPP^{TM} architecture's configuration protocols [15]. It avoids these deadlocks and yet it allows subsequent configurations to overlap.

The performance increase of overlapping configuration is obvious if the next configuration is only waiting for one single object of the previous configuration. All but one objects of the new configuration can already be configured before the old one has been removed. The configured part of this new configuration may have already computed a big part of its job. Completely independent successive configurations will even execute completely concurrently.

Finally, an example will demonstrate our method. In Figure 6, four configurations (A to D) are requested in a sequence. The available configuration and computation parallelism will be exploited automatically at runtime without additional information from the programmer.

The upper part of the figure displays the configurations as defined by the programmer. These four configurations process two data streams *data1* and *data2*. Both streams are preprocessed in individual configurations (A and C). The configurations are automatically removed after termination, cf. Section 4.3. After preprocessing, the data streams are stored in local memories. In A, this is done to decouple the input stream from the output stream processing (B). In C, the reason is to reuse processing elements in D. The data in both memories is post-processed and then output in the given order D after B, i.e., *data2* after *data1*.

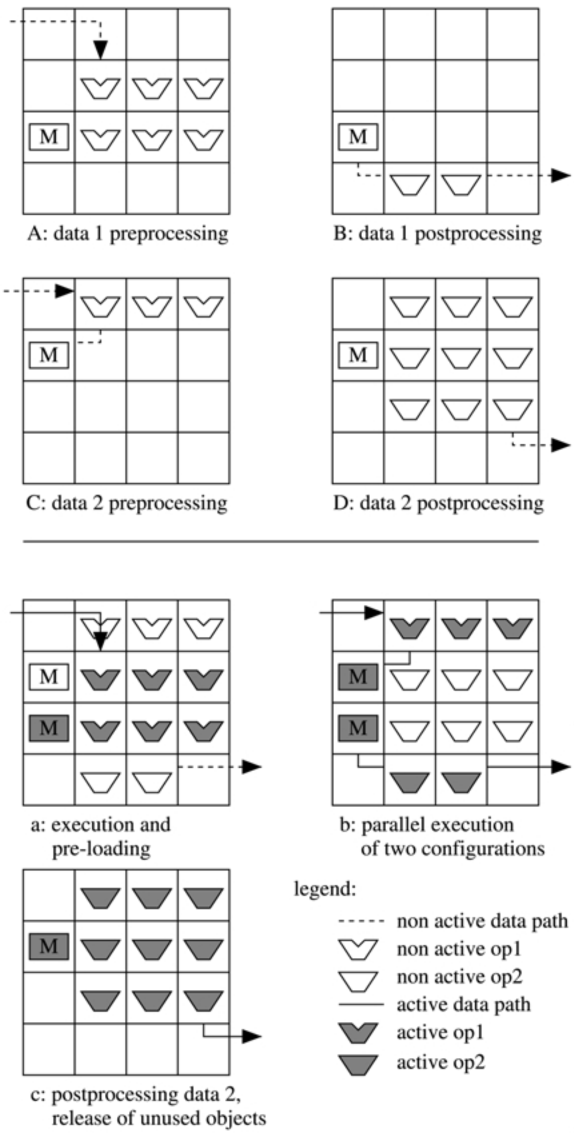


Figure 6. Example of overlapped configuration and computation.

The lower part of the figure shows a possible execution scenario of these configurations on a PAE array. Note that the correct execution is guaranteed even if runtime effects change the exact configuration order.

Three snapshots are displayed in the scenario (a to c). In snapshot (a), the configuration A is already configured and running. While input data is being processed, parts of configurations B and C are already loaded but not yet started. Configuration and computation occur concurrently.

In snapshot (b), two configurations (B and C) run in parallel. C receives new data while B post-processes its data. We see that multiple configurations can execute independently. Additionally, all available PAEs for configuration D are already configured.

The last snapshot (c) shows configuration D running. All objects not needed anymore are now freed and could be used by further configurations. As soon as the last data word leaves the data buffer, the memory can be released and reused while the current configuration still processes the last data word.

4.3. Self-reconfiguration

Reconfiguration and pre-fetching requests can be issued by any CM in the tree (including the SCM which can respond to external requests) and also by event signals generated in the array itself. These signals are wired to the corresponding leaf CM. Hence running applications can request a self-reconfiguration of the device. It is possible to execute an application consisting of several phases without any external control. By selecting the next configuration depending on the result of the current one, it is possible to implement conditional execution of configurations and even arrange application phases (i.e., configurations) in loops.

The array automatically reconfigures itself when a phase is finished, and the next phase immediately starts computing. However, in order to enable the CM to load the next phase, the state of all PAEs must be reset to "free". A special event input is used for this purpose. The PAE detecting the end of the phase (e.g., a counter) sends the event to all other affected PAEs and thereby resets them. Alternatively, the respective CM resets the PAEs' states.

4.4. Differential configuration

So far, we only discussed the configuration and removal of complete configurations. This is appropriate for applications in which the configurations of the phases differ largely. However, in some cases phases are very similar, e.g., in adaptive filter applications. For such cases, differential configurations are much more effective then complete configurations. They do not change a PAE completely, but change only parts of its configuration, e.g., a constant input or an ALU opcode.

As opposed to complete configurations, differential configurations only describe changes with respect to a given complete configuration. Each differential configuration therefore relates to a complete base configuration. Consider an adaptive FIR filter as an example: The base configuration configures the adders and multipliers, the delays and all required wiring. While all this remains unchanged, a subsequent differential configuration only changes the constant coefficients of the filter taps. This largely reduces the number of configuration cycles compared to a complete configuration. Since the few changed constants can be cached in the PAEs it is even possible to synchronize the data flow with the reconfiguration. Hence the filter can be adapted with virtually no delay in the processed data stream.

5. Application mapping

In order to exploit the unique capabilities and performance of the novel XPP^{TM} architecture, it is necessary to map applications carefully. Therefore the Native Mapping Language, a PACT proprietary structural language with reconfiguration primitives, was developed. It gives the programmer direct access to all hardware features. Additionally, we are developing a C complier.

5.1. Native Mapping Language (NML)

In NML [13], configurations consist of modules which are specified as in a structural hardware description language, similar to, for instance, structural VHDL. PAE objects are explicitly allocated, optionally placed, and their connections specified. Hierarchical modules allow component reuse, especially for repetitive layouts. Additionally, NML includes statements to support configuration handling. A complete NML application program consists of one or more modules, a sequence of initially configured modules, differential changes, and statements which map event signals to configuration and pre-fetch requests. Thus configuration handling is an explicit part of the application program.

The complete XPP^{TM} Development Suite XDS [14] is available for NML programming. Figure 7 shows the design flow. The main component is the mapper `xmap` which compiles NML source files, places and routes the configurations, and

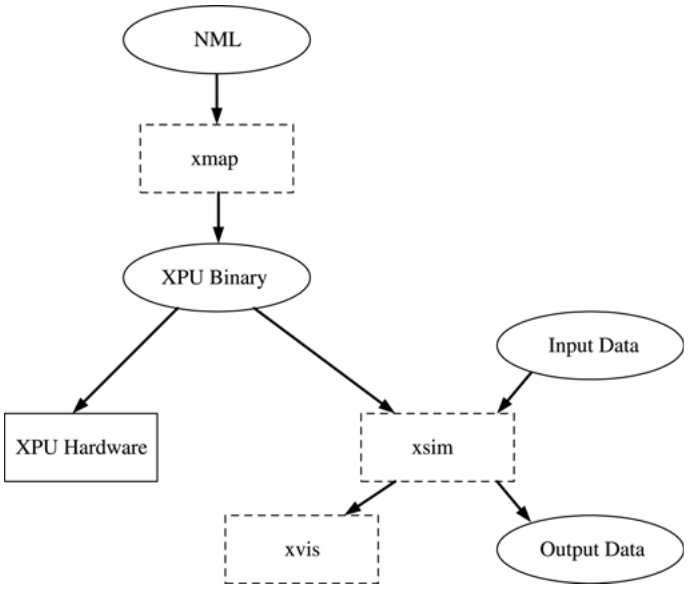
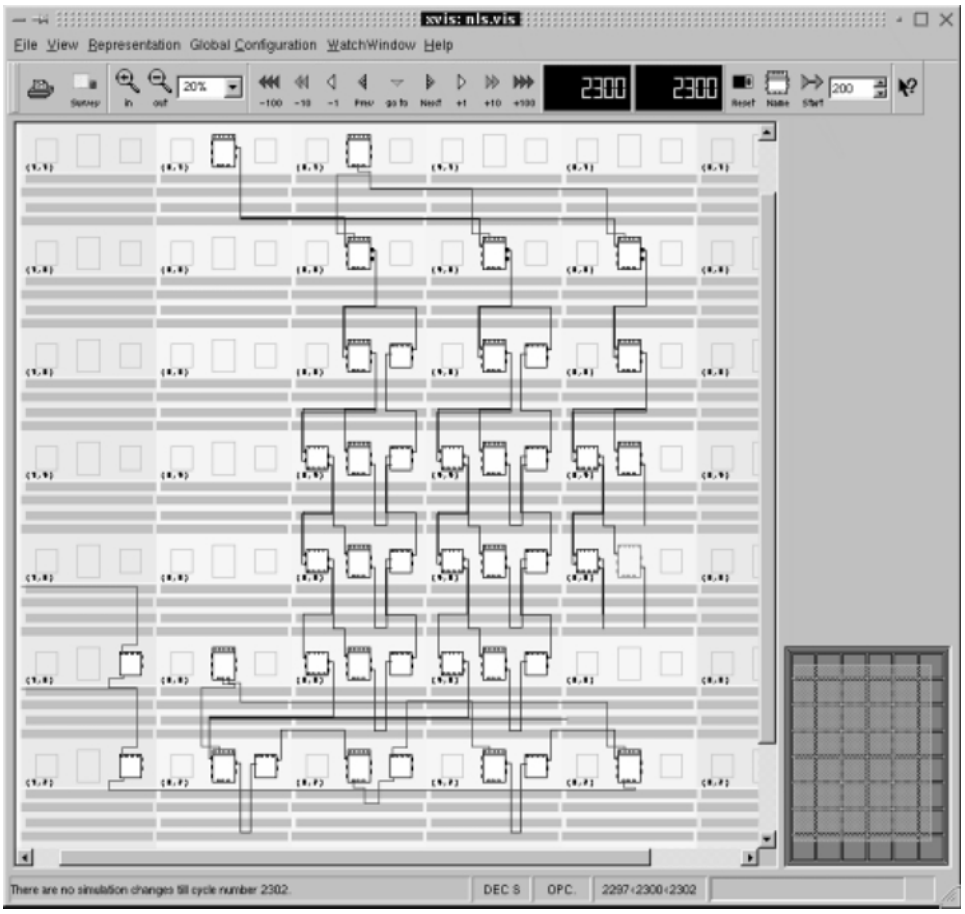


Figure 7. NML design flow.


xvis: nls.vis
File View Representation Global Configuration WatchWindow Help
20%
−100 −10 −1 Prev go to Next +1 +10 +100
2300
2300
Reset Name Start
200
DEC S
OPC.
2297<2300<2302

There are no simulation changes till cycle number 2302.

Figure 8. `xvis` screen shot.

generates binary files. These can either be simulated and visualized cycle by cycle with the `xsim` and `xvis` tools in the suite, or directly executed on an XPP^{TM} device.

Figure 8 shows a screen shot of `xvis`. For each clock cycle, all configured ALUs, registers and buses as well as all register values are visualized.

5.2. *XPP-VC compiler*

The Vectorizing C Compiler XPP-VC translates C functions to NML modules. They are restricted to a standard C subset and use an XPP^{TM}-specific I/O library. Using the NML mapper, these functions run directly on an XPP^{TM} device. The compiler uses vectorization techniques [19] 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. Hence many ALUs are continuously and concurrently active, exploiting the XPP^{TM} device's high performance potential.

To execute a large program on an XPP^{TM} Core, it is split into several consecutive temporal partitions [3]. Each partition is mapped to its own NML configuration. Hence the configuration flow paradigm introduced in Section 1.1 is also supported by XPP-VC.

For maximum performance, the XPP-VC design flow allows inclusion of manually optimized NML modules from a library, i.e., to mix C and NML. Hence it is possible to hand-code the most critical kernels of an application in NML while other application parts are specified in C. An NML module can either be used as a complete partition or configuration, or as part of a larger one.

In a later stage, XPP-VC will be extended to a codesign compiler covering full ANSI C. It extracts the most time consuming parts of the original C source code for XPP^{TM} execution. I/O commands are inserted automatically, and the selected parts are compiled by XPP-VC. The remaining parts of the C source will run on a conventional host processor using the XPP^{TM} device as an adaptive coprocessor. The host communicates with the XPP^{TM} Core via its I/O ports, shared memories, and the external SCM interface. The host program will automatically load and remove configurations on the XPP^{TM} device. Thus the physical size of the XPP^{TM} hardware will not restrict the amount of program code running on the XPP^{TM} hardware. Figure 9 shows the resulting C compiler design flow.

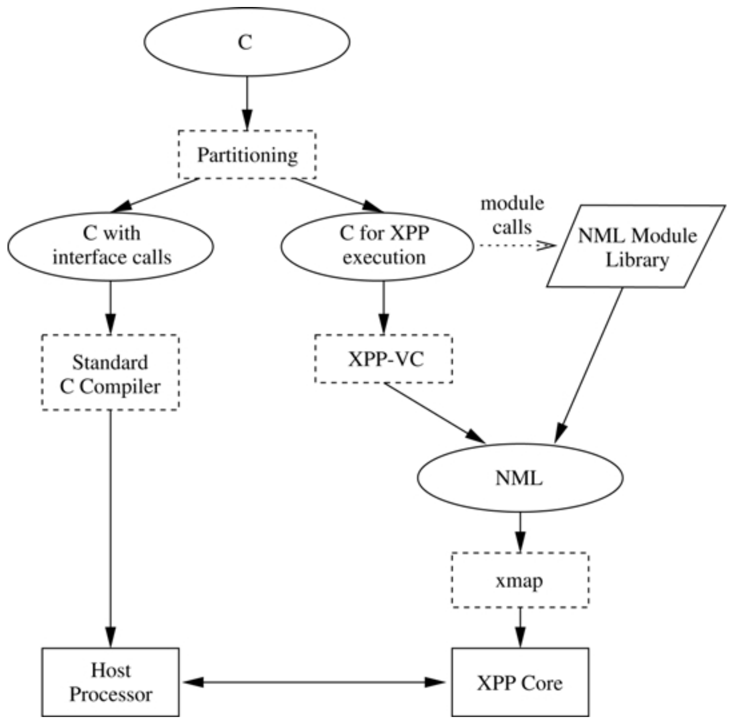


Figure 9. XPP-VC design flow.

6. Applications and performance

Currently, an improved version of the XPU128-ES prototype is being developed. Depending on the CMOS process used, devices run at different frequencies. Values are given for 150 MHz. Using all operators of the device (an adder and a multiplier in the ALU object, and an adder or subtracter in the BREG object) yields 384 32-bit fixed point operations per cycle, resulting in a peak performance of 57.6 GigaOps/sec at 150 MHz. This compares to a peak performance of only a maximum of eight operations per cycle for a high-end VLIW DSP.

Table 1 shows the simulated performance of several applications, assuming a device running at 150 MHz. Regular applications can utilize all PAEs. A 128-tap FIR filter achieves 38.4 GigaOps/sec or 19.2 billion MAC (multiply-accumulate) operations/sec. The same application with complex computations achieves 43.5 GigaOps/sec as the BREG objects are used for computations as well. The other applications in Table 1 are a 3×3 median filter, a coordinate transformation kernel (in triangle image rendering), matrix-matrix multiplication, the inner loop of the MPEG4 standard, and an FFT kernel. They do not utilize all PAEs and therefore achieve a lower performance, however they leave more unused PAEs for other applications running in parallel.

In the following we discuss the last two applications in more detail. The implemented part of the MPEG4 algorithm contains a 2D DCT and quantization as well as inverse DCT and inverse quantization. We implemented a 1D DCT algorithm which processes two image pixels per clock cycle utilizing 20 PAEs. Since two iterations of the 1D DCT are required, a 2D DCT applied to an 8×8 image block takes 64 clock cycles (plus 10 cycles pipeline latency). To process a 128×128 pixel image, 8,192 clock cycles (plus 10 cycles latency) are required. Eight additional PAEs and two lookup tables (internal RAM blocks) are sufficient to implement a simple adaptive quantization scheme. Summarizing these results, 31 PAEs (20 for the DCT including memory access routines, two lookup tables for the quantization matrix, one lookup table for the zig-zag read address generation and 8 PAEs for the quantization) are sufficient to process one pixel per cycle. Since the 2D DCT and the quantization have to be combined with an inverse DCT and an inverse quantization, basically the same computation process has to be applied a second time. 62 PAEs would be necessary to process one pixel per clock cycle in a parallel implementation of the complete MPEG4 inner loop. Hence an XPP^{TM} device consisting of 64 PAEs

Table 1. Application performance

Application	Ops/cycle	GigaOps/sec
Integer FIR	256	38.4
Complex FIR	290	43.5
Median filter	38	5.7
Coordinate transformation	36	5.4
Matrix multiplication	64	9.6
MPEG4 inner loop	31	4.7
FFT8 kernel	32	4.8

can compress a video sequence with a resolution of 1,024 $\times$ 1,024 pixels and full color (3 $\times$ 8 bit) at 50 frames per second.

The FFT application analysis yields the following results: An 8 point complex FFT kernel was implemented on a 32 PAE device which supports 12.12 fixed point operations. Using this FFT8 kernel, 16 iterations are needed to compute a 64 point complex FFT. Based on the FFT8 kernel, a sequence of 64 samples can be processed at once and sent to the kernel. After 11 clock cycles latency the first 64 complex results are stored in XPP^{TM} FIFO buffers. Using a butterfly address generator structure, these intermediate results are read out and again sent to the FFT8 kernel, after 64 twiddle factor multiplications. Thus after $128 + 11$ clock cycles, the FFT64 result is available at the XPP^{TM} output port. Using 64 PAEs, the analysis has shown that a FFT64 can be implemented such that one complex input sample is processed per clock cycle.

Let us now consider the reconfiguration performance. Because of its course-grain nature, an XPP^{TM} device can be configured rapidly. Since only a partial configuration of those PAEs and switch objects which are actually used is necessary, the configuration time depends on the application. For instance, the FIR filter in Table 1 requires 13 configuration words per PAE. Hence the initial configuration (for the entire device) takes 1,664 cycles. As mentioned in Section 4, computation and configuration can overlap. If a convenient configuration order is chosen, each PAE can start processing data as soon as it is configured. Thus the number of active PAEs increases constantly during configuration until the peak performance is achieved.

Rapid partial reconfiguration also enables effective multitasking. Several independent smaller applications which all fit on a device at the same time run concurrently at their peak performance. Hence the overall number of operations per cycle is the sum of those of the individual tasks. The tasks can be configured and removed individually while the remaining tasks run at full speed.

The pre-fetching and self-reconfiguration features introduced in Section 4 are also useful for adaptive algorithms, such as adaptive filters in non-linear systems modeling. In this application, one part of the XPP^{TM} Core always holds a state detection and control module. This module requests a new partial filter configuration when it is required. While the old filter configuration performs computations at full speed, the new one is configured into another part of the device. Next, the data stream is redirected to the new filter. If the filters have the same structure, it is possible to synchronize the reconfiguration with the data stream, cf. Section 4.4. In both cases the performance is not degraded, and the data stream is not interrupted. Other applications which take advantage of self-reconfiguration include beam-forming algorithms used in mobile phone base stations.

7. Conclusions

We have introduced the novel XPP^{TM} data processing architecture, our approach to application mapping, and some applications and performance results. We showed that regular applications can be directly and very efficiently mapped to XPP^{TM}

devices, yielding a very high performance. Packet synchronization techniques automatically handle pipeline stalls and similar situations. The peak performance of a commercial device is estimated to be 57.6 GigaOps/sec, with a peak I/O bandwidth of several GByte/sec. Simulated DSP and multimedia applications achieve up to 43.5 GigaOps/sec (32-bit fixed point). The XPP^{TM} approach provides a significant improvement in performance over standard processor and DSP implementations, and much more flexibility than ASIC implementations. Furthermore, configuration caching and pre-fetching in the hierarchical CM tree allows switching rapidly between configurations in a user-transparent manner, thereby providing a "virtual" processor much larger than the actual hardware.

Within two or three years we expect devices containing up to 512 ALUs organized in 32 PACs with a CM tree depth of 4 or 5, running at 750 MHz. The resulting peak performance will be about 800 GigaOps/sec.

Note

1. For more information on the mentioned devices, refer to the vendors' web pages at www.xilinx.com, www.atmel.com, www.altera.com, and www.triscend.com.

References

1. V. Baumgarte, F. May, A. Nückel, M. Vorbach, and M. Weinhardt. PACT XPP—A self-reconfigurable data processing architecture. In *Proc. International Conference on Engineering of Reconfigurable Systems and Algorithms ERSA '2001*, Las Vegas, NV, 2001.
2. T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and C compiler. *IEEE Computer* 33(4):62–69, 2000.
3. J. M. P. Cardoso and M. Weinhardt. XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture. In *Proc. Field-Programmable Logic and Applications; 12th International Conference*. Springer-Verlag, 2002.
4. C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD—reconfigurable pipelined datapath. In *Proc. Field-Programmable Logic and Applications; 6th International Workshop*, Springer-Verlag, 1996.
5. R. Hartenstein, R. Kress, and H. Reinig. A new FPGA architecture for word-oriented datapaths. In *Proc. Field-Programmable Logic; 4th International Workshop*, Springer-Verlag, 1994.
6. J. R. Hauser and J. Wawrzynek. Garp: A MIPS processor with a reconfigurable coprocessor. In *Proc. FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1997.
7. M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, and F. J. Kurdahi. Design and implementation of the morphoSys reconfigurable computing processor. *Journal of VLSI and Signal Processing-Systems for Signal, Image and Video Technology*, March 2000.
8. T. Miyamori and K. Olukotun. A quantitative analysis of reconfigurable coprocessors for multimedia applications. In *Proc. FPGAs for Custom Computing Machines*, IEEE Computer Society Press, 1998.
9. PACT Informationstechnologie GmbH: 1998a, 'WO 98/26356'. Patent.
10. PACT Informationstechnologie GmbH: 1998b, 'WO 98/29952'. Patent.
11. PACT Informationstechnologie GmbH: 1998c, 'WO 98/35299'. Patent.
12. PACT Informationstechnologie GmbH: 1999, 'WO 99/44120'. Patent.
13. PACT Informationstechnologie GmbH: 2001a, 'NML Language Reference Manual'.
14. PACT Informationstechnologie GmbH: 2001b, XDS User Manual: Using the Mapper, the Simulator, and the Visualizer.
15. PACT Informationstechnologie GmbH: 2001c, XPP Technology White Paper. www.pactcorp.com.

16. B. Salefski and L. Caglar. Re-configurable computing in wireless. In *Proc. 38th Design Automation Conference*, Las Vegas, NV, 2001.
17. R. R. Vemuri and R. E. Harr (eds.) *IEEE Computer Special Issue on Configurable Computing*, IEEE Computer Society Press, April 2000.
18. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, and P. Finch. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, 1997.
19. M. Weinhardt and W. Luk. Pipeline vectorization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(2), 2001.