NoC Datapath for Polymorphic Processors in Embedded Systems

Joshua Weber, Erdal Oruklu, Jafar Saniie
Department of Electrical and Computer Engineering
Illinois Institute of Technology, Chicago, Illinois, 60616

Abstract—Polymorphic processing has the goal of producing a processor combining the advantages of general purpose processing with the significant gains achievable by custom application specific computing. To achieve these ends a novel polymorphic processor architecture is presented. Incorporating networking on a chip (NoC) techniques into the datapath design has the potential to provide noticeable advantages when compared to a traditional processor datapath, especially for reconfigurable platforms such as FPGAs. This paper presents an architecture integrating NoC concepts into the design of a processor datapath in order to create a polymorphic processor. The paper will further explore and analyze the effect of topology choices and NoC design on the performance of polymorphic processors with specific focus on the impacts of NoC datapath integration.

I. INTRODUCTION

Applications have become common drivers for performance improvements, exerting a constant pressure to improve the performance of computer processor, especially within the embedded systems field. As limitation of power and physical constraints on clock cycle come into play, processor designers are increasingly turning to alternate techniques to improve performance. The integration of application specific processing elements with general purpose processors and system on a chip (SoC) integrated designs is the primary driver for future performance improvements [1]. Custom designed processing elements are able to drastically outperform general purpose processors for data driven processing, while general purpose processors excel at flexibility and system control. To overcome interconnect scalability when integrating high numbers of computational units a new networking on a chip (NoC) paradigm is often applied.

The advent of field programmable gate arrays (FPGA) and the constant increase in both performance and resources provided has enabled a new field of custom logic design: reconfigurable computing. Once again drivers of increased performance and lower costs have pushed for integration of processors and reconfigurable fabrics. Polymorphic processors integrate the two, producing a processor which can be constantly reconfigured to behave as an optimal application specific processor.

The production of a polymorphic processor is difficult to realize. Many of the primary obstacles will be presented and analyzed in this paper. A new NoC based datapath that will better achieve the goals of polymorphic processing will be proposed and discussed. In addition, simulation of this new architecture will be presented.

II. POLYMORPHIC PROCESSORS

Reconfigurable computing has been the subject of much research [2], and it has been shown that reconfigurable computing can provide a significant improvement in performance over standard general purpose processors [3]. Reconfigurable architectures offer a reduction in size and cost, improved time to market, and increased flexibility. The majority of reconfigurable systems focus on the integration of a general purpose processor with a reconfigurable resource array (most often a FPGA). The goal of a polymorphic processor is to integrate the two units into a common design methodology and run-time control system. The current trends and types of architectures has been studied and is summarized in Fig. 1.

In Fig. 1(a) the reconfigurable array is communicating with the general purpose processor through the I/O data bus. This design is the most straightforward and well documented approach combining a reconfigurable element with a general purpose processor. Most often this is used to produce highly configurable SoC based designs, allowing fast time to market using existing SoC tools.

The primary advantage of this design is the ease of implementation. The processor choice is a standard general purpose processor which is well understood and is very easily implemented. The processor is coupled to traditional reconfigurable fabrics. This often allows concurrent development of the processor software and the reconfigurable array, allowing both to use industry standard and well used design techniques. The major drawbacks are the limited bandwidth available over the processor I/O bus. Most applications that can achieve large performance gains from reconfigurable fabrics are highly data driven. The limited bandwidth of the I/O bus for most processors limits the speed-up achievable. In addition as all communication to the reconfigurable fabric must be controlled and passed through the processor, a large degree of overhead is present for any computation done on the reconfigurable resources. This overhead is often significant and robs the system of the performance gains from execution on reconfigurable resources. This approach is by far the most common and accounts for all commercial reconfigurable platforms. Many commercial tools exist to produce SoC designs coupling reconfigurable fabrics to standard general purpose processors.

In Fig. 1(b),1(c) we see the reconfigurable array moved closer to the processor to decrease the communication cost and overhead: first through integration of the fabric into the
Fig. 1. Architectural Classes for Reconfigurable Computing

base memory space of the processor, allowing communications directly with the fabric without use of a memory controller or peripheral bus, and second as a reconfigurable co-processor, granting direct access to the reconfigurable fabric at the instruction level.

The trend for these designs is a more tightly coupled design with the goal of decreasing the overhead and bandwidth limitations that arise from the Fig 1(a) designs. This enables a higher communication bandwidth and more fine grained coupling to the execution software application. There are some increasing drawbacks to this design. With the tighter coupling, the addition of the reconfigurable logic begins to affect the design and operation of the processor. The use of a reconfigurable coprocessor, Fig. 1(c), requires modification to the instruction set architecture (ISA). Modification of the ISA introduces compatibility issues for existing applications running entirely on the processor and not utilizing the reconfigurable fabric. Furthermore, the design process for reconfigurable fabric and software application must be unified, increasing the complexity of overall system design.

In Fig. 1(d) the reconfigurable fabric is a component of the datapath. The incorporation of the reconfigurable fabric into the datapath provides a very fine grained resource. Use of the reconfigurable elements is streamlined and integrated into the processor execution, greatly reducing any overhead or needed context switching. As a drawback, processor design and programming are much more complex. Modification and extension of the ISA is almost always needed. In addition optimization of the datapath is more difficult.

With the increasing size, speed, and complexity of programmable fabrics and the increasing prevalence of soft core processors, the processor core can be embedded within the reconfigurable fabric as shown in Fig. 1(e). This allows the whole processor itself to be reconfigurable and enables very fast communication between processor and reconfigurable blocks. When following this SoC based approach, any of the previous architectural types can be used.

These approaches have all been applied in very different application domains [4], [5]. In addition research has been performed into creation of polymorphic processors for multiple application domains [6].

As polymorphic processor grow in complexity they become more limited by the resources required to interconnect all components. This is further complicated by integration of reconfigurable fabric within the processor. It is not known at design time how many discrete units exist in the design, nor the individual requirements of each component. To address many of these issues NoC techniques will be applied to create a polymorphic processor.

III. NoC INTEGRATED DATAPATH

As reconfigurable resources grow and SoCs become more complex, an increasing number of discrete communicating units must be incorporated and the contribution of interconnect complexity becomes a limiting factor. This limitation in scalability can be overcome by a new paradigm, network-on-chip (NoC) [7]. This approach borrows heavily from the network design field and applies the same techniques to the SoC interconnections. Each component of the SoC is treated as a member of a micronetwork of components. There has been a great deal of research into this field, showing increased performance in many application domains. In general there are 2 types of NoC approaches. The first is a shared medium approach. A shared bus is connected to all components. Each component then communicates by a shared media protocol, most often a TDMA based design. Alternatively, the construction of mesh topologies is often used. These topologies can be either regular and homogeneous or ad hoc and heterogeneous. Many networks propose hierarchal design topology to achieve increased performance. Just a few of the researched NoC designs are Nova Interconnect [8], AEtheral [9], and XPipes [10]. NoC research is a rapidly developing field with recent
work into multiprocessor systems [11], cache coherency [12], and reconfigurable polymorphic NoC [13].

A. Proposed Architecture

This paper proposes a unique approach to the design of a polymorphic processor. The best performance gains and most tightly integrated coupling of processor and reconfigurable elements occurs with integration into the datapath. As the number of reconfigurable elements grow, the scaling of the datapath will becoming rapidly more complex and the interconnect count will grow significantly. This rapidly limits how many separate datapath elements can be added, and correspondingly, how many reconfigurable elements can be integrated into the design.

To overcome this limitation to scalability, NoC techniques can be applied to the design. NoC addresses the problem of scaling communication on a chip by introducing networking methodologies. To apply this to the goals of polymorphic processing we will replace the datapath entirely with a NoC, as shown in Fig. 2. This architecture works by encapsulating instructions into packets, which are then inserted into the NoC. The NoC then delivers them to the proper functional unit. The functional units themselves are reconfigurable. This creates a very dynamic and flexible processor architecture. The number and type of functional units can be reconfigured. Allowing the processor to be customized for widely varying workloads.

B. Implementation Consideration

Implementation of this architecture will be challenging as it is a significant divergence from traditional microprocessor architecture design. In addition, like the datapath integration above, the new polymorphic processor will need to support a dynamic ISA. This will require modifications to compiler technology to support the new polymorphic processor and changing ISA. Further issues will also need to be addressed. With the creation of a polymorphic processor new control techniques will need to be designed to provide support for reconfiguration of the processing elements. Furthermore the creation of a polymorphic processor with a dynamic datapath will potentially have a large impact on programming methodologies requiring a change in software design methods.

The processor will have a fairly traditional instruction fetch and decode unit which will feed decoded instructions directly into the packetizer/router unit. This unit provides much of the logic for control and execution of the processor. It will attempt to extract as much instruction level parallelism as possible from the instruction stream, allowing for many instructions to be executed concurrently. The packetizer will then encapsulate a packet and address and schedule its execution in a functional unit. All scheduling and execution control will be performed within this unit.

The packetized instructions will then pass through the NoC to be delivered at the proper functional unit. For most instructions multiple functional units will be required, for example a stop at the register file, continuing on to an ALU execution unit to calculate a memory address, then to the memory controller to memory access, and finally concluding at the packetizer for instruction completion. The NoC will connect to both fixed function units, i.e. register files and memory controller, and reconfigurable functional units. The reconfigurable elements can be configured to represent traditional standard processor units, such as ALU and FPU, or also can be customized to more exotic functions optimized for specific application domains.

This polymorphic processor should provide many advantages. It will allow for very fine grained customozation to each target application. This customization can occur in real time, allowing the processor to reconfigure itself to be optimized as it encounters various applications. It will have some drawbacks and difficulties that will need to be overcome. The flexible NoC communications datapath will never be able to achieve the same optimizations as a fixed path datapath. As such there will be some inherent loss of performance, but it will be compensated for by large gains from the use of application specific functional units and instantiation of multiple functional units, providing an increase in instruction level parallelism achievable. Overall this architecture should provide a new degree of reconfigurable computing and the integration of processors and reconfigurable resources.

C. NoC Topology

The implementation of the NoC will make a very large impact on the overall performance. As all instruction must be
routed multiple times over the NoC to get to each necessary functional unit, the delay from the NoC will cause a significant change in the performance of the processor. The most critical issues in NoC performance will be packet delay. Packets will need to have a short delay time in order to not slow down execution of instructions. Other important considerations are resource consumption of interconnects, complexity of implementation, and scalability.

Design and implementation of the NoC will be a challenge. Traffic analysis for instructions as packets will need to be performed, as it is unlikely that the instruction stream will follow tradition data generation patterns. A NoC topology will need to be produced that can provide adequate throughput and delay characteristics while consuming few logic and interconnect resources. Table I shows some common network topologies and their relative merits.

### IV. TARGETED PLATFORM - SPARC LEON3 PROCESSOR

To ease implementation and leverage existing design tools, e.g., compilers, linkers, and simulators, the polymorphic processor design will be based around and extend a pre-existing instruction set architecture (ISA). SPARC is well supported and has a large existing ecosystem. The SPARC architecture is a simple, reduced instruction set computing (RISC) ISA with a great deal of flexibility, including support for implementation designed co-processors and supporting existing ISA extensions.

Leon3 is a SPARC processor designed and offered by Aeroflex Gaisler. It is a fully synthesizable VHDL model of a 32-bit SPARC V8 processor. This processor is highly configurable and ideal for SoC applications. In addition, it is distributed in full VHDL source code under the GNU GPL license, making it an ideal platform for research work. With the full source code available the processor can be redesigned to incorporate any research design ideas. Utilization of this existing processor will give a frame of reference for performance, provide the ability to leverage existing IP and tools, and provide a starting point for all modifications to the datapath.

### V. NOC DATAPATH SIMULATION

#### A. Split Spectrum Processing

The split spectrum processing (SSP) algorithm is chosen as an initial target application. Research into implementation of this algorithm in embedded systems has been shown in prior work [14]. The algorithm makes a good test application. It works with a large dataset and performs computationally complex large sized FFTs.

Ultrasonic target detection is made difficult by the presence of high scattering microstructure noise. The use of frequency diverse ultrasonic testing provides signal echo data containing high amounts of statistical variation in scattering noise. This scattering noise is the result of a large number of small, randomly distributed scatters arising from the microstructure of the material. The algorithm, as shown in Fig. 3, works by decomposing the input signal into the frequency domain. Subband filtering is performed to isolate frequency dependent noise from frequency independent target data. The data is then transformed back into the time domain and recombined to provide large increases in target detection, with improvements of >10dB.

![Fig. 3. Split spectrum processing](image)

#### B. Instruction Execution Trace

The algorithm was implemented in C code and compiled targeting the SPARC ISA. Tools used were the BCC cross compiler with -O3 optimization targeting a CPU with full support for floating point (FP) instructions. The application code was then executed on the TSIM SPARC Leon3 simulator. Using the trace buffer of the simulator, the real time execution instruction trace was captured. Total execution of the algorithm is performed in 1,198,492 instructions. This is the real execution trace for the algorithm under realistic data set processing. For comparison the NoC datapath simulation will use this instruction trace as input.

#### C. Processor Datapath Simulator

A SPARC NoC datapath simulator was designed and built, using the C# language. For ease of implementation and to focus on NoC impacts, many assumptions were made regarding the operation of the processor. It is assumed that the processor has execution units that represent the core components currently existing in the SPARC processing core. These units consist of a register file, memory controller, instruction fetch unit, arithmetic logic units (ALU), and floating point units (FPU).
An additional new unit is designed, a instruction packetizer unit. This unit takes the place of the instruction decode unit. It functions to analyze each incoming instruction, create a correctly addressed packet, and then issues it into the NoC for execution. Each instruction type has a separate address list based upon its execution requirements. For example, a standard ADD instruction must first go to the register file to access its operands, then onto the ALU for execution of the arithmetic operation, returning to the register file to write the results into the destination register, and finally returning to the packetizer to be disposed. It is important to see that each packet/instruction begins and ends its life in the packetizer unit. This enables the packetizer to control execution flow and enforce in order execution return of results.

Since all packets are centrally run through the packetizer unit, it is an ideal place to perform dependency checking. As the NoC may have non-deterministic delivery of packets, instructions coexisting in the NoC may execute out of order. To maintain proper operation, all instruction in execution should be independent, working on separate pieces of program data. The packetizer maintains dependency tracking to make sure that no registers may contain stale or invalid data when accessed by another instruction. In addition the packetizer monitors for branch instruction, holding execution until the branch has been correctly resolved. This dependency checking makes the processor highly dependent on instruction level parallelism available in the program. As the NoC datapath is slower per instruction then the reference SPARC datapath to overcome the overhead penalty the processor must be able to take advantage of instruction level parallelism. This is very reasonable expectation for the types of algorithms targeted by polymorphic and reconfigurable platforms.

The execution units are simple implementations only simulating the execution time. All ALU and FPU computations are assumed to complete in one clock cycle. The register file is assumed to be able to perform two dual port access per clock cycle, simulating the ability to perform register access and write back in a single cycle, and all memory access is assumed to be completed in one clock cycle with 100% perfect cache hit rates. Overall the simulator tries to closely follow the execution of the reference SPARC design.

**D. NoC Topology**

One of the most critical design decisions when utilizing the NoC datapath is the choice of NoC topology. The execution of each instruction will require multiple trips through the NoC to each functional unit. The delay incurred by each packet during its transit of the NoC will significantly impact the overall time it takes to execute each instruction. For programs with a high degree of instruction level parallelism, there may be a high number of instruction in-flight at any point in time, with a maximum traffic of one packet produced and consumed by each functional unit every clock cycle. Furthermore, to truly support the goal of the production of a polymorphic processor, the NoC must be able to support the creation of an unknown number of additional heterogeneous functional units with the potential to scale to a large number of discrete units. In addition, cost in resources and interconnects must be balanced against performance gains for NoC design decisions. All of these factors put many constraints on the implementation of the NoC and design pressures on topology choices.

The simulator presented here utilizes a basic ring topology, as shown in Fig. 4. This topology is very easy to implement and has low resource consumption. In addition its execution is well understood and easy to model. For these reasons it was chosen as an initial topology for study into the impact of NoC datapath on processor execution. The ring structure also mimics the overall path of most instructions. Most instructions progress in a roughly linear fashion through the NoC to each functional unit, stopping at each to perform a processing step. By carefully organizing the NoC layout of functional units, the ring topology can closely follow the layout of a traditional processor datapath, drastically reducing the routing overhead penalty.

However the ring structure also has many drawbacks as a topology choice. The ring routing delay is directly related to the number of elements in the ring. As the number of function elements increase, giving a performance gain, the routing delay also increases, giving a performance penalty. This relationship tends to rob the processor of much of the performance gains from the addition of more functional units. For topologies with better scaling performance, this performance penalty will scale better allowing performance to scale more linearly with the addition of more functional units.

For this simulation it is also assumed that the NoC clock rate is operated at eight times the master processor clock. The choice of a clock multiplier for the NoC is two-fold. The first is that the processing at each NoC cycle is fairly simple and only consists of packet routing and transfer. The second is that the delay incurred by each packet in the NoC is a very significant source of instruction execution delay. By using a clock multiplier we can help to hide the network routing delay,
which is especially important for the inefficient ring topology. For a simple ring architecture an eight times multiplier is a very reasonable assumption, running the master clock of the processor at 40 MHz with an easily supported 320 MHz NoC clock. The simulator further assumes a single NoC clock cycle transfers a packet from one unit to the next in the ring.

E. Results

The results for multiple simulation runs can be seen in Table II. This table contains the results from a series of different simulation runs, each having a different set of instantiated functional units. As a base reference a run with no instruction level parallelism is also provided. During this run, each instruction executes independently of all others, essentially a design with no pipelining capabilities. The final four designs show different implementations with an increasing number of functional units. The primary advantage of a NoC datapath becomes evident as the number of functional units is increased.

Examining the results shows the clear interplay between the number of parallel execution units and the performance of the program. Furthermore it shows the balancing of increased performance from additional functional units versus the penalties applied from the large NoC delay. The simple change from a non-pipelined to pipelined design enables a noticeable improvement in overall performance due to the high level of instruction parallelism present in the target application. As more functional units are instantiated the performance of the processor improves, although it is mitigated by increases in NoC delay. Eventually the addition of functional units decreases overall performance caused by a limited extraction of parallelism being overcome by NoC delay penalties. This behavior is shown in the final case, with four ALUs and two FPUs. The addition of more functional units decreases overall system performance. This trend, of slowly increasing performance with an eventual limitation and decrease, is present for all applications, although the actual optimal point will be different for each application.

VI. Conclusion

This paper has presented a background exploration of reconfigurable computing and polymorphic processors. In addition it has presented a very unique approach to processor datapath design, based on a dynamic NoC for datapath interconnection. Through conceptual exploration of the impact of this architecture, it is apparent that the design and implementation of the NoC is critical to the overall performance of the processor, most noticeably the impact from end-to-end delay. A simulation of a SPARC based ring topology NoC datapath is presented, showing a clear demonstration of performance gains achievable through increase of the processor into a reconfigurable superscalar design. Readily apparent in the design is the interplay between increasing functional units, limits in parallelism present in the program, and corresponding increases in NoC delay characteristics. This relationship is critical. The optimal number of functional units, and overall performance is dependent on the program itself. Different application programs will perform optimally with a different number of functional units. The ability of this processor architecture to change easily and quickly enables a processor that can track the optimal performance criteria for each target application, allowing a uniform architecture to achieve application specific optimal performance.

REFERENCES