PERFORMANCE EFFICIENCY OF CONTEXT-FLOW SYSTEM-ON-CHIP PLATFORM

Rami Beidas, Jianwen Zhu

Electrical and Computer Engineering
University of Toronto
Ontario M5S 3G4, Canada
{rbeidas, jzhu}@eecg.toronto.edu

ABSTRACT

Recent efforts in adapting computer networks into system-on-chip (SOC), or network-on-chip, present a setback to the traditional computer systems for the lack of effective programming model, while not taking full advantage of the almost unlimited on-chip bandwidth. In this paper, we propose a new programming model, called context-flow, that is simple, safe, highly parallelizable yet transparent to the underlying architectural details. An SOC platform architecture is then designed to support this programming model, while fully exploiting the physical proximity between the processing elements. We demonstrate the performance efficiency of this architecture over bus based and packet-switch based networks by two case studies using a multi-processor architecture simulator.

1. INTRODUCTION

The continued advancement in semiconductor technology allows system-on-chips (SOC) to accommodate an increasing number of computational elements and embedded memory modules. So far, the industry has been using common busses and design specific communication channels to interconnect these components. Such global-wiring communication architectures are unable to scale with the large dies fabricated in the near future with a 0.1μm technology or below [3]. To overcome this problem and accommodate future applications that need massive parallelism, researchers proposed the use of interconnection networks, previously used to interconnect supercomputer components, to fulfill on-chip communication requirements [4]. While a burst of efforts have appeared under the banner of network-on-chip, we observe some common, yet important ommissions.

First, while traditional computer architecture is well abstracted with a programming model, new SOC architectures have not made much progress on that front. An SOC platform is either modeled in system-level languages, such as SystemC [5] or SpecC [6], where a distinction between application, architecture and hardware does not exist, or using traditional parallel programming models, which are usually very complex. For example, the popular Message Passing Interface (MPI) programming model [7] defines an API with 127 C functions and there is no easy path to parallelize a sequential program into an MPI program other than the use of array-oriented scientific applications. Second, while traditional networks in supercomputers are designed with the bandwidth limitation imposed by chip pin count, new SOC platforms do not take full advantage of the much relaxed physical constraints and almost unlimited on-chip bandwidth.

We propose a new solution to address these problems and the following contributions are made in this paper. First, we propose a new programming model, where, in contrast to the common practice of supersetting the C language with new syntax or APIs, we subset the C language by imposing very few constraints revolving around a new concept, called *context*, which is essentially an abstraction of autonomous dynamic data structures. An application written with this programming model is not only simple, as it is "less" than the usual C code, but also safe in the sense of Java, as it is free of problems such as free memory access and dangling pointers. Second, we propose a new SOC platform architecture, called the *context-flow* architecture, revolving around an on-chip network infrastructure called a *tunnel*, which takes full advantage of the physical proximity of tightly coupled processing elements. The tunnel implements the on-chip remote procedure call abstraction, therefore achieving the transparency of the programming model, since an application does not have to change with respect to the change in the underlying architecture, yet with a cost almost as cheap as local procedure calls, thereby achieving performance efficiency. Third, we have built a development suite by extending the popular SimpleScalar environment, which was designed for single processor architecture evaluation, so that complex applications can be compiled and simulated on the multi-processor context-flow architecture platform. We validate the performance efficiency of this architecture by real world applications.

The rest of the paper is organized as follows. In Section 2, we introduce the context-flow programming model. In Section 3 the design of baseline context-flow architecture is then described. In Section 4, we describe the performance evaluation framework we built for our architecture before we demonstrate in Section 5 its performance efficiency on two applications, namely an MP3 decoder and a cryptography accelerator. We discuss related work in Section 6 before we draw conclusions.

2. CONTEXT-FLOW PROGRAMMING MODEL

A programming model is an abstraction that separates application from architecture. This separation is important to allow applications be developed and reused across different architectures, and vise versa. A programming model can be defined at different levels of abstraction, and a hardware/software infrastructure is usually needed to support such abstraction. For example, an instruction set is a programming model defined at the low level to abstract away architectural details such as pipelining and out-of-order issue, and a massive amount of hardware logic is used to realize this abstraction. A programming language is defined at the higher level to abstract away the differences between different instruction sets, and a compiler is used to realize such abstractions. For the same programming model, a middleware infrastructure, such as CORBA [8] or DCOM, can be used to abstract away architectural details of a distributed environment to implement a distributed application the same way as a sequential one.

The importance of programming model, however, is ignored in the hardware-centric CAD community. Even though platform-based design is advocated to allow the reuse and customization of pre-aggregated components, the concept of platform has not been formalized with a programming model for applications. Recent interest in building the communication infrastructure on massive parallel SOC has led to the concept of network-on-chip. Building a programming model for network-on-chip either has to use explicit communication with send/receive system calls, a wide departure from the traditional imperative programming model, or has to build another middleware infrastructure on top of the network, leading to performance degradation with the number of layers one communication session has to go through.

We propose a new programming model formally defined in Definition 1.

Definition 1 *Given a program with a set P of procedures, operating on a program state consisting of a set B of dynamically allocated memory blocks. A block $b_i \in B$ is said to point to $b_j \in B$, if there exists a program point when the content of b_i contains the address of b_j. A* **context** *$C \subset B$ is a set of blocks that are closed under the point-to relation, that is, any block b_i reachable from block $b_j \in C$ is also an element of C. A* **contexted procedure** *is a procedure p such that all parameters of p points to memory blocks of the same context. A* **context-flow program** *$\langle P, C\rangle$ is a program such that all its procedures $p \in P$ are contexted procedures, and all its memory blocks, which capture the program state, belongs to a context $c \in C$.*

A context-flow program (CFP) is extremely *simple*: it is simply a C program with the same sequential semantics. It therefore can be compiled using any conventional compiler and executed on any conventional machine. Contexts can be implemented by using the API shown in Figure 1. While the API consists of only three functions, it is the complete API seen by the application programmer. Here, `cfNewContext` creates a context and returns a unique identifier. `cfDelContext` destroys a context, thereby reclaiming all the memory blocks contained in the context. `cfAlloc` allocates a memory block of certain size from the specified context. We now argue that a CFP is in fact simpler than a usual C program: note that the counterpart of `cfAlloc`, which should be responsible for memory block deallocation, is not provided by design. In fact, the memory is deallocated at the context level by `cfDelContext`. This relieves the task of fine-grained memory management, thereby simplifying the programming task in a way similar to garbage collection.

This simplification can lead to program *safety* in the same sense of what garbage collection brings to modern languages such as Java. A CFP is free from dangling pointers and free memory access problems thanks to the closure property of contexts: there cannot be any references to freed memory blocks, since the memory containing the reference should belong to the same context, and therefore be freed already as well. On the other hand, the implementation of context is far cheaper than a garbage collector, in fact cheaper than the *malloc/free* in a normal program: the cost of memory allocation can be confined to constant time using a stack based mechanism.

```
int      cfNewContext( void );
int      cfDelContext( int c );
void*    cfAlloc( int c, int size );
```

Figure 1: Context-flow API.

Context is designed to be an abstraction of autonomous data structures. It can be anything ranging from arrays, linked lists, trees, graphs, or the combination of all. The concept of context offers a macroscopic view of the program and therefore makes coarse-grained parallelization much easier, which shall become apparent in the next section.

3. CONTEXT-FLOW ARCHITECTURE

An architecture is an aggregate of architectural components such that an application can be executed or implemented through a well defined programming model. A *micro-architecture* is an aggregate of components such as fetch stage, decode stage, execution stage and memory stage to implement a sequential application in C or other programming languages by its instruction set. On the other hand, a *macro-architecture* is an aggregate of components such as processing elements (PEs) and memories to implement a parallel application by a programming model such as MPI. The composition of a macro-architecture in a traditional parallel system is pre-defined, whereas in the case of SOC, the composition is often customized according to one application or one family of applications. A macro-architecture is said to be *homogeneous* if all PEs are of the same type, e.g., processors, and *heterogeneous* if PEs can be microprocessors, DSPs, ASIPs or custom hardware cores.

We consider the design of a macro-architecture, called the context-flow macro-architecture (CFA), formally defined in Definition 2.

Definition 2 *Given a context-flow program $\langle P, C\rangle$, a* **context-flow macro-architecture** *is a tuple $\langle E, M, \mathcal{SC}, N\rangle$, where E is a set of processing elements (PEs), M is a set of memory banks, $\mathcal{SC} : \mathcal{P} \mapsto \mathcal{E}$ is the static* **architectural configuration** *which maps a procedure in the program to a processing elements, and $\mathcal{N}$ is the on-chip network where a runtime configuration $\langle \mathcal{F} : \mathcal{C} \rightarrow \mathcal{M}, \mathcal{A} : \mathcal{M} \mapsto \mathcal{P}\rangle$ is maintained to bind each context to a distinct memory bank, and connects each memory bank to a PE for direct access, in such a way that a program point $\langle p, c\rangle \in P \times C$ is active implies that $\mathcal{SC}(p) = \mathcal{A}(\mathcal{C}(c))$.*

Unlike an application in traditional programming model, A CFP is *highly parallelizable*, since different procedures, each accessing their own private data structures maintained in different context, can be run in a CFA in different PEs in parallel, without the concern of dependency hazard or cache coherence that frequently occur in the traditional shared or distributed memory architecture. The accesses of contexts do switch from one procedure to another, when a procedure call occurs. When the *remote procedure call* (RPC) abstraction is implemented by the on-chip network of a CFA, whose runtime configuration in Definition 2 is dynamically adjusted, then a CFP is also *highly transparent*, meaning that it does not need to be changed no matter how the PEs in a CFA is allocated, and how the procedures are mapped.

The key problem in the design of a CFA is the design of its on-chip network. We start by first defining a programming model, which abstracts how it interacts with the PEs that it connects. We define the programming model in the form of an instruction set, as shown in Figure 2. The instruction set is simple enough to contain only 10 instructions. It is encoded by the values of the wires on each port that connects a PE to the network. From the perspective of the network, it encodes a command or request from a PE. From the perspective of a PE, the instruction set is a complement of its own for which it can assume the availability of a co-processor for actual execution – effectively by driving the right wires in the corresponding ports.

```
int     cfiAllocBank( void );
void    cfiFreeBank( int bankid );
void*   cfiMalloc( int size );
word    cfiLoad( int addr );
void    cfiStore( int addr, word data );
void    cfiCnctBank( int bankid );
void    cfiRPC( int procid );
void    cfiRet( int procid );
word    cfiAckRPC( void );
word    cfiAckRet( void );
```

Figure 2: Context-flow instruction set.

We now consider how to implement an on-chip network that can implement this instruction set efficiently. There are several alternatives, each employing a different network topology.

As shown in Figure 3 (a), a *bus based* CFA maintains a private memory bank for each of its PEs, in other words, the connection configuration $\mathcal{C}$ in Definition 2 is static. The context is also maintained in its private memory bank. On the other hand, every time a RPC is invoked, the content of the corresponding context needs to be copied to the memory bank that belongs to the callee, and this data transfer is carried out by a shared bus.

As shown in Figure 3 (b), a *packet-switch based* CFA is the same as bus-based except that the data transfer can be performed more efficiently: while a shared bus may invite transfer congestion, a well designed packet-switched network can distribute the communication traffic evenly.

Like previous efforts, these two alternatives do not take full advantage of the fact that the network we are designing is on-chip, and the PEs are physically close to each other. We propose a new based on-chip network, called a CFA *tunnel*. As shown in Figure 3 (c), the tunnel maintains a pool of separate memory banks, as well as an intelligent crossbar switch. Each context is dynamically mapped to a single memory until it is deallocated, and the crossbar ensures the access to the memory is dynamically switched to the callee whenever an RPC occurs. Note that our crossbar should not be confused with the crossbars in previous efforts, which is designed still for the purpose of data transfer. Instead, the goal of our crossbar is to provide the direct, wired access for memories. RPC, or the flow of contexts from one PE to another, can then be achieved at virtually no cost!

It is important to note that there is a physical limit for the scalability of the CFA tunnel. As the network gets larger, the delay of the crossbar grows quickly, thereby increasing the cost of each memory access. This can be contained by employing a two-layer strategy, where PEs are partitioned into clusters based on the communication traffic among them, and intra-cluster network is based on the tunnel, whereas the inter-cluster network is based on packet switch. In this paper, we focus only on the study of the flat network, which we believe is appropriate for the applications we are interested in.

4. PERFORMANCE EVALUATION FRAMEWORK

We target complex applications which are usually described in C using high-level language features such as pointer references and complex data structures. The speculated performance advantage can only be validated on such applications. A performance evaluation environment, which can simulate CFA with reasonable architectural details for any CFP applications, is therefore needed.

A good example of an architectural evaluation environment is the SimpleScalar toolset developed at Wisconsin [9]. It is designed to study new innovations in micro-architecture such as pipelining, branch prediction, out-of-order issue etc. The environment provides a complete compiler tool chain that can compile a C application into a binary in the PISA instruction set. An instruction set simulator can then be used to simulate the binary, while collecting performance metric of interest. Figure 4 (a) shows the pseudo code of *sim-safe*, a fast simulator provided in SimpleScalar, which maintains the processor state by a simulated memory (*mem) and registers (*regs). It starts by loading the application binary into a simulated memory, and then entering a loop which fetches an instruction from the simulated memory at a time, decodes it, and then performs an action that is consistent with the instruction semantics, while updating simulated registers and memory accordingly.

In the sequel, we first introduce how the SimpleScalar infrastructure is extended into a multi-processor, CFA performance evaluation environment. We then show how a C program is mapped into a CFA in our environment by a simple, yet complete example.

4.1. Sim-CFA

We consider a homogeneous CFA where each PE is implemented by a processor equipped with the PISA instruction complemented by the context-flow instruction set defined in Section 3. The processor state in a single processor environment first needs to be replicated, as shown in Figure 4 (b). While each PE has its own private address space, an unused memory space segment of each PE, from address 0x00000000 to 0x03FFFFFF, is mapped to context memory pool. With this approach, high-level language features, such as array references, pointer indirection and structure member references, can still be used directly in the source code to access objects within the context.

The simulator was modified to run multiple SimpleScalar processors simultaneously modeling the multiple threads executing in parallel on the system PEs. For this purpose, the memory space and register files were replicated, one per PE, and the main execution loop of the simulator was modified to execute one instruction from each PE code at each simulation cycle.

SimpleScalar suite provides a very useful annotation interface where unused bits in the instructions can be used to introduce new

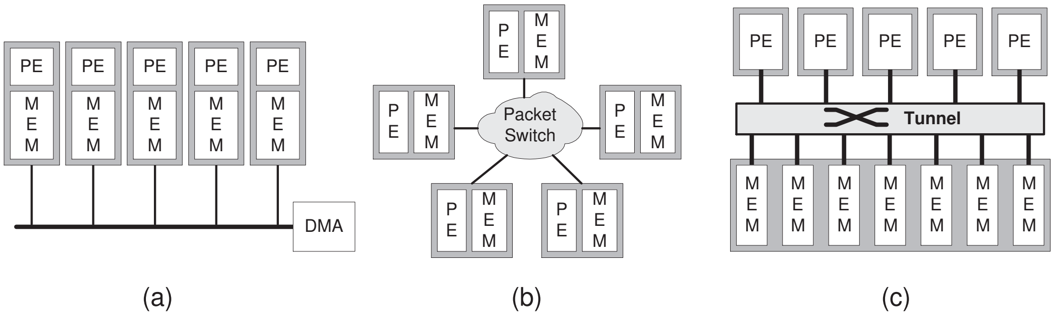

Figure 3: Alternative Implementations of Context-Flow Architectures

```
RegsType     regs;
MemSpaceType mem;

void simCore(  ) {
  /* create memory space &
   *load target program
   */
  memCreate(mem);
  loadProg(prog, mem);

  while(TRUE) {
    /* fetch next instruction
     * to execute
     */
    inst = Fetch(mem, reg.PC);

    /* decode, execute, and
     * commit the instr
     */
    switch (  opcode(inst)  )
    {
      case ADD: perform_add;
      case SUB: perform_sub;
                  ⋮
    }
    /* go to next instr */
    reg.PC=reg.NPC; reg.NPC++;
  }
}
```

(a)

```
RegsType     regs[NUM_OF_PES];
MemSpaceType mem [NUM_OF_PES];

void simCore(  ) {
  /* create memory space &
   * load target program */
  for( each PE p ) {
    memCreate(mem[p]);
    loadProg(prog[p], mem[p]);
  }

  while(TRUE) {
    for( each PE p ) {
      /* fetch ... */
      inst = Fetch(mem[p], reg.PC[p]);

      /* decode, execute, and commit */
      if( annotated(inst) )
        switch ( annotation(inst) ) {
          case RPC: perform RCP;
          case AllocBank: perform alloc;
                             ⋮
        }
      else if (memAccess(inst) &&
               addr<0x04000000)
        access context flow memory banks;
      else { /* normal code */
        switch (  opcode(inst)  )
          case ADD: perform add;
          case SUB: perform sub;
                             ⋮
        }
      /* go to the next instruction */
      reg.PC[p]=reg.NPC[p]; reg.NPC[p]++;
    }
  }
}
```

(b)

Figure 4: The Original and Modified SimpleScalar Simulator Core

instructions without the change of compiler tool suite. A new instruction is defined by giving a non-zero annotation value to predefined instruction opcodes. This annotation value can be detected at runtime and interpreted by emulating the corresponding behavior. We use this feature to help introduce the context-flow instruction set to each PE. Some of the instructions will be used to implement the context-flow API (Section 2), while others will be used by the compiler described in the next section to implement RPC.

As shown in Figure 4 (b), the simulation engine starts by loading the binaries for each PE into the simulated memories. At each simulation cycle, for each PE, the simulator fetches an instruction from memory and decodes it. If its annotation field is non-zero, meaning that it is a context-flow instruction, it will invoke the corresponding on-chip network simulation to process a request on one of the ports of the network. If it is a memory access whose address falls into the range from 0x00000000 to 0x03FFFFFF, the corresponding location inside the context memory pool will be accessed. Otherwise, it will interpret the instruction the same way as SimpleScalar does.

We implemented different networks defined in Section 3, including bus based, packet switched and tunnel based. Note that at this stage of implementation, our packet switched network is very preliminary: we assume a perfect network where no congestion can ever occur (equivalent to point-to-point), which can nevertheless give the performance upper bound. Another simplification we use for now to obtain a first order approximation of heterogeneous CFA, where processing elements can be custom hardware, is to include a linear speedup number for a PE intended for ASIC, thereby getting an approximate execution time.

Our simulator collects several useful performance statistics during simulation. *Throughput* measures the rate at which CFA can accept the top-level RPC. *Utilization* measures the percentage at which the PEs are busy computing rather than idling.

4.2. Architecture Configuration and Application Compilation

```
top(float* B, int n)
{
    sqrtArray(B, n);
    addArray(B, n);
}
```

```
sqrtArray(float* B, int n)
{
  for(i=0; i<n; i++)
     B[i] = sqrt(B[i]);
}

addArray(float* B, int n)
{
  for(i=0; i<n; i++)
     B[i] += 2.0;
}
```

Figure 5: Original C Implementation of a Simple Array Processor

Consider that we need to implement an array processor that calculates $f(A) = \sqrt{A} + 2.0 : A \in R^n$. A possible traditional C implementation that breaks down the calculation into two steps is shown in Figure 5. To transform the program into a CFP, the first step is context definition. In this example, the context is simply the data array. Figure 6 presents a transformation of the source code that runs on two PEs, mapping top() and sqrtArray() to PE0 and addArray() to PE1. Procedure mappings to system PEs are defined in "config.dat" along with these procedures' stamps. This file is used to generate proxies and main functions for each PE via an automatic code generator (Figure 7). Note that the main() for each PE simply runs an infinite loop waiting for call to the procedures it implements. WAIT_FOR_RPC() and READ_2_ARGS() are simply macros that use cfiAckRPC() and cfiLoad(), respectively.

Once coded/generated, the source files of each PE along with proxies' definition are compiled by the SimpleScalar gcc compiler *ss-gcc*. Sim-CFlow then can simulate the modeled system by run-

```
sqrtArray(float* B, int n) {
  for(i=0; i<n; i++)
     B[i] = sqrt(B[i]);
}

addArray(float* B, int n) {
  cfiRPC(ADD_ID);
}

top(float* B, int n)
   sqrtArray(B, n);
   addArray(B, n);
}

main() {
  while(1) {
    WAIT_FOR_RPC();
    if(callee==TOP_ID) {
      READ_2_ARGS(A, n);
      top(A,n);
    } else {
      if(callee==SQRT_ID) {
      READ_2_ARGS(A, n);
      sqrtArray(A,n);
    }
  }
}
```

PE0

```
addArray(float* B,int n)
{
  for(i=0; i<n; i++)
     B[i] += 2.0;
}

main() {
  while(1) {
    WAIT_FOR_RPC();
    if(callee==ADD_ID) {
      READ_2_ARGS(A, n);
      addArray(A,n);
    }
  }
}
```

PE1

Figure 6: The Context-Flow Version of a Simple Array Processor

ning the generated binary files to generate detailed performance reports.

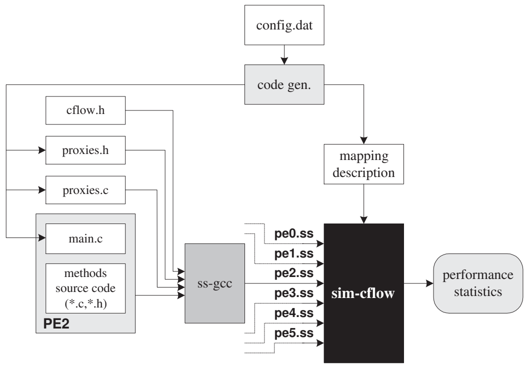

Figure 7: Sim-CFlow Simulation Process

5. TEST CASES AND PERFORMANCE RESULTS

In this section we present performance results of several architectural configurations in comparison of our proposal. Evaluations were applied to two real-life applications, namely, MPEG1-LayerIII decoder and cryptography acceleration processor. The performance evaluation framework presented in Section 4 was used to hold the experiments.

5.1. MPEG1-LayerIII Decoder

MPEG1-LayerIII, commonly referred to as MP3, is the de-facto standard of high-quality high-compression of audio data. MP3 decoders became of interest after their popular use in portable multimedia devices.

An overview of the decoder stages is presented in Figure 8. The highlighted stages were implemented in our testbench. Each

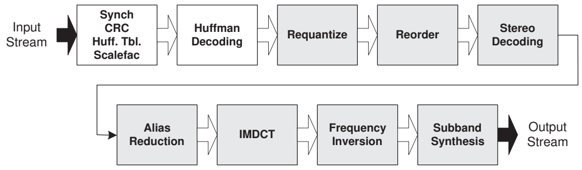

Figure 8: MP3 Decoder

stage is implemented in a single procedure processing one data granule at a time. Procedures are grouped in PEs such that the sum of method delays within PEs are as close as possible, targeting efficient thread-level pipelining. Due to the absence of accurate hardware implementation performance numbers, the delay of each method is determined using the number of memory accesses per call, assuming a perfect pipeline implementation of the processors and that memory bandwidth is the primary bottleneck. Current datapath synthesis tools (such as Module Compiler by Synopsys) can easily pipeline the computational parts of the target algorithm.

In our experiment, each configuration uses 6 4-KBytes SRAM banks. Simulation results are shown in Table 1, where the second column reports the throughput in cycles per request. The third column reports the average PE utilization.

Architecture	Throughput	PE Util.
Context-Flow	3439	71%
Single-PE	9800	100%
Shared-bus	5944	41%
Perf. Packet Switched	5043	48%

Table 1: MP3 Decoder Results

5.2. Cryptography Acceleration Processor

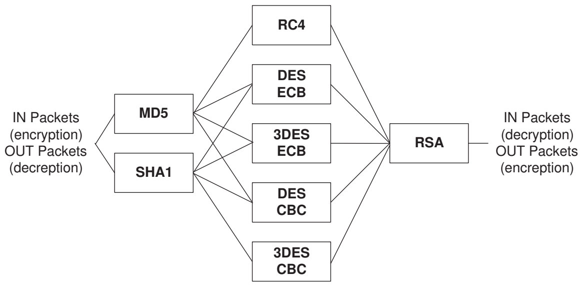

Figure 9: Crypto Accelerator Flow

Cryptography acceleration processors are becoming of central interest with the increase of SSL-based traffic over the internet. In our bench mark, we implemented a number of symmetric and asymmetric algorithms commonly used in SSL and IPSec. The implemented functions and the possible flows of packets are shown in Figure 9. Delay of processing methods were obtained from actual RTL implementations [10] and comparison results [11]. The longest path of an input packet is to go through all three categories of processing, namely hashing (MD5 or SHA1), symmetric or private-key encryption (DESECB, DESCBC, 3DESECB, 3DESCBC, or RC4), asymmetric or public-key encryption (RSA). Packets could skip hashing, public-key encryption, or both.

To carry out the experiment, we coded a packet generator that generates a packet mix which uses various processing paths according to a given distribution. A set of packets was generated and an appropriate mapping, not necessarily optimal, to a 6 PE system was used to get the results summarized in Table 2.

Architecture	Throughput	PE Util.
Context-Flow	742	65%
Single-PE	9039	100%
Shared-bus	1808	26%
Perf. Packet Switched	1156	44%

Table 2: Crypto Accelerator Results

5.3. Discussion

By looking at the evaluation results, we start by noting the importance of parallelizing the application on multiple PEs. The single PE implementation of the crypto processor is 12x slower than the tunnel-based implementation. It is also clear that the tunnel approach provides better performance when compared to alternative multi-PE configurations. A speedup of 2.43x and 1.56x were obtained by using tunnel-based architecture instead of a shared-bus and packet switch, respectively. The 56% increase in performance was achieved even though that the packet switch implementation assumes congestion-free traffic, which is usually not the case with real designs. The performance enhancement can also be viewed by comparing the average utilization of PEs, which also implies a better utilization of system memory resources. Similar results were also obtained with the MP3 application. In short, using our proposed architecture and thread-level pipelining results in a fairly large performance gain without any change in processing element designs.

6. RELATED WORK

The MIT Raw machine was one of the earliest designs to utilize on-chip interconnection networks [12]. It uses several 2-D mesh networks to connect an array of identical programmable tiles of RISC processing cores. Dally in [4] suggests the use of on-chip interconnection networks for future SOC where traditional interconnection techniques do not scale. It suggests the use of regular interconnection topologies, such as torus and mesh networks, as a means of communication between square tiles of identical dimensions, but not necessarily homogeneous. The work in [14] elaborates on this architecture targeting design exploration at the system level. Their work proposes mapping algorithms that target the power/performance optimization problems for the regular communication architecture.

The use of crossbar based interconnects started to become popular in recent years. The Berkeley IRAM [15] and Stanford Smart Memory system [16] both use a crossbar to interface a single general purpose programmable RISC PE to an array of memory banks, targeting the high bandwidth that crossbars provide. However, the high-level interface we implemented in our tunnels is not used in those systems as only a single PE is interfaced to the memory pool.

7. CONCLUSION AND FURTHER WORK

In this paper, we introduced the context-flow programming model and proposed a supporting platform architecture. A simulation environment was developed and used to evaluate the new architecture in comparison with traditional interconnection organizations. The results obtained confirm the performance improvement of context-flow architecture using real-life applications.

Several issues are still open for further investigation and development. The SimpleScalar based simulator used in our study will undergo several enhancements for a better representation of heterogeneous systems. Future work will also investigate compiler techniques for automatic code translation of system description into context-flow program.

8. REFERENCES

[1] Mark Horowitz, Ron Ho, and Ken Mai, "The future of wires," in *Proceedings of the IEEE*, April 2001, pp. 490–504.

[2] William J. Dally and Brian Towles, "Route packets, not wires: On-chip interconnection networks," in *Proceeding of the 38th Design Automation Conference*, June 2001, pp. 684–689.

[3] `http://www.systemc.org`.

[4] D. Gajski, J. Zhu, D. Doemer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Boston, March 2000.

[5] `http://www-unix.mcs.anl.gov/mpi`.

[6] *OMG Web Site*, `http://www.omg.org/`.

[7] Doug Burger and Todd M. Austin, "The SimpleScalar tool set, version 2.0," Tech. Rep., Computer Science Department, University of Wisconsin, 1997.

[8] Rudolf Usselmann, "DES/Triple DES IP cores," September 2001.

[9] Bruce Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*, John Wiley & Sons, second edition, October 1995.

[10] Elliot Waingold, Michael Taylor, Devabhaktuni Srikrishna, Vivek Sarkar, Walter Lee, Victor Lee, Jang Kim, Matthew Frank, Peter Finch, Rajeev Barua, Jonathan Babb, Saman Amarasinghe, and Anant Agarwal, "Baring it all to software: Raw machines," *IEEE Computer*, vol. 30, no. 9, pp. 86–93, September 1997.

[11] Jingcao Hu and Radu Marculescu, "Exploiting the routing flexibility for energy/performance aware mapping of regular NoC architectures," in *Proceedings of the Design Automation and Test Conference in Europe*, March 2003.

[12] David Patterson, Thomas Anderson, Neal Cardwell, Rich ard Fromm, Kimberley Keeton, Christoforos Kozyrakis, Randi Thomas, , and Kathy Yelick, "Intelligent RAM (IRAM): Chips that remember and compute," in *IEEE International Solid-State Circuits Conference*, February 1997.

[13] Drew Wingard, "Micronetwork-based integration for SOCs," in *Proceeding of the 38th Design Automation Conference*, June 2001, pp. 673–677.