Parallel Programming Models for a Multi-Processor SoC Platform Applied to High-Speed Traffic Management

Pierre G. Paulin[1], Chuck Pilkington[1], Michel Langevin[1],
Essaid Bensoudane[1], Gabriela Nicolescu[2]
[1]Central R&D, STMicroelectronics, Ottawa, Canada
[2]Ecole Polytechnique de Montreal, Montreal, Canada
pierre.paulin@st.com

ABSTRACT
In this paper, we describe the MultiFlex multi-processor SoC programming environment, with focus on two programming models: a distributed system object component (DSOC) message passing model, and a symmetrical multi-processing (SMP) model using shared memory. The MultiFlex tools map these models onto the StepNP multi-processor SoC platform, while making use of harware accelerators for message passing and task scheduling. We present the results of mapping an Internet traffic management application, running at 2.5Gb/s.

Categories and Subject Descriptors
B7.1 [**Integrated Circuits**]: Types and Design Styles – *VLSI, Advanced technologies*, *Microprocessors and microcomputers.*

General Terms
Algorithms, Design.

Keywords
System-on-chip, multi-processor systems, embedded software.

1. INTRODUCTION
The continued increase in the non-recurring expenses for the manufacturing and design of systems-on-chip (SoC) is leading to the need for significant changes to the design of SoC platforms. These factors are the drivers behind the emergence of domain-specific flexible SoC platforms [1]. The key requirement is for the effective use of platforms via high-level programming models to abstract platform details, as discussed by J. M. Paul [2].

2. SURVEY OF MP MODELS
A number of multi-processor programming models designed for SoC scale application have been presented. The MESCAL approach [3] is based on a programming model defined as a set of API's abstracting a micro-architecture. Based on this definition, in [4] the authors present a programming model that is essentially dedicated to the IXP1200 network processor.

Kiran [5] proposes a parallel programming model for communication, in the context of behavioral modeling of signal processing applications. This model integrates the message passing and shared memory communication paradigms. It exploits the advantages of both paradigms providing up to an order of magnitude improvement in the communication latency over a pure message-passing model. However, this approach only addresses communication modeling, and not the implementation.

Forsell [6] presents a sophisticated programming model that is realized through multithreaded processors, interleaved memory modules, and a high capacity interconnection network. This is based on PRAM (parallel random access machine) paradigm. However, this is restricted to a fixed network-on-chip architecture. In comparison with the systems cited above, we believe our methodology has four key contributions:

1. Interoperable message passing and SMP model support.
2. An extremely efficient implementation of these models is achieved using novel hardware accelerators for message passing, context switching and dynamic task allocation.
3. Support of homogenous programming styles for MP-SoC platforms composed of heterogeneous H/W-S/W processing elements. This is achieved via an interface definition language and compiler supporting a neutral message format.
4. All application programming is done using high-level language (C or C++, with calls to programming model APIs).

3. STEPNP MP-SOC PLATFORM
Figure 1 depicts the *StepNP* flexible multi-processor SoC architecture platform, which was described in detail in [7]. The StepNP platform includes models of standard or configurable processors, a network-on-chip, configurable H/W processing elements, as well as networking-oriented I/O's. Aside from these domain-specific I/O's, this is a general-purpose platform.

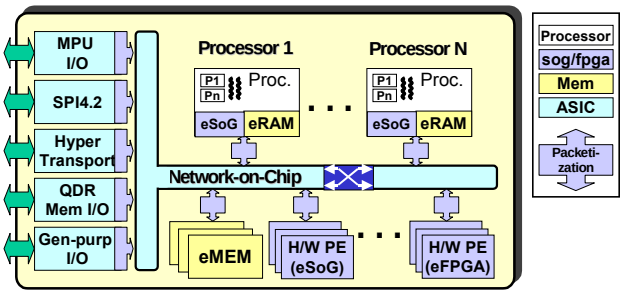

Figure 1. ***StepNP*** **MP-SoC Platform**

CODES+ISSS'04, September 8-10, 2004, Stockholm, Sweden.

The StepNP platform makes a very important assumption on the interconnect topology: namely, it uses a single interconnect channel that connects all I/O and processing elements. An orthogonal, scaleable, interconnect approach with predictable bandwidth and latency is essential. Here, we use STMicroelectronics' interconnect technology generation framework [9], which supports out-of-order and split-transactions. A common issue with all NoC topologies is communication latency [10]. Effective latency hiding is therefore key in achieving efficient parallel processing. For this reason, the StepNP platform includes models of *hardware multithreaded* processors [8]. Multi-threading lets the processor execute other streams while another thread is blocked on a high latency operation.

4. PROGRAMMING MODELS

The *MultiFlex* application development environment was developed specifically for multi-processor SoC systems. Our previous work addressed modeling, debug and analysis [8]. The developments described here address the automatic mapping of high-level applications onto MP-SoC platforms. Two parallel programming models are supported in the MultiFlex system. These models are inspired by leading-edge approaches for large distributed systems development, but adapted and constrained for the SoC domain:

- Distributed System Object Component (*DSOC*) model. This model supports heterogeneous distributed computing, reminiscent of CORBA and Microsoft DCOM distributed component object models. It is a message-passing model and it supports a very simple CORBA-like interface definition language (dubbed *SIDL* in our system).
- Symmetric multi-processing (*SMP*), supporting concurrent threads accessing shared memory. The SMP programming concepts used here are similar to those embodied in Java and Microsoft C#. The implementation performs scheduling, and includes support for threads, monitors, conditions and semaphores.

Both programming models have their strengths and weaknesses. In the MultiFlex system, both can be combined in an interoperable fashion, depending on the application requirements, as will be demonstrated in the traffic manager.

5. DSOC PROGRAMMING MODEL

The DSOC programming model relies on a high-level representation of parallel communicating objects, as illustrated in the simple example of Figure 2, where the four objects represent application functions. Each DSOC object has a language-neutral SIDL interface. As illustrated in Figure 2, the DSOC objects can be assigned to general-purpose processors running a standard operating system (e.g. for Object 2), to multiple hardware multi-threaded processors (Objects 1 and 3), or to hardware processing elements (Object 4).

Due to the underlying heterogeneous components involved in the implementation of the inter-object communication, a translation to a neutral data format is required. In the MultiFlex system, this is achieved with an SIDL compiler. In this context, the use of SIDL is similar to the Java Remote Method Invocation philosophy, where object interfaces are defined in terms of a Java interface. Similarly, SIDL looks much like a pure virtual C++ class. As explained below, the use of SIDL is key to the message passing implementation.

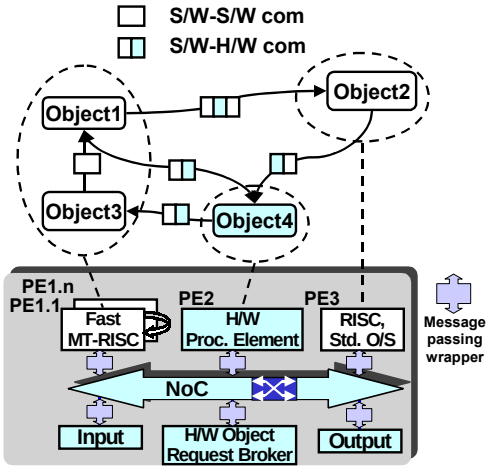

Figure 2. DSOC Model to Platform Mapping

Our implementation of the DSOC programming model relies on two key services. As we are targeting this platform at high performance applications, a key design choice is the implementation of these services in hardware.

- The hardware *Message Passing Engine* (MPE) is used to optimize inter-process communication. It translates outgoing messages into a portable representation, formats them for transmission on the network-on-chip, and provides the reverse function on the receiving end.
- The hardware *Object Request Broker* (ORB) engine is used to coordinate object communication. As the name suggests, the ORB is responsible for brokering transactions between clients and servers.

5.1 DSOC Message Passing Implementation

In the MultiFlex system, a compiler is used to process the SIDL object interface description, and generate the client or server wrappers that are appropriate for the (H/W or S/W) processing element sending/receiving a message. For processors, the SIDL compiler generates the low-level communication software driving the message passing hardware. For hardware processing elements, the compiler generates the data conversion hardware and links it to the NoC interface.

The end result is that the client wrapper takes client calls, and, with the help of the message passing engine, "marshals" the data into a language neutral, portable representation. This marshaled data is transferred over the NoC to the server wrapper. The server wrapper "unmarshals" the data, and invokes the server object implementation. The return values are then marshaled, sent back to the client, and unmarshaled in a similar way. Due to the hardware support for message passing, the software overhead for the remote invocation is a few dozen instructions. Note that no software context switching is done. If the server method returns a result, the client hardware thread is stalled until the result is ready.

5.2 The DSOC Object Request Broker

The role of the ORB in parallel execution and system scheduling is key. Parallel execution in a DSOC application is achieved using one or more of the following mechanisms:

- Many client objects may execute in parallel.
- A service may be load balanced over a number of resources.
- A client call may return to the client before the server has completed the request, allowing both client and server to execute in parallel.

Simply stated, the approach used here involves replicating object servers over a "server farm". The ORB matches client requests for service with a server, according to some criteria. In the current implementation, the least loaded server is selected.

In our approach, logical threads are mapped one-to-one to the physical threads of the hardware multi-threaded processing elements. This may seem like a limitation, but on the other hand, even fairly small systems we envisage have 64 hardware threads or more (e.g. 8 processors with 8 threads each). In actual systems we have designed, the limitation on the number of threads has not proved to be an issue (indeed, until recently, some Unix workstations had process table restrictions limiting the number of processes to under 256).

As a result of our mixed HW/SW implementation, the software overhead for a complete DSOC call is very low. For example, a call with a few integer parameters takes less than 50 instructions for a complete round trip between client and server. This includes:

- Call from the client to the server proxy object.
- Insertion of the call arguments into message passing engine.
- Retrieval of the arguments at the server side.
- Dispatching of the call to the server object.
- Insertion of the result into the server message passing engine.
- Reading of the results by the server proxy from the client message passing engine.
- Return from the server proxy object to the client with result.

The client-side code emitted by the SIDL compiler for the above looks much like a normal function call. However, instead of pushing arguments on a stack, the arguments are pushed into the MPE. Instead of a branch to subroutine instruction, a special MPE command is given to trigger the remote call. If the object call returns a result, the client thread is stalled until the request is serviced. No special software accomplishes the stall, rather the client immediately reads the return result from the MPE, and this read stalls the client thread until results are ready. All this is inlined, so the client side DSOC code can be a handful of assembler instructions.

The server-side code is slightly more complex, as it first reads an incoming service identifier and function identifier from the MPE. It then does a table lookup, and branches to the code handling this object method. This is implemented in less than a dozen RISC instructions, typically. From there, arguments are read from the MPE, and the object implementation is called. Finally, results (if any) are put in the MPE for transmission back to the client. Again, the overhead for this is roughly the same as a local object method call.

The end result of this HW/SW architecture is that we are able to sustain end-to-end DSOC object calls from one processor to another, at a rate of about 35 million per second, using 500MHz RISC-style processors.

6. SMP PROGRAMMING MODEL

Modern languages such as Java and C# support both tightly coupled SMP-style programming (with shared memory, threads, monitors, signals, etc.), as well as support for distributed object models, as described above. Unfortunately, SoC resource constraints make languages such as Java of C# impractical for high-performance embedded applications. For example, in a current STMicroelectronics multi-media application, the entire "operating system" budget is less than 1000 instructions. As we have seen in the previous section, DSOC provides an advanced object-oriented programming model that is natural to Java or C# programmers, with essentially no operating system software or language run-time. Next, we will describe how we support a high-level SMP programming model in the same resource-constrained environment.

SMP functionality in the MultiFlex system is implemented by a combination of a lightweight software layer and a hardware *Concurrency Engine* (CE). The SMP access functions to the concurrency engine are provided by a C++ API. It defines classes and methods for threads, monitors (with enter/exit methods), condition variables (with methods for signal and wait), etc.

The CE appears to the processors as a memory-mapped device, which controls a number of concurrency objects. For example, a special address range in the concurrency engine could correspond to a monitor, and operations on the monitor are achieved by reading and writing addresses within this address range. Most operations associated with hundreds or thousands of instructions on a conventional SMP operating system are accomplished by a single read or write operation to a location in the concurrency engine.

To motivate the need for a hardware concurrency engine, consider the traditional algorithm for entering a monitor. This usually consists of the following:

1. Acquire lock for monitor control data structures. This is traditionally done with some sort of atomic test and set instruction, with a spin and back-off mechanism for heavily contested locks.
2. Look at busy flag of monitor. If clear, the thread can enter the monitor. If the busy flag is set, the thread must: a) link itself into a list of threads trying to enter the monitor, b) release the lock for the monitor, c) save the state of the calling thread (e.g., CPU registers) and switch to another thread.

This control logic is quite involved. In contrast, with the MultiFlex concurrency engine, entering a monitor, or signaling a condition, is done with one memory load instruction at a special address in the concurrency engine that indicates the monitor object index and the operation type. Similarly, forking up to 8192 (2^{13}) threads at a time can be accomplished with one memory write. The atomic maintenance of the linked lists, busy flag indicators, timeout queues, etc., is done in hardware.

Any operation that should block the caller (such as entering a busy monitor) will cause the concurrency engine to defer the response to the read until the blocking condition is removed (e.g., by the owner of the monitor exiting). This causes suspension of

execution of the hardware thread. The split-transaction nature of the StepNP interconnect makes this possible, since the response to a read request can be delivered at any time in the future, and does not block the interconnect. Therefore, the response to a read from a CE location representing a monitor entry will not return a result over the interconnect until the monitor is free.

Notice that no software context switching takes place for concurrency operations. The hardware thread is simply suspended, allowing other hardware threads enabled on the processor to run. This can often be done with no “bubbles” in the processor hardware pipeline. The large number of system hardware threads would make software context switching unnecessary for most applications.

The concurrency engine is also responsible for other tasks such as run queue management, load balancing, etc. Our experiments to date indicate that a simple first-come first-served task scheduling H/W mechanism results in excellent performance with good resource utilization.

Therefore, the MultiFlex system provides a SMP programming model with essentially no "operating system" software, in the conventional sense. The C++ classes controlling concurrency are implemented directly with in-line read/write instructions to the concurrency engine. A C-based POSIX thread (‘p-thread’) API is also available, with only slightly lower efficiencies.

This high-performance SMP implementation simplifies programming for the application developer. With conventional SMP implementations, the cost of forking a thread, synchronizing, etc, must be carefully considered, and balanced against the granularity of the task to be executed. Making the task granularity too large can reduce opportunities for parallelism, while making tasks too small can result in poor performance, due to SMP overhead. Finding the right balance requires a great deal of trial and error. However, with the high performance MultiFlex SMP implementation the trade-off analysis is much simplified.

7. H/W SUPPORT FOR RTOS

The overhead of a software-only RTOS context switch is typically over one thousand cycles, and in the context of MP-SoC’s with long NoC latencies, can exceed ten thousand cycles [10]. The use of hardware engines for message passing, context switching and task scheduling are therefore essential in both the DSOC and SMP model implementations in order to achieve efficient mapping of small- to medium-grain parallel tasks onto MP architectures.

For example, Mooney et al [11] have reported an experiment where RTOS context switch times take 3218 cycles and communication takes 18944 cycles, for a medium-grained computation taking 8523 cycles. This results in application execution efficiency of only 30%. They also experimented with a partial hardware acceleration for scheduling and lock management, and observe efficiencies approaching 63%. However, they note the improvement due to hardware acceleration of the scheduling and synchronization was limited by the software context switch overheads.

Kohout et al perform RTOS task scheduling in H/W and demonstrate up to 10X speedup of RTOS processing time (from 10% to 1% overhead). Absolute times for RTOS processing are not given, but the interrupt response time portion was between 1400 to 2200 ns, for a clock cycle of 200 MHz [12].

In the MultiFlex system running on StepNP (assuming the same 200 MHz clock frequency), context switches occur in 5 ns (1 clock cycle), message passing in less than 200 ns (i.e. 20 to 40 instructions typically), and scheduling of DSOC and SMP objects in less than 200 ns (20 to 40 instructions). More importantly, it is the combination of the three accelerator engines and H/W multi-threaded processors that enables the effective mapping of medium- to fine-grain parallelism onto MP architectures like StepNP. In the traffic manager application examples below, we are dealing with fine-grained parallel tasks that represent less than 500 RISC instructions typically. Finally, since the StepNP architecture hides latency with H/W threads rather than caches, there are no cache-related overheads in context-switching.

8. A TRAFFIC MANAGER APPLICATION

To illustrate the concepts discussed in this paper, we have mapped a MultiFlex model of the IPv4 packet traffic management application of Figure 3 a) onto the StepNP multi-processor platform instance depicted in Figure 3 b). In contrast with the simpler IPv4 packet forwarding application presented in [7], this application has the following challenges:

- While there is natural inter-packet parallelism, there are also numerous inter-packet dependencies to account for, due to packet queuing and scheduling for example.
- The intra-packet processing consists of a dozen tasks with complex data dependencies. Also, these tasks are medium- to small-grain (less than 500 RISC instructions).
- All user-provided task descriptions are written in C++. No low-level C or assembly code is used. The DSOC and SMP runtime support code is also entirely written in C++.

We will demonstrate that, in spite of these constraints, the MultiFlex tools support the efficient mapping of high-level application descriptions onto the StepNP platform. Packet processing at 2.5Gbps implies that for 54 byte packets (worst case), the processing time per packet is less than 100 clock cycles (at 500MHz). As the traffic manager requires over 750 RISC instructions (compiled from C++), this implies a lower bound of 8 processors (at one instr./cycle), nearly fully utilized.

8.1 Traffic Manager Functionality

A packet traffic manager is a functional component located between a packet processor and the external world, which can be a switch fabric. We assume the packet processor is performing header validation/modification, as well as address lookup and classification.

A SPI (System Packet Interface) is used to interface with the application, both as input and output. Such interface, as the SPI4.2, can support a bandwidth of 10Gbps, where packets are transmitted as sequence of fixed-size segments interleaved between multiple logical ports (up to 256 ports for SPI4.2).

The main functions of the traffic manager are packet reassembly and queuing from input SPI segment, packet scheduling, rate shaping, packet dequeuing, and SPI output segmentation.

Typically, the queues are implemented as linked-lists of fixed-size buffers, and large queues are supported using external memories. SRAMs are used to store the links and DRAMs for the

buffer content. We assume in the following that both the SPI segment size and buffer size are 64 bytes.

8.2 DSOC Model

A DSOC model of the traffic manager application is depicted in Figure 3 a). This model is composed of the following tasks:

- *IngSPI*: input SPI protocol
- *IngSeg*: temporary buffer for input SPI segment
- *DataMgr*: interface to link-buffer data storage
- *QueMgr*: link-list management, supporting N lists for packet reassembly and N*C lists for packet queuing, where N is the number of SPI ports, and C the number of traffic classes.
- *IngPkt*; packet reassembly and queuing
- *SchPkt*: packet scheduling (strict priority per port)
- *EgrPkt*: packet dequeuing and segmentation
- *ShPort*: output port rate-shaping
- *EgrSeg*: temporary buffer for output SPI segment
- *EgrSPI*: output SPI protocol

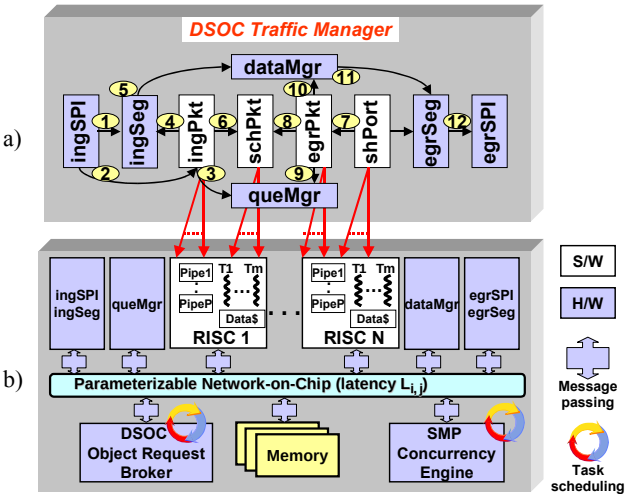

Figure 3. Application Platform

Each task is a parallel DSOC object (whose internal function is described in C++). The object granularity is user-defined. The arrows in Figure 3 a) represent the object calls between the DSOC objects. The object invocations are summarized as follows:

Ingress direction: 1) *ingSPI* invokes *ingSeg* to buffer segment; 2) at end-of -segment, *ingSPI* invokes *ingPkt* to manage a segment; 3) *ingPkt* invokes *queMgr* to push a buffer in the queue associated with the segment input port; 4) *ingPkt* invokes *ingSeg* to forward the segment to the address associated with the pushed buffer; 5) *ingSeg* invokes *dataMgr* to store the segment; 6) at the end-of-packet, *ingPkt* invokes *queMgr* to append the packet (input queue) in its associated output queue, and invokes *schPkt* to inform about the arrival of a new packet.

Egress direction: 7) *shPort* invokes *egrPkt* to request a segment for an output port; 8) at end-of-packet, *egrPkt* invokes *schPkt* to decide from which class a packet need to be forwarded for a given output port; 9) *egrPkt* invokes *queMgr* to pop a buffer from the queue associated with the output port and scheduled class; 10) *egrPkt* invokes *dataMgr* to retrieve the buffer content of the pop buffer; 11) *dataMgr* invokes *egrSeg* to store the segment; 12) *egrSeg* invokes *egrSPI* to output the segment.

8.3 StepNP Target Architecture

The application described above is mapped on the StepNP platform instance of Figure 3 b). In order to support wire-speed network processing, a mixed H/W and S/W architecture is used. The use of DSOC objects, combined with the SIDL interface compiler, allows easy mapping of tasks to H/W or S/W.

The simple but high-speed *ingSPI/ingSeg*, *egrSeg/egrSPI, queMgr* and *dataMgr* tasks are mapped onto H/W. A similar partition is used for the Intel IXP network processor [13]. The remaining blocks of the DSOC application model are mapped onto S/W. Multiple instances of each of these blocks are mapped on processor threads in order to support a given wire-speed requirements. For the output processing, in order to use processor local memory to store output status, each instance of the *schPkt*, *egrPkt* and *shPort* are associated with a disjoint subset of ports.

The platform was configured with the following parameters:

- RISC processor ISA: ARM v4
- Number of processor pipeline stages: 4
- Processor clock frequency: 500 MHz
- Number of H/W threads (per processor): 8
- One-way NoC latency (jitter): 40 +/-10ns

Using a configuration with 16 ports and 2 classes, and a mixed of packet with random length between 54B and 1024B, simulation shows that a bandwidth of at least 2.5Gbps can be supported with 7 ARM processors. However, when using short 54B packets (worst case condition), the supported bandwidth drops to 2.1Gbps. Because some of the functional blocks are mapped on a thread-per-port basis, it is not possible to simply increase the number of ARM's to support a higher bandwidth. The simplest way to achieve this is to relax the constraint on using local memory to store output port status. This is achieved by using shared-memory, as described next.

8.4 DSOC+SMP Model

A mixed DSOC and SMP model of the traffic manager application is depicted in Figure 4. The main differences with the previous model are that 1) shared-memory is used to store temporary segments, and 2) port status data is protected with semaphores.

The DSOC and SMP model is mapped to the same platform as for the DSOC model (with minor modification of the *ingSPI* and *egrSPI* H/W blocks). Using the same configuration parameters as for the reported DSOC-only experiment above, a bandwidth of 2.6Gbps is supported for the case of 54B packet using 9 ARM's. In this configuration, one ARM is used for the *shPort*, while the other 8 ARM's are used to perform any of the *ingPkt*, *schPkt* and *egrPkt* functions, as scheduled by the object request broker.

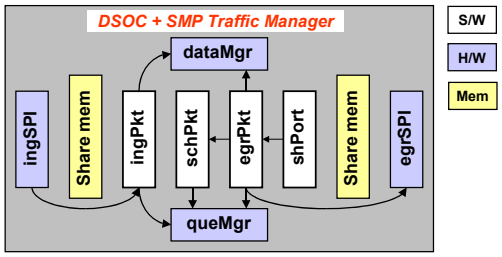

Figure 4. DSOC + SMP Traffic Manager Model

This automatic task balancing by the DSOC object request broker, combined with shared memory management using the SMP support, allow for easy exploration of different application configurations, as described next.

8.5 Experimental Results

Experimental results for different number of ports and classes are summarized in Table 1, showing the number of ARM's required to support a bandwidth of at least 2.5Gbps when using strict-priority scheduling.

We can see that increasing the number of classes requires more processing, while increasing the number of ports has almost no impact. The table also shows the bandwidth achieved using a variant of *schPkt* functionality supporting 3 class categories: 1) high-priority, 2) fair-sharing, and 3) best-effort. Fair-sharing classes are scheduled following a round-robin scheme. The table indicates that the processing impact of this scheduler is more significant when there are less supported ports.

Table 1. Experimental Results

#port	#class	#ARM	Bandwidth (Gbps)	
			strict-priority	round-robin
16	2	9	2.63	NA
16	8	9	2.54	2.33
64	8	9	2.55	2.47
64	32	10	2.54	2.44
256	32	10	2.50	2.45

The average processor utilization in all the experiments varied from 85% to 91%, allowing us to get close to the 8 processor theoretical lower bound. For the most complex scheduler (the round-robin version for 32 classes), the *egrPkt+schPkt* pair runs in 401 instructions on average. Of these, 87 instructions are needed for seven DSOC calls, or 22% of instructions. This is similar to the cost of seven procedure calls and demonstrates the importance of the fast message passing, task scheduling and context switching hardware for these fairly fine-grain tasks.

The implementation of the MultiFlex hardware O/S accelerator engines required for the traffic manager above requires 58K gates and 18K bytes of memory in total. This includes one ORB, one CE and ten MPE's. The total area for all of these is less than 0.6 mm^2 for ST's 90nm CMOS technology.

The MultiFlex technology has also been recently applied to the mapping of a high-level MPEG4 video codec (VGA resolution at 30 frames per second) onto a mixed multi-processor and hardware platform. In this case study, we demonstrated that 95% of the MPEG4 functionality can be mapped onto five simple RISC processors, running at 200MHz with an 88% average utilization rate. This demonstrates the general nature of the programming models supported. We are also currently exploring the mapping of Layer 1 modem functions for a 3G basestation.

9. SUMMARY

We have described the MultiFlex MP-SoC programming environment, which supports two parallel programming models: an object-oriented message passing model, and a symmetrical multi-processing model using shared memory. The MultiFlex tools map these models onto the StepNP multi-processor SoC platform. We presented the results of mapping an Internet traffic management application, running at 2.5Gb/s. The combined use of the MultiFlex MP compilation and allocation tools, supported by high-speed hardware-assisted messaging and dynamic task allocation, supports the rapid exploration of algorithms written using interoperable DSOC and SMP programming models, automatically mapped to a range of parallel architectures. Processor utilizations of 85% to 91% have been demonstrated.

10. REFERENCES

[1] P. Magarshack, P. G. Paulin, "System-on-Chip Beyond the Nanometer Wall", *Proc. of 40th Design Automation Conference (DAC)*, Anaheim, June 2003.

[2] J. M. Paul, "Programmers' Views of SoCs", in *Proc. of CODES/ISSS*, October 2003.

[3] K. Keutzer, S. Malik et al, "System Level Design: Orthogonalization of Concerns and Platform-Based Design", *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, Vol. 19, No. 12. December 2000.

[4] N. Shah et al, "NP-Click: A Programming Model for the Intel IXP1200", in *Proc. of Workshop on Network Processors, Intl. Symp. on High Perf. Arch.*, Feb 2003.

[5] S. Kiran, et al. "A Complexity Effective Communication Model for Behavioral Modeling of Signal Processing Application", in *Proc. of 40th Design Automation Conference,* Anaheim, June 2003.

[6] M. Forsell, "A Scalable High-Performance Computing Solution for Network on Chip", *IEEE Micro*, Sep-Oct 2002.

[7] P. G. Paulin et al, "Application of a Multi-Processor SoC Platform to High-Speed Packet Forwarding", *Proc. of DATE (Designer Forum)*, Paris, Feb. 2004.

[8] P. G. Paulin, C. Pilkington, E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors", *IEEE Design & Test of Computers*, vol. 19, no.6, Nov. 2002.

[9] See //www.stmcu.com/inchtml-pages-STBus_intro.html.

[10] A. Jantsch, H. Tenhunen (Eds.), "Networks on Chip", *Kluwer Academic Publishers*, 2003.

[11] J. Lee, V. Mooney et al, "A Comparison of the RTU Hardware RTOS with HW/SW RTOS", *Proceedings of the ASP-DAC*, Jan. 2003, pp. 683-688.

[12] P. Kohout et al, "Hardware Support for Real-Time Operating Systems", *Proc. of Codes-ISSS*, Newport Beach, Oct. 2003.

[13] E. Johnson, "IXP24/2800 Programming", Intel Press, 2003.