Parallel Programming Models for a Multiprocessor SoC Platform Applied to Networking and Multimedia

Pierre G. Paulin, *Member, IEEE*, Chuck Pilkington, Michel Langevin, Essaid Bensoudane, Damien Lyonnard, Olivier Benny, Bruno Lavigueur, David Lo, Giovanni Beltrame, Vincent Gagné, and Gabriela Nicolescu, *Member, IEEE*

***Abstract*—The MultiFlex system is an application-to-platform mapping tool that integrates heterogeneous parallel components—H/W or S/W— into a homogeneous platform programming environment. This leads to higher quality designs through encapsulation and abstraction. Two high-level parallel programming models are supported by the following MultiFlex platform mapping tools: a distributed system object component (DSOC) object-oriented message passing model and a symmetrical multiprocessing (SMP) model using shared memory. We demonstrate the combined use of the MultiFlex multiprocessor mapping tools, supported by high-speed hardware-assisted messaging, context-switching, and dynamic scheduling using the StepNP demonstrator multiprocessor system-on-chip platform, for two representative applications: 1) an Internet traffic management application running at 2.5 Gb/s and 2) an MPEG4 video encoder (VGA resolution, at 30 frames/s). For these applications, a combination of the DSOC and SMP programming models were used in interoperable fashion. After optimization and mapping, processor utilization rates of 85%–91% were demonstrated for the traffic manager. For the MPEG4 decoder, the average processor utilization was 88%.**

***Index Terms*—Multimedia computing, multiprocessor interconnection, parallel programming.**

I. Introduction

THE continued increase in the nonrecurring expenses for the manufacturing and design of nanoscale systems-on-chip (SoCs), in the face of continued time-to-market pressures, is leading to the need for significant changes to their design and manufacturing. These factors are the drivers behind the emergence of domain-specific software (S/W) programmable SoC platforms [1], [2] consisting of large, heterogeneous sets of embedded processors, reconfigurable hardware (H/W) and networks-on-chip (NoCs) [3]. However, these flexible platforms are only the foundation of a complete solution. The key requirement is the effective utilization of the platform by end-users. This implies better application programming tools, which in turn rely on high-level parallel programming models. All of these will be required to reduce the design nonrecurring expenses and improve the overall quality of the end-product.

In this context, we are working on MultiFlex, which is an application-to-platform mapping tool that integrates heterogeneous parallel components—H/W or S/W— into a homogeneous platform programming environment. This leads to higher quality designs through encapsulation and abstraction.

Manuscript received July 1, 2005; revised January 23, 2006.

P. G. Paulin, C. Pilkington, M. Langevin, E. Bensoudane, D. Lyonnard, O. Benny, B. Lavigueur D. Lo, G. Beltrame, and V. Gagné are with STMicroelectronics, Ottawa, ON K2H 8R6, Canada (e-mail: pierre.paulin@ st.com).

G. Nicolescu is with the École Polytechnique de Montréal, Montréal, QC H3C 3A7, Canada (e-mail: gabriela.nicolescu@polymtl.ca).

Digital Object Identifier 10.1109/TVLSI.2006.878259

The paper is organized as follows. Section II provides a review of SoC platform programming models and highlights our contributions in relation to other work. Section III gives the overview of the StepNP multiprocessing SoC (MP-SoC) platform. Section IV presents our vision on programming models for MPSoC. Sections V and VI explain the distributed system object component (DSOC) and symmetrical multiprocessing (SMP) programming models and their implementations in our system. Section VII introduces the hardware support for SMP and describes the interoperability of DSOC and SMP programming models. Sections VIII and IX give the experimental results obtained for two representative applications—an Internet traffic manager and an MPEG4 video encoder. Section X outlines directions for future work, and Section XI presents conclusions.

II. Review of SoC Platform Programming Models

The motivation for new high-level programmer views of SoC, and the associated languages which support the development of these views, is presented in [4]. The advantages of high-level SoC platform programming models (e.g., hiding hardware complexity, enhancing code longevity, and portability) and different programming model paradigms (e.g., message passing and shared memory) are presented in the context of NoC design [5], [6].

MESCAL (Modern Embedded Systems: Compilers, Architectures, and Languages) [7] is a large research program addressing many SoC system-level design issues. A key direction involves defining a programmers' model of a heterogeneous SoC platform, where details of the underlying architecture are abstracted, while retaining a sufficient level of control so that the application programmer can develop an efficient solution. The MESCAL work includes several case studies of how to provide a programmer's view for existing architectures. For example, a programming model inspired from Click is presented that is dedicated to the Intel IXP family of network processors [8]. Other projects under the MESCAL research umbrella have defined a small set of concurrency primitives in a shared address space (e.g., thread creation and synchronization), as well as a subset of the message passing interface (MPI) standard.

Our MultiFlex work shares many of the background motivations of MESCAL. Like MESCAL, we support a small set of concurrency primitives in a shared address space. However, we try to align more with the POSIX threads standard. Again, like MESCAL, we also support a message-passing style of parallel programming. However, whereas MESCAL may be viewed as a subset of MPI, the MultiFlex approach may be viewed as a subset of CORBA.[1] MultiFlex objects may be implemented en-

[1][Online]. Available: http://www.omg.org

tirely in hardware, with no additional software abstraction layer required. This is in contrast with the MESCAL approach of providing software layers over the underlying hardware in order to provide the necessary abstraction. Finally, the MultiFlex approach has a set of software tools and hardware implementations that are ready for commercial product development. The MESCAL research is more ambitious in scope, but is currently in the exploratory phase, and more convergence is necessary before this work can be viewed as a comprehensive solution that is directly useable by industry.

An abstract task-level interface named TTL is presented in [9]. TTL provides an SoC platform interface for implementing applications as communicating hardware and software tasks on a platform infrastructure. In the TTL framework, tasks communicate with each other through ports, which can exchange a vector of tokens of some fixed type. The communication may be blocking or nonblocking or ordered or out-of-order and may optionally allow direct access of data in the channel, avoiding unnecessary data movement. The various permutations of these options are supported, which allows the developer to trade off efficiency and portability. A threading application programming interface (API) is also provided, for parallel execution in a shared memory space.

MultiFlex and TTL both approach the heterogeneous SoC problem with an interface-centric strategy. However, MultiFlex is at a higher abstraction level, in that communication between MultiFlex objects consists of method calls which may have arbitrary type signatures, with parameter types including, but not limited to, vectors of tokens of some type. MultiFlex supports custom memory allocators, which may result in zero-copy communication for particular bindings. However, MultiFlex uses the same high-level object-oriented interface for both zero-copy and message passing communication, whereas TTL must rewrite the application to explicitly change the communication API calls. MultiFlex and TTL interfaces support dataflow type applications fairly naturally. However, TTL interfaces may be difficult to use for more complex control interfaces, due to the fixed token types. TTL assumes that both sending and receiving ends share the same data representation. This is a limiting assumption, as a future SoC will have many heterogeneous components, and data bit widths and representations will not necessarily be homogeneous. Even if they are, it is not guaranteed that different compilers will represent data structures in the same way. The MultiFlex interface definition language and communication synthesis tools avoid these TTL limitations.

Like TTL, MultiFlex communication may be blocking or nonblocking. MultiFlex also allows the option of a hardware or software broker between objects (or "tasks" in the TTL terminology), which can implement load balancing and scheduling. Like TTL, the MultiFlex communication interface may be directly supported by the SoC platform hardware. The MultiFlex threading support is more complete, with support for POSIX threading and synchronization primitives, as well as allowing the option of hardware acceleration, if required.

A parallel programming model for the communication in behavioral modeling of signal processing application is proposed by Kiran *et al.* [10]. This model, named the shared messaging model (SMM), integrates the message passing and shared-memory communication paradigms. It exploits advantages of both paradigms, providing up to an order of magnitude improvement in the communication latency over the message passing model. SMM avoids the zero-copy versus copy problem of the TTL model, in that SMM provides one API that is used by application developers, and the zero copy option is deferred to the implementation and mapping phase.

The solution is conceptually the same as the MultiFlex solution, in that custom allocators are used. In any case, TTL and SMM appear to provide the same basic abstractions, and so the advantages of MultiFlex over TLL also apply to the SMM.

Forsell [11] presents a sophisticated support for a concrete NoC (Eclipse) that is realized through multithreaded processors, interleaved memory modules, and a high-capacity interconnection network. This is based on a parallel random access machine (PRAM) paradigm, rather than the message passing paradigms of TTL and SMM. The Eclipse proposal attempts to solve the problems of PRAM approaches that are frequently cited by message-passing advocates (primarily interconnect bandwidth usage, memory coherency problems, memory consistency problems, and access latency). The claim in the Eclipse work is that message-passing approaches have problems of a poor parallel-computing models, requiring programmers to explicitly handle synchronization, data partitioning, and inter-resource communication [11]. This appears to be the standard objection to message passing (often called the "marshaling problem") that is cited by PRAM advocates.

The MultiFlex approach is to acknowledge that the PRAM and message-passing advocates both have valid viewpoints. However, rather than choosing one model over the other, the MultiFlex system supports both in a unified, interoperable framework. With some platforms, with some applications, the PRAM-derived approach will be better. In other situations, a message-passing approach will be better. The SoC platform programming model should support both approaches and allow the system developer to exercise judgment as to the most appropriate tool for the problem at hand.

The Eclipse approach does suggest interesting techniques for increasing the viability of PRAM-derived models. We believe one key technique is hardware multithreading. We also advocate this technique for MultiFlex implementation platforms, as described in Section III. The Eclipse work also presents a novel NoC proposal, which is an acyclic variant of a two-dimensional (2-D) sparse mesh. Although interesting, the power and area requirements of the proposal are not given, and thus message-passing advocates will probably remain unconvinced that significant headway has been made for PRAM programming models. In any case, most of the Eclipse work may be regarded as dealing with platform implementation issues and does not address SoC platform programming models in any depth. We assume that any programming model abstractions suitable for PRAM-derived architectures, such as POSIX threads, would be suitable for the Eclipse approach. Therefore, an Eclipse platform would be a natural target for the MultiFlex SMP style programming model primitives.

To summarize, we believe the MultiFlex approach has four key contributions in comparison with the systems cited above.

1) Interoperable distributed objects and SMP programming model support, with abstractions more inline with widely adopted industry standards (POSIX threads and distributed object systems such as CORBA).

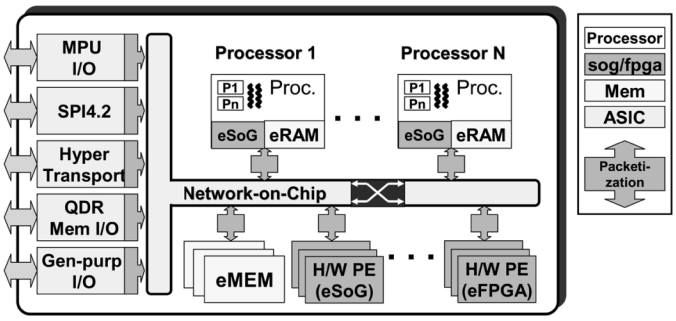


Fig. 1. *StepNP* MP-SoC Platform.

2) An extremely efficient implementation of these models is achieved using novel hardware accelerators for message passing, context switching, and dynamic task scheduling and allocation. Lower performance implementations are possible using software libraries that support the programming models.
3) Support of homogenous programming styles for MP-SoC platforms composed of heterogeneous H/W-S/W processing elements. This is achieved via a system interface definition language (SIDL) and an associated compiler, which support a neutral data format for message passing.
4) All application programming may use a high-level language (C or C++, with high-level calls to the parallel programming model APIs).

III. StepNP MP-SoC Platform

The MultiFlex programming model abstractions may be implemented on a range of SoC targets with varying degrees of efficiency. We have developed the *StepNP* [12] multiprocessor SoC (MP-SoC) architecture platform as a vehicle for exploring efficient SMP and message-passing implementations. Fig. 1 depicts the StepNP platform, which includes:

- models of (re)configurable processors;
- a NoC;
- reconfigurable H/W (embedded FPGA or embedded configurable sea-of-gates) and standard H/W;
- general-purpose and domain-specific I/Os.

In Fig. 1, the platform is shown with a communications-oriented I/O configuration. Note that, aside from the selection of domain-specific I/O, StepNP is a general-purpose, flexible MP platform.

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. This channel is referred to as an NoC. An orthogonal, scaleable, NoC interconnect approach with predictable bandwidth and latency is essential. Here, we use STMicroelectronics' STBus interconnect technology generation framework,[2] which supports a wide range of interconnect topologies, including buses, bridges, and crossbars. The STBus protocol supports similar advanced features to OCP-IP, for example, out-of-order and split-transactions. Despite the name, STBus is not a bus per se, but is in fact an NoC interconnect generation framework, which supports the automatic generation of a range of interconnect topologies made up of buses, bridges, and crossbars. The STBus toolset generates an RTL-synthesizable implementation. We have integrated the STBus SystemC models into StepNP.

A common issue with all NoC topologies is communication latency [3]. Effective latency hiding is therefore key in achieving efficient parallel processing. For this reason, the StepNP platform includes models of hardware multithreaded processors [13]. Multithreading lets the processor execute other streams while another thread is blocked on a high-latency operation. A hardware multithreaded processor has separate register banks for different threads, allowing low-overhead switching between threads, often with no disruption to the processor pipeline.

The StepNP simulation framework allows easy integration of a range of general-purpose to application-specific processor models. We have integrated public domain instruction-set models of the most popular RISC processors. The base StepNP architecture platform includes the public-domain models of the ARMv4[3] and the PowerPC (versions 603, 603a, and 604), and MIPS (32- and 64-b) instruction-set architectures.

In order to explore network-specific instruction-set optimizations, the Tensilica Xtensa configurable processor model[4] has been integrated by our academic research partners [14]. Other researchers within ST have demonstrated the use of embedded FPGA to implement user-defined instructions, therefore implementing a reconfigurable processor [1515].

For the exploration of application-specific instruction-set processors (ASIPs), we support the inclusion of instruction-set simulation (ISS) models generated from the CoWare/LisaTek ISS model generator toolset.[5] As a first demonstrator of this approach, we have developed a LisaTek-based ISS for the Xilinx MicroBlaze soft RISC processor and are currently extending it for hardware multithreading. Researchers can use this as a basis for further architecture extension or specialization.

IV. Programming Models

It is our conviction that programming model development will be evolutionary, rather than revolutionary, and the trend will be to support established software languages and technologies, rather than the development of entirely new programming paradigms. Currently, and in the foreseeable future, large systems will be written mostly in C++, Java, or languages supported by the Microsoft Common Language Runtime (CLR), such as C#. The Java and CLR-supported languages have established programming models for both tightly coupled and loosely coupled programming. Briefly stated, tightly coupled computing is done with some variant on an SMP model (i.e., threads, monitors, conditions, and signals), and heterogeneous distributed computing is accomplished with some variant on a component object model (e.g., CORBA, Enterprise Java Beans, Microsoft DCOM,[6] and its evolutions). Recent proposals for C++ evolution[7] have also called for SMP and distributed object models inside the C++ standard library specification.

The two SoC parallel programming models used in the MultiFlex system are inspired by mainstream approaches for large

[2][Online]. Available: http://www.stmcu.com/inchtml-pages-STBus_intro.html

[3][Online]. Available: http://www.fsf.org

[4][Online]. Available: http://www.tensilica.com

[5][Online]. Available: http://www.coware.com

[6][Online]. Available: http://www.microsoft.com/com/tech.DCOM.asp

[7][Online]. Available: http://www.research.att.com/~bs/C++0x_panel.pdf

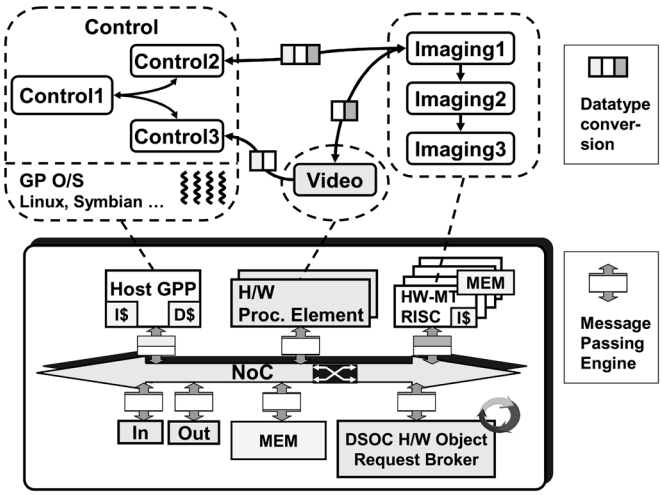


Fig. 2. DSOC model to platform mapping.

system development, but adapted and constrained for the SoC domain.

- 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 system interface definition language (dubbed SIDL in our system).
- 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, depending on the application. In the MultiFlex system, both can be combined in an interoperable fashion, as will be demonstrated further in the traffic manager and MPEG4 video encoder applications. On the one hand, this approach should be natural for programmers familiar with Java or C#, but, on the other hand, it should be sufficiently efficient (both in terms of execution efficiency and resource requirements) for use in emerging SoC devices. The following sections describe both of the MultiFlex programming models in more detail.

MultiFlex does not supply tools for extracting parallelism from sequential descriptions. In practice, this has not been an issue for the industrial multimedia, networking, and wireless applications with which we have been involved. In most cases, the application developer is deeply familiar with the inherent parallelism and is able to express this parallelism in the DSOC or SMP programming models in a matter of days or weeks—even for complex applications like MPEG4 video encoders. Relative to the total product development timeframe, this does not have a significant schedule impact.

V. DSOC Programming Model

The DSOC programming model relies on a high-level representation of parallel communicating objects, as illustrated in the simple example of Fig. 2, where the seven objects represent application functions. Each DSOC object has a language-neutral SIDL interface.

As illustrated in Fig. 2, the DSOC objects can be assigned to general-purpose processors running a standard operating system (e.g., for objects *Control1*, *Control2*, and *Control3*), or to multiple hardware multithreaded processors (objects *Imaging1* to *Imaging3*), or to hardware processing elements (object *Video*).

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 the 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, and is patterned after the `sc_interface` approach in SystemC. As explained below, the use of SIDL is key to the message passing implementation.

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

- The hardware message passing engine (MPE) is used to optimize interprocess communication. It translates outgoing messages into a portable representation, formats them for transmission on the NoC 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.
- Hardware Thread Management coordinates and synchronizes execution threads. All logical application threads are directly mapped onto hardware threads of processing units. No multiplexing of software threads onto hardware threads is done in the current implementation presented here. While the global thread management is performed by the object request broker (i.e., synchronizing and matching client threads with server threads), the cycle-by-cycle scheduling of active hardware threads on individual processors is done by the processor hardware, which currently uses a round robin scheduler. A priority-based thread scheduler is also being explored.

A. DSOC Message Passing Implementation

In the MultiFlex system, a compiler is used to process the SIDL object interface description and generate the communication 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 each hardware processing element, the compiler generates the data conversion hardware model and links it to the NoC interface.

The end result is that the initiating object wrapper takes method 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 target object wrapper. The target wrapper "unmarshals" the data and invokes the target object implementation. The return values are then marshaled, sent back to the initiator, and unmarshaled in a similar way. Due to the hardware support for message passing, the software overhead for the remote invocation is between

20–30 instructions. Note that no software context switching is done. If the target method is declared as blocking, the client hardware thread is stalled until the result is ready.

B. DSOC ORB

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

- Objects using non-blocking interfaces may execute in parallel.
- A service may be load balanced over a number of resources.
- Services may be implemented by calling other servers.

DSOC objects may be directly connected, connected in a chain (for example, to implement a data flow style of programming), connected through custom binding objects, or go through one or more object request brokers. The hardware ORB is useful if high-performance run-time load balancing is required. 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 multithreaded processing elements. This may seem like a limitation, but, on the other hand, for even fairly small systems we envisage have 64 hardware threads or more (e.g., eight processors with eight threads each). In actual systems we have designed, the limitation on the number of threads has rarely 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 the following:

- 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 is blocking, 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 of this can be inlined at compilation time, 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 ten to twelve 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.

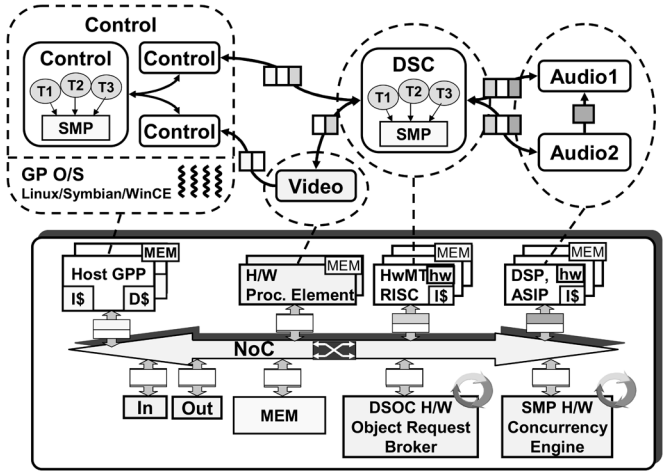


Fig. 3. Mixed DSOC-SMP model to platform mapping.

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 500-MHz RISC-style processors.

VI. SMP Programming Model

Modern languages such as Java and C# support both tightly coupled SMP-style programming (with shared memory, threads, monitors, and signals), as well as support for distributed object models, as described above. Unfortunately, SoC resource constraints make languages such as Java or C# impractical for high-performance embedded applications. For example, in a current STMicroelectronics multimedia 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), as depicted in the platform of Fig. 3. 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), and condition variables (with methods for signal and wait). A POSIX thread C binding is also available.

The concurrency engine 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 steps.

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: 1) link itself into a list of threads trying to enter the monitor; 2) release the lock for the monitor; and 3) 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, and timeout queues 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 the suspension of the 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 concurrency engine 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 blocking 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 and load balancing. 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 an 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.

This high-performance SMP implementation simplifies programming for the application developer. With conventional SMP implementations, the cost of forking a thread or synchronizing 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 performances, 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 tradeoff analysis is greatly simplified.

VII. Impact of DSOC/SMP Acceleration and Interoperation

Here, we take a closer look at the impact of the hardware-based message passing and scheduling accelerators and describe the SMP and DSOC programming model interoperability. Some of the more far-reaching system-level implications are also discussed.

A. H/W Support for Programming Models

The overheads of conventional software-based SMP and message passing programming models dramatically confine the space of potential system solutions. If these overhead roadblocks can be removed or minimized, we believe this will enable entirely new classes of parallelism, with far-reaching consequences.

The limitations imposed by SMP overheads are well recognized, and, over the years, many techniques for accelerating SMP operations have been studied. One common approach is the hardware acceleration of hardware locks. A hardware-lock solution for simultaneous multithreaded processors, and a review of related mechanisms, is presented in [16].

Another limitation is the context switch overhead. The overhead of a conventional software-only SMP context switch is typically over one thousand cycles, and in the context of MP-SoCs with long NoC latencies, can exceed ten thousand cycles [3]. Lee *et al.* [17] have reported an experiment where SMP context switch times take 3218 cycles and communication takes 18 944 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 the scheduling and the lock management, and observed efficiencies approaching 63%. However, they noted that the improvement due to the hardware acceleration of the scheduling and the synchronization was limited by the software context switch overheads.

Other approaches focus on the acceleration of task scheduling. Kohout *et al.*perform task scheduling in H/W and demonstrate up to 10× speedup of processing time (from 10% to 1% overhead). Absolute times for processing are not given, but the interrupt response time portion was between 1400–2200 ns for a clock cycle of 200 MHz [1818]. Ignios[8] provides a commercial solution for H/W accelerated task scheduling.

The results from these various H/W acceleration techniques are encouraging. However, point solutions will not realize all of the potential benefits due to problems in other areas. We therefore advocate a comprehensive set of techniques so that the benefits are realized at the system level. This not only involves H/W acceleration of context switching, SMP synchronization, and message passing; it also requires clean programming model abstraction of the H/W facilities and platform mapping tools.

In the MultiFlex system running on StepNP (assuming the same 200-MHz clock frequency), the exploitation of hardware multithreaded processors [13] supports context switches of 5 ns (1 clock cycle); message passing requires less than 200 ns (i.e., 20–40 instructions typically); and scheduling of DSOC and SMP objects is achieved in less than 200 ns (also 20–40 instructions). More importantly, it is the combination of the three accelerator engines and H/W multithreaded processors

[8][Online]. Available: http://www.ignios.com

that enables the effective mapping of medium- to fine-grained parallelism onto MP architectures like StepNP. In the traffic manager and video codec 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.

B. Interoperability of DSOC and SMP Programming Models

As discussed in Section II, MultiFlex supports both SMP and message passing programming approaches. This is in contrast to other approaches, which typically favor either one or the other. We remain agnostic as to which approach is "better," leaving this choice in the hands of the application developer. Some aspects of a system will be best solved using SMP-type abstractions. Message passing may be better for other aspects. Based on our experience, any proposed abstraction that is limited to only one of these basic approaches will be very limiting for a SoC-scale platform programming model.

An example of MultiFlex DSOC and SMP interoperability is illustrated in Fig. 3. In this example, the control, digital still camera (DSC), audio, and the video hardware objects all communicate through well defined interfaces using the DSOC message passing model described above. On the other hand, the SMP programming model is used internally by one control object and one DSC object. In the object mapping phase, the DSC objects are placed on the hardware multithreaded RISC processors, and the SMP programming model is implemented with the SMP hardware scheduler. The control objects are mapped into user mode of the general-purpose processor, so the SMP primitives are implemented using native user-level threads, e.g., POSIX threads on Linux.

The use of explicit user-visible shared memory between objects is not encouraged, as this introduces nonportability and requires additional mechanisms and abstractions to manage the shared memory between objects. However, the DSOC object binding layer may use shared memory communication internally to implement first-in first-out buffers and/or zero-copy communication, if the platform has the necessary shared memory hardware.

By default, all DSOC object invocations are single-threaded, that is, the programmer does not need to worry about multiple concurrent invocations of the methods of one server object. Of course, multiple object instances implementing the same service may execute in parallel. Also, the DSOC implementation layer may provide mechanisms to enable multithreaded invocations of the same server object, in case the developer requires this, and is prepared to deal with parallel execution inside the object.

In Sections VIII and IX, we illustrate the use of the DSOC and SMP programming models for two application domains: high-speed packet processing and video encoding. Both make significant use of programming model interoperability.

VIII. 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 Fig. 4(a) onto the StepNP multiprocessor

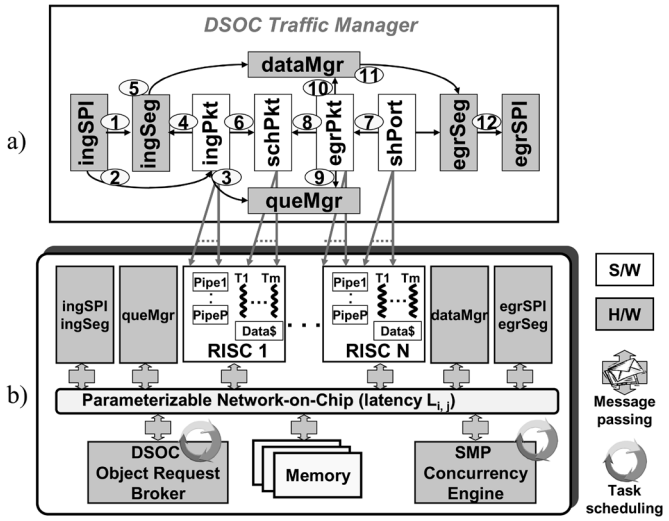


Fig. 4. Traffic manager application platform.

platform instance depicted in Fig. 4(b). In contrast with the simpler IPv4 packet forwarding application presented in [12], this application has the following challenges.

- While there is natural inter-packet parallelism, there are also numerous inter-packet dependencies to account for, due, for example, to packet queuing and scheduling.
- The intra-packet processing consists of a dozen tasks with complex data dependencies. Also, these tasks are medium- to small-grained (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 2.5 Gb/s implies that for 54 byte packets (worst case), the processing time per packet is less than 100 clock cycles (at 500 MHz). As the traffic manager requires over 750 RISC instructions (compiled from C++), this implies a lower bound of eight processors (at one instruction/cycle), nearly fully utilized.

A. 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.

An system packet interface (SPI) is used to interface with the application, both as input and output. Such an interface, for example, SPI4.2, can support a bandwidth of 10 Gb/s, where packets are transmitted as sequences 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.

B. DSOC Model

A DSOC model of the traffic manager application is depicted in Fig. 4(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*.

Each task is a parallel DSOC object (whose internal function is described in C++). The object granularity is user-defined. The arrows in Fig. 4(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; and 12) *egrSeg* invokes *egrSPI* to output the segment.

C. StepNP Target Architecture

The application described above is mapped on the StepNP platform instance of Fig. 4(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 [19]. The remaining blocks of the DSOC application model are mapped onto S/W. Multiple instances of each of these blocks are mapped onto 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 proc.): 8*;
- *One-way NoC latency (jitter): 40 ns* (± 10 *ns*).

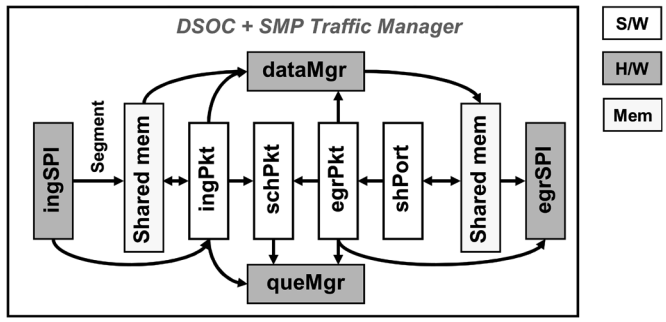


Fig. 5. DSOC + SMP traffic manager model.

TABLE I
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

Using a configuration with 16 ports and two classes and a mix of packets with random length between 54–1024 B, simulation shows that a bandwidth of at least 2.5 Gb/s can be supported with seven ARM processors. However, when using short 54-B packets (worst case condition), the supported bandwidth drops to 2.1 Gb/s. Because some of the functional blocks are mapped on a thread-per-port basis, it is not possible to simply increase the number of ARMs 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.

D. DSOC + SMP Model

A mixed DSOC and SMP model of the traffic manager application is depicted in Fig. 5. The main differences with the previous model are that: 1) shared-memory is used to store temporary segments and 2) port status data are protected with semaphores.

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

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

E. Experimental Results

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

TABLE II
MULTIFLEX O/S ACCELERATOR ENGINE COSTS

Component	# Gates	Memory	Total area
DSOC ORB	12K	6.4 KB	0.11 mm^2
SMP CE	15K	6.1 KB	0.11 mm^2
MPE (10 PE)	3K x10	0.5 KB x10	0.38 mm^2
Total (10 PE)	**57K**	**17.5 KB**	**0.60 mm^2**

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.

The average processor utilization in all the experiments varied from 85% to 91%, allowing us to get close to the eight 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 medium- to fine-grained tasks.

Table II summarizes the gate count and approximate area (using ST's 90-nm CMOS process technology) of the MultiFlex hardware O/S accelerator engines required for the traffic manager example above. This assumes ten processors (eight threads per processor). The ORB and concurrency engine support up to 128 process identifiers each.

To the best of our knowledge, there is no public-domain traffic manager implementation benchmark available, hence, we cannot perform a direct comparison of our results with published experimental results. However, the results we obtained with our own implementation of a traffic manager with a StepNP-based platform are consistent with application performance achievable with the commercial IXP2400 NPU [19], which is composed of eight RISC processors (named uEngine), clocked at 600 MHz, and each supporting eight hardware threads. For instance, it is reported that a 2.5-Gb/s packet application can be supported by the IXP2400,[9] including features such as packet segmentation and reassembly, and packet queuing and dequeuing. We believe that such results are obtained using hand-optimized assembly application coding and hand-optimized application mapping on processors; in our case, although our implementation may potentially be less optimized, the application is captured at a more abstract level, and the mapping on the platform is much simpler as a result of automatic load balancing.

IX. MPEG4 VIDEO ENCODER

The MPEG4 standard is a video codec application framework which addresses various encoding and decoding services at different bit rates and complexity, depending on the end-user application requirements. In this architecture, exploration and mapping demonstration, we are targeting consumer-oriented low-cost and low-power mobile multimedia applications.

A. Related Work

Many MPEG4 codec implementations are proposed in the literature. In [20], the authors propose a DSP-based architecture with an embedded pre-/post-processing engine. This implementation was proposed for communications applications such as video telephony. In [21], an implementation is proposed for the Philips Co-vector Processor (CVP). This implementation is designed to meet the 3G standards requirements with low power consumption. The Xilinx MPEG4 Part 2 Simple Profile Encoder core is implemented on a Xilinx FPGA.[10] Cradle Technologies[11] mapped an MPEG4 encoding version on Cradle's multiprocessor DSP (MDSP). This architecture employs multiple DSPs, RISCs, and DMAs, as well as a hierarchical memory management system. This type of architecture offers great flexibility and programmability by shifting the burden design from hardware to software, but the performance is lower than a specialized hardware implementation.

Emerging media microprocessors, such as VIRAM and Imagine, are specialized architecture to run data-parallel code [22], [23]. They offer a programmable architecture that nearly achieves the performance of special hardware for multimedia applications. These processors have the advantage of good software support while achieving good energy efficiency.

In the next sections, we describe a proprietary MPEG4 video encoder algorithm for VGA images, with a target performance of 30 frames per second (fps). This algorithm has been mapped using the MultiFlex tool on a StepNP platform instance, using the DSOC and SMP programming models.

Each of the implementations described above have their own specific service and bit-rate requirements. This makes it difficult to draw a fair comparison between the various architectures proposed. Moreover, the use of an ST-proprietary algorithm for motion estimation in our MPEG4 algorithm compounds the difficulty of a fair comparison. Therefore, rather than comparing implementations, we analyze the results and evaluate the mapping efficiency of software to the proposed architecture. Therefore our objective is to find the best match between the algorithm and the proposed architecture. To achieve this goal, we measured the load average, the execution time and the data bandwidth for various configurations. We compared the results with the achievable theoretical upper bound. The results presented on this paper are based on system level simulations; therefore, area and cost results are beyond the scope of this study.

B. MPEG4 Application and Architecture Overview

The MPEG4 codec is a good application driver to test the applicability of MultiFlex for the multimedia domain for several reasons.

1) The MPEG4 video codec exhibits a large degree of parallelism at various levels of granularity: the image is divided into slices of macroblocks, and each macroblock is divided into blocks. For most of computation processing stages, the encoding could be performed in parallel on a slice, a

[9][Online]. Available: http://www.intel.com/design/network/papers/ixp2400.htm

[10][Online]. Available: http://www.xilinx.com

[11][Online]. Available: http://www.cradle.com

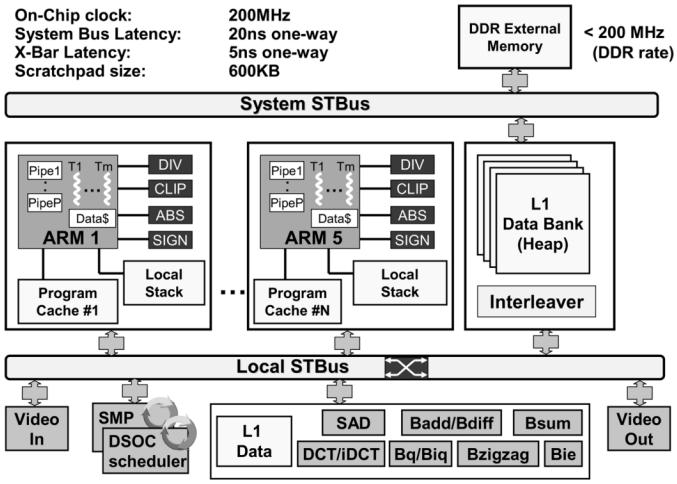


Fig. 6. Target MPEG4 video encoder MP-SoC.

macroblock or a block. Since macroblocks are particularly small, the possible number of threads working in parallel is very high (in the order of one thousand). All of these threads access the same frame buffer, making it a good testbench for our SMP accelerators.

2) Executing an MPEG4 encoder at VGA resolution at a rate of 30 fps, requires 4.08 GIPS of computational power, for ST's implementation of the encoder algorithm. This makes a full software implementation unrealistic for a low-cost consumer target market. Therefore, a mixed HW/SW implementation will be necessary.
3) The application was provided in sequential C code, which was manually parallelized using the DSOC and SMP programming models. The modifications affected the existing code minimally, showing how these programming models can be easily integrated starting from sequential code. The complete parallelization, architecture exploration (e.g., hardware, software, and memory) and optimization required less than two months for one person.

This application consists of 8068 lines of C code to be mapped on the target MP-SoC architecture shown in Fig. 6. This platform includes a variable number of multithreaded ARM processors with local memory (for the stack), a two-level instruction cache hierarchy, a shared data storage (five-way multiple bank with address interleaver), the SMP concurrency engine (to support both thread and task level parallelism), the DSOC ORB engine (to support the message passing programming model), and an interconnection channel implemented using the STBus interconnect generator. The ARM clock frequency is 200 MHz, the internal crossbar one-way latency is equal to 10 ns (two cycles) and the system bus one-way latency is 20 ns. The application is optimized during the exploration phase using special instruction coprocessors closely coupled to the ARM processors. These implement special-purpose instructions: division, value clipping, absolute value, and sign. Loosely coupled dedicated hardware coprocessors are connected to the local STBus communication channel. The choice of these hardware coprocessors is the result of initial guidance from the video platform architect, as well as the subsequent application profiling, platform mapping, and performance analysis, as explained below.

The MPEG4 encoder takes images from a frame buffer in system memory and applies step by step all the necessary operations to produce the encoded MPEG4 stream. Each frame is encoded individually, taking pixels from system memory while keeping necessary data (previous frame pixels, motion vectors, quantization coefficients, etc.) in the L1 data bank memory. The MPEG4 standard considers three types of frame encoding: I-frames, P-frames, and B-frames. I-frames contain a whole image, encoded without any reference to previous or future frames, and P-frames contain images encoded as motion vectors applied to the previous frame blocks or the difference between current and previous frame. B-frames apply the same principle but code the differences between both the previous and the next frame. Our application supports the encoding of I- and P-frames. Each frame is divided in blocks that are grouped into *macroblocks*. The encoding algorithm applies several operations sequentially.

1) Motion Estimation: if the current frame is a P-frame, it is compared with the previous one to determine motion vectors.
2) Macroblock difference, prediction and discrete cosine transform (DCT): the difference with previous frame blocks is computed, and the blocks are encoded in frequency with a DCT transformation.
3) Macroblock quantization and inverse quantization: blocks are quantized to reduce their size.
4) Macroblock inverse DCT (IDCT), zigzag, and run-level encoding: blocks are brought back from frequency to color space, ordered, and compressed with a run-level algorithm.
5) Stream generation: the output is written to a video-out device.

C. Architecture Exploration

The HW/SW architecture was achieved using a combination of approaches, including architect guidance, high-level application profiling, and a stepwise refinement process of platform mapping and performance analysis.

The sequential application was initially profiled statically, using `gprof` and `iprof` (GNU open source tools) on a Linux machine. Profiling defined a lower bound on the number of processors needed for execution: the computing power needed by the applications amounts to 4.08 GIPS. A full software solution would require a minimum of 21 ARM CPUs running at 200 MHz (each one providing at most 200 MIPS). Table III shows the results for the nine functions that take most of the execution time during the encoding of a frame. These functions represent only a very small portion of the application code (approximately 6% of 8086 lines of code), but they cover 82.9% of all computational resources needed for execution.

The first top-level HW/SW partition was proposed by the architecture design team, based on their experience with a previous MPEG4 codec design. Based on this input, the block DCT (BCDT), block IDCT (BICDT), and block sum of absolute differences (BSADs) were selected to be implemented as hardware coprocessors. These functions are present in all versions of the MPEG algorithm [24], and making them hardware blocks does not hinder the overall flexibility of the system. These blocks correspond to 44% of all computation time but less than 5% of all the application lines of code.

TABLE III
PROCESSING-INTENSIVE FUNCTIONALITIES IDENTIFICATION OF THE MPEG4 ENCODER

Function	Execution Time [%]	Lines	Fraction of full source [%]
BSAD	27,98	90	1,11
BQ	19,17	100	1,21
BDCT	10,36	80	1,11
BZIGZAG	6,22	5	0,06
BIDCT	5,70	110	1,20
BADD	4,66	15	0,18
BDIFF	3,63	17	0,21
BQI	2,59	37	0,45
BSUM	2,59	10	0,12
TOTAL	**82,9**	**464**	**5,65**

A preliminary platform mapping using these four hardware components showed that 15 processors were required to achieve 30 fps for the remaining functions mapped into software. This number was considered to be too high by the video platform architects. Previous designs had shown that an implementation with six processors or less was achievable.

In order to reduce the processor costs, the remaining five functions in the table were also assigned to hardware. The hardware blocks were modeled as timed-functional SystemC components implementing DSOC servers: at system initialization, they register with the hardware object request broker and wait for requests from the DSOC clients. Functions executed in software or hardware in the platform access these servers to perform the most computationally intensive tasks.

In this final HW/SW partition, the remaining 17% of the application computation (94% of original lines of code) remains in software. The profiling of the distributed application shows that 800 MIPS are required to run the application on the ARM processors. The data access bandwidth of these processors is 1.7 GB/s.

D. Parallelism Exploitation

To exploit the MP-SoC architecture, the application was split into parallel sections working on independent data. This parallelization phase has been optimized manually (by identifying data dependencies at very coarse-grained, e.g., image macroblocks) for the target MPEG4 application, guided by the results obtained with automatic platform mapping. The inner loops were parallelized using the fork-join constructs of the SMP programming model. Data dependencies were carefully analyzed and any modification to the reference source code has been validated against the reference data, to avoid losing the original program behavior due to some neglected dependencies. The global flow of the parallelized application makes use of a combination of the SMP and DSOC programming models.

Once the parallel independent tasks have been identified, the load balancing of the forked threads is performed dynamically by the SMP concurrency engine, which optimizes the distribution of the load over all the processing elements. The DSOC programming model is used to access the hardware accelerators.

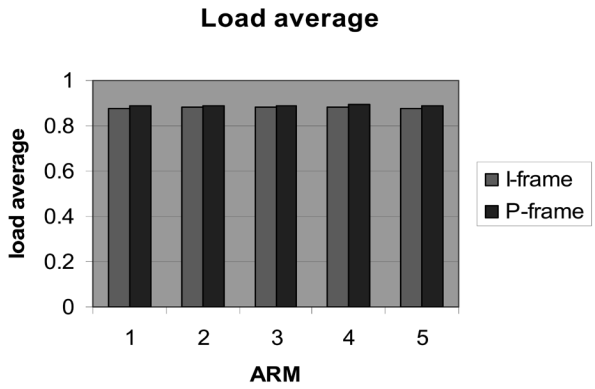


Fig. 7. Distribution of average load on processors.

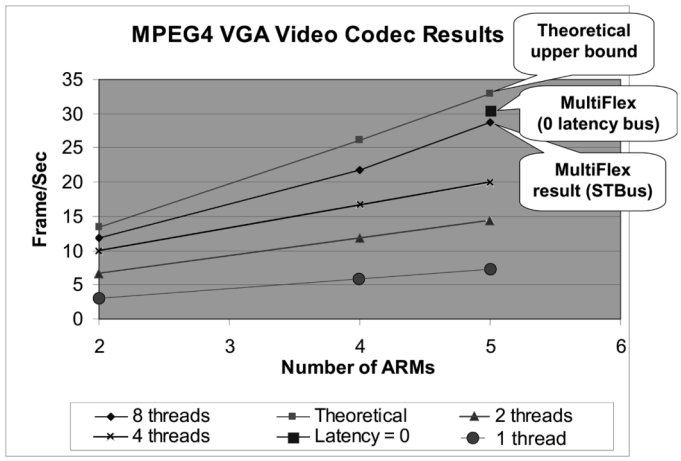


Fig. 8. Frames per second achieved for various numbers of processors.

E. Load-Balancing Results

The graph of Fig. 7 shows a well-distributed load over the five ARM processors for both I- and P-frames, thanks to the dynamic load balancing supported by the MultiFlex system. The results demonstrate the effectiveness of the combination of the application parallelization with the capabilities of the concurrency engine (CE) and object-request broker (ORB) dynamic hardware-based schedulers. The average load of all processors is 88% overall. A well-balanced workload allows a better usage of architecture resources and, therefore, better application performances.

F. Overall Performance Results

The graph of Fig. 8 summarizes the overall performance results, expressed in frames per second achieved, for a range of architecture parameters. These include the number of processors (two–five) and the number of threads per processor (one–eight). The upper curve represents the theoretical upper limit for a perfect parallelization (i.e., results for a single processor accessing local memory and then simply multiplied by the number of processors). This theoretical result does not include any interprocessor communication code and assumes zero bus latency.

This architecture uses a cycle-accurate model of the STBus channel, consisting of four parallel links. Processors models are cycle-based (with approximately $\pm 10\%$ accuracy). The models for the local memories and the multi-bank memory are cycle-accurate. Finally, the timing model of the hardware accelerators are modeled as timed-functional components, with delays based

on the actual delay figures of a previous MPEG4 codec product design.

The best result for the MultiFlex system makes use of 8 hardware threads per processor. In this case, 28.5 fps is achieved or 86% of the theoretical best result of 33 fps.

The result depicted with the small square (30 fps for five processors), is achieved using an artificial zero latency channel. This demonstrates that the multithreading is effective in absorbing most of the channel latency, as the achieved performance with real buses is very close to this 30-fps result. However, with one to four threads, the bus latency cannot be absorbed. Note that the additional cost in area of moving from a single thread core to an eight-thread core is approximately 1.5×, based on ST internal multithreaded core implementations. However, the speedup achieved is over 4× (from 7 to 28.5 fps).

The additional cost of the four message passing engines and the two schedulers is approximately 0.4 mm^2, as can be calculated using the figures in Table II of Section VIII-E, for a 90-nm process technology.

Overall, the results show a good application speed-up and a very good usage of computation resources. Moreover, the use of hardware multithreading is effective in hiding the channel latency. However, we believe that there is still room to optimize the data bandwidth and the memory accesses by exploiting data locality within the application, as discussed in Section IX-G.

G. Communication Channel Bandwidth Results

When the application is simulated on the full architecture with all caches and stack memories local to each processor, the overall bandwidth on the internal STBus channel is measured at 1.6 GB/s (close to the 1.7-GB/s results obtained using ARM-only instruction-set simulator profiling).

The throughput to the concurrency engine and the DSOC ORB schedulers was also measured. This bandwidth represents only 3.8% of the total local channel bandwidth. Therefore, the presence of these schedulers has little impact on overall bandwidth requirements.

Although this result is achievable with a four-link STBus, we consider this still an expensive solution. We are currently exploring a two-level memory hierarchy, consisting of a distributed L1 data cache architecture and a shared L2 cache. This will reduce the data bandwidth and benefit from the spatial and temporal locality of the application.

We are also exploring application functional pipelining combined with better scheduling policy. Each section of the code is partially overlapped with the others, reducing the need of memory, and letting a frame be encoded line by line. Basically, a *frame window* of a given number of lines is chosen, and each code section is executed sequentially over the frame window as it slides over the frame.

Using a combination of these optimizations, our early results show we can reduce the processor-based data bandwidth from 1.6 GB/s to less than 1.1 GB/s.

X. Future Work

The MultiFlex technology is currently being applied for a range of multimedia and wireless applications. These include the following.

- Mapping of a high-level H.264 video encoder (VGA resolution at 30 frames per second) onto a mixed multiprocessor and hardware platform.
- The exploration of the next-generation application processor architecture for ST's Nomadik mobile multimedia platform.[12] Early results of this work are presented in [25].
- Exploring the mapping of Layer1 modem functions for a 3G basestation. This includes CRC attachment, channel coding, first interleaver, second interleaver and deinterleaver, rate matching, spreading, and despreading. These base functions will be integrated with the public-domain 3G stack from the Eurecom engineering school.[13]
- The mapping of digital still camera applications such as demosaicing, using an edge-directed directional interpolation algorithm, onto an MP-SoC platform consisting of multiple configurable processors and H/W coprocessors.

Future development of the MultiFlex mapping technology includes the following.

- Support for priority-based scheduling, improved resource class management, and improved support for blocking and nonblocking message passing.
- Support for the management of logical software threads on top of available hardware-based threads. This is useful for applications with a large number of transitory logical threads or for platforms with limited number of hardware threads.
- Development of a fully software-based version of the MultiFlex schedulers for applications with coarse-grained parallelism or for existing legacy platforms that do not include the hardware schedulers and message passing engines.
- Code size reduction: For a pure software DSOC implementation, the run-time footprint for the DSOC libraries is around 1000 assembler instructions, depending on the details of the instruction set architecture (ISA) and platform configuration. Each DSOC interface method requires an additional overhead (approximately 50 instructions), again depending on the ISA and the number of parameters. We have encountered situations where this overhead was problematic. Similarly, the run time footprint of the pure-software SMP programming model libraries is around 1000 instructions, which may be too large for some applications. We are investigating ways of reducing this.
- Efficient data flow: The DSOC programming model supports dataflow programming styles. However, we have not yet implemented efficient point-to-point hardware accelerators. The current implementations all move data over the NoC through a central buffer, which may consume more power than point-to-point connections. Also, asynchronous execution and buffering support should be improved.
- Component-based programming model support: MultiFlex component support is inspired by the Fractal component model[14] and is currently in active development. This will

[12][Online]. Available: http://www.st.com/stonline/prodpres/dedicate/proc/flyer/flyer.htm

[13][Online]. Available: http://www.wireless3G4Free.com

[14][Online]. Available: http://fractal.objectweb.org

provide frameworks for composing and (re)configuring DSOC objects.

On the MultiFlex technology research front, we are working with researchers from the Politecnico di Milano on the integration of their power estimation framework [26] and dynamic voltage scaling. We are also cooperating with researchers at École Polytechnique de Montréal in the areas of configurable processors [14], token-ring-based NoC topologies, priority-based scheduling, loop fusion, loop tiling, and buffer allocation [27], and with the Université de Montréal in the area of memory architecture exploration.

XI. Summary

We have described the *StepNP* flexible SoC platform and the associated *MultiFlex* platform programming environment. The StepNP platform consists of multiple configurable hardware multi-threaded processors, configurable and reconfigurable hardware processing elements, shared memory and networking-oriented I/O, all connected via a network-on-chip (NoC). The key characteristic of the StepNP platform is that, although it is composed of heterogeneous hardware and software PEs, memories, and I/O blocks, the use of a single standardized protocol to communicate with a single global NoC allowed us to build a *homogeneous* programming environment supporting automatic application-to-platform mapping.

The MultiFlex MP-SoC programming environment supports two parallel programming models: a DSOC message passing model and an SMP model using shared memory. The MultiFlex tools map those models onto the StepNP multiprocessor SoC platform. Using this approach, the system-level application development is largely decoupled from the details of a particular target platform mapping. Application objects can be executed on a variety of processors, as well as on configurable or fixed hardware. Moreover, this approach makes use of hardware-assisted messaging and dynamic task scheduling and allocation engines that support the platform mapping tools in order to achieve low-cost communication and high processor utilization rates.

We presented the results of mapping an Internet traffic management application running at 2.5 Gb/s and a MPEG4 video encoder for VGA resolution running at 30 fps. The combined use of the MultiFlex MP-SoC mapping tools, supported by high-speed hardware-assisted messaging and dynamic task scheduling, 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%–91% have been demonstrated for the traffic manager. For the MPEG4 encoder, the average processor utilization was 88%. The low granularity of the tasks parallelized (typically less than 500 RISC instructions) highlights the importance of the efficient hardware engines used for task scheduling, context switching, and message passing.

Moreover, in the case of the MPEG4 video encoder, we have demonstrated a mixed HW/SW solution which combines flexibility and high performance. In this case, 94% of the functionality (as measured in lines of C code) is mapped onto five simple RISC processors. The remaining 6% are mapped onto regular, parallel hardware components which perform 80% of the equivalent MIPS.

Finally, the application of MultiFlex to these two very different application domains serves to demonstrate the general nature of the programming models supported.

References

[1] J. Henkel, "Closing the SoC design gap," *IEEE Comput. Mag.*, pp. 119–121, Oct. 2003.

[2] P. Magarshack and P. G. Paulin, "System-on-Chip beyond the nanometer wall," in *Proc. 40th Design Autom. Conf.*, 2003, pp. 419–424.

[3] A. Jantsch and H. Tenhunen, Eds., Networks on Chip Kluwer Academic Publishers, 2003.

[4] J. M. Paul, "Programmers' views of SoCs," in *Proc. CODES/ISSS*, 2003, pp. 156–161.

[5] L. Benini and G. De Micheli, "NoC: A new SoC paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, Jan. 2002.

[6] G. DeMicheli, "Networks on a chip," in *Proc. MPSoC*, 2003, pp. 5–36.

[7] K. Keutzer, "System level design: Orthogonalization of concerns and platform-based design," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 12, pp. 1523–1543, Dec. 2000.

[8] N. Shah, "NP-Click: A programming model for the intel IXP1200," in *Proc. Workshop Network Processors, Int. Symp. High Perf. Arch.*, 2003, pp. 100–111.

[9] P. van der Wolf, "Design and programming of embedded multiprocessors: An interface-centric approach," in *Embedded Systems Handbook*. Boca Raton, FL: CRC, 2006.

[10] S. Kiran, "A complexity effective communication model for behavioral modeling of signal processing application," in *Proc. 40th Des. Autom. Conf.*, 2003, pp. 412–415.

[11] M. Forsell, "A scalable high-performance computing solution for network on chip," *IEEE Micro*, vol. 22, no. 5, pp. 46–55, Sep.—Oct. 2002.

[12] P. G. Paulin, "Application of a multi-processor SoC platform to high-speed packet forwarding," in *Proc. DATE (Designer Forum)*, 2004, pp. 58–63.

[13] P. G. Paulin, C. Pilkington, and E. Bensoudane, "StepNP: A system-level exploration platform for network processors," *IEEE Des. Test Comput.*, vol. 19, no. 6, pp. 17–26, Nov. 2002.

[14] D. Quinn, "A system-level exploration platform and methodology for network applications based on configurable processors," in *Proc. Des. Autom. Test Eur. (DATE)*, 2004, pp. 364–369.

[15] M. Borgatti, "A 0.18 μm, 1GOPS reconfigurable signal processing IC with embedded FPGA and 1.2 GB/s, 3-Port flash memory subsystem," in *Proc. Int. Solid-State Circuits Conf. (ISSC)*, 2003, pp. 50–55.

[16] D. M. Tullsen, Supporting Fine-Grained Synchronization on a Simultaneous Multithreading Processor UCSD , CSE Tech. Rep. CS98-587, 1998.

[17] J. Lee, "A comparison of the RTU hardware RTOS with HW/SW RTOS," in *Proc. ASP-DAC*, 2003, pp. 683–688.

[18] P. Kohout, "Hardware support for real-time operating systems," in *Proc. Codes-ISSS*, 2003, pp. 45–51.

[19] E. J. Johnson, IXP2400/2800 Programming, Intel Press, Hillsboro, OR, 2003.

[20] S. Kurohmaru, "A MPEG4 programmable CODEC DSP with an embedded pre/post-processing engine," in *Proc. IEEE Custom Integr. Circuits Conf.*, 1999, pp. 69–72.

[21] B. An, "Implementation of MPEG4 on philips co vector processor," in *Proc. 14th Annu. Workshop Circuits, Syst. Process., ProRISC*, 2003, pp. 8–17.

[22] U. J. Kapasi, S. Rixner, W. J. Dally, B. Kkailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *IEEE Comput.*, vol. 36, no. 8, pp. 54–62, Aug. 2003.

[23] S. Chatterji, "Performance evaluation of two emerging media processors: VIRAM and imagine," in *Proc. Int. Parallel Distrib. Process. Symp. (IPDP'2003)*, pp. 229a–229a.

[24] D. J. Murray and W. VanRyper, Encyclopedia of Graphics File Formats O'Reilly Associates, 1996.

[25] P. G. Paulin, "Distributed object models for multi-processor SoC's, with application to low-power multimedia wireless systems," in *Proc. Des. Autom. Test Eur. (DATE)*, 2006, pp. 482–487.

[26] W. Fornaciari, F. Salice, and D. Sciuto, "Power modeling of 32-bit microprocessors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 11, pp. 1306–1316, 2002.

[27] T. Omnès, Y. Bouchebaba, C. Kulkarni, and F. Coelho, , C. Piguet, Ed., "Recent advances in low power design and functional co-verification automation from the earliest system—Level design stages," in *Low-Power Electronics Design*. Boca Raton, FL: CRC, 2004.

Pierre G. Paulin (M'82) received the B.Sc. and M.Sc. degrees in engineering physics and electrical engineering from Laval University, Quebec City, QC, Canada, in 1982 and 1984, respectively, and the Ph.D. degree in electronics engineering from Carleton University, Ottawa, ON, Canada, in 1988.

From 1988 to 1994, he was with Bell-Northern Research, the R&D arm of Nortel Networks, Ottawa, first working on high-level synthesis research and then managing an embedded software tools group. In 1994, he joined STMicroelectronics, Grenoble, France, and led the Embedded Systems Technology and System-Level Design teams. In 2000, he transferred to STMicroelectronics, Ottawa, and is currently Director of the System-on-Chip Platform Automation group. His research interests include design automation technologies for multiprocessor systems, embedded systems, and system-level design.

Dr. Paulin won the Best Presentation Award at the Design Automation Conference (DAC) in 1986, was nominated for Best Paper Awards at DAC in 1987 and 1989, and won the Best Paper Award at the International Conference on Hardware/Software Codesign and System Synthesis (CODES + ISSS) in 2004.

Chuck Pilkington received the B.Sc. degree in physics from the University of Waterloo, Waterloo, ON, Canada, in 1983, and the M.Sc. degree in electrical engineering from the University of Toronto, Toronto, ON, Canada, in 1985.

From 1985 to 2001, he worked on a range of parallel processing R&D programs in academia and industry, in cooperation with the Canadian Defence Research Establishment. In 2001, he joined STMicroelectronics, Ottawa, ON, Canada, and is currently a Senior Staff Engineer with the System-on-Chip Platform Automation Group. His research interests include SoC platform modeling and system software for high-performance parallel processing.

Michel Langevin received the B.Sc., M.Sc., and Ph.D. degrees in computer science from the University of Montréal, Montréal, QC, Canada, in 1986, 1988, and 1993, respectively.

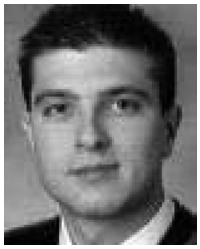
From 1993 to 1996, he was a Postdoctoral Fellow with GMD, Bonn, Germany, before joining Nortel Networks, Ottawa, ON, Canada, where he worked, until 2002, first as a System Modeler and then as a System ASIC Architect, in the specification, modeling, RTL implementation and verification of a novel Terabit/s switch architecture (U.S. patent 6307852). In 2002, he joined STMicroelectronics, Ottawa, and is currently a Senior Staff Engineer with the System-on-Chip Platform Automation Group. His research interests include system specification, hardware/software implementation, and verification.

Essaid Bensoudane received the B.Sc. degree in electrical engineering from l'Institut Polytechnique de Grenoble, Grenoble, France, in 1997, and the M.Sc. degree in automation and systems engineering from École Polytechnique, Montréal, QC, Canada, in 1999.

He joined Opal-RT from 1999 to 2001, where he worked in a range of projects related to parallel real-time systems. In 2001, he joined STMicroelectronics, Ottawa, ON, Canada, as a Research Engineer with the System-On-Chip Platform Automation Group.

Damien Lyonnard received the B.Sc. and M.Sc. degrees from the University Joseph Fourier, Grenoble, France, in 1998 and 1999, respectively, and the Ph.D. degree from TIMA/INPG, Grenoble, France, in 2003, all in physics.

He joined the System-on-Chip Platform Automation Group, STMicroelectronics, Ottawa, ON, Canada, in 2003.



Olivier Benny received the B.Eng. degree in computer engineering and M.Sc. degree in electrical engineering from École Polytechnique of Montréal, Montréal, QC, Canada, in 2001 and 2004, respectively.

He was with Interstar Technologies Inc., Montreal, from 2001 to 2002, where he was involved with product development of a distributed fax server software. In 2004, he joined STMicroelectronics, Ottawa, ON, Canada, as a System-Level Engineer, where he is currently working in the field of application mapping to multiprocessor systems.

Bruno Lavigueur received the B.Eng. degree in computer engineering and the M.Sc. degree in electrical engineering from École Polytechnique of Montréal, Montréal, QC, Canada, in 2001 and 2004, respectively.

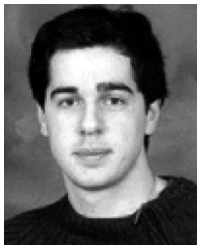
In 2004, he joined STMicroelectronics, Ottawa, ON, Canada, and is currently a Research and Development Engineer with the System-on-Chip Platform Automation Group.

David Lo received the B.Sc. and M.Sc. degrees in electrical engineering from University of Western Ontario, London, ON, Canada, in 1991 and 1994, respectively, and the Ph.D. degree in systems and computer engineering from Carleton University, Ottawa, ON, Canada, in 2005.

From 1994 to 1999, he was a Senior System Developer for a medical equipment company. In 2004, he joined the System-on-Chip Platform Automation Group, STMicroelectronics, Ottawa, ON, Canada, as an R&D Engineer.

Giovanni Beltrame received the M.Sc. degree in electrical engineering and computer science from the University of Illinois, Chicago, in 2001, the Laurea degree in computer engineering from Politecnico di Milano, Milan, Italy, in 2002, the M.Sc. degree in information technology from CEFRIEL, Milan, Italy, in 2002, and the Ph.D. degree in computer engineering from Politecnico di Milano, in 2006.

During his Ph.D. studies, he worked on analysis and optimization of multi-processor platforms. He is currently with STMicroelectonics, Ottawa, ON, Canada.



Vincent Gagné received the B.Sc. degree in computer science and mathematics and the M.Sc. degree in computer science from the University of Montréal, Montréal, QC, Canada, in 2004 and 2006, respectively.

He was with STMicroelectronics, Ottawa, ON, Canada, in 2004 as a research intern and joined the System-on-Chip Platform Automation Group as a Research Engineer in 2006.

Gabriela Nicolescu (M'03) received the degree in engineering and the M.S. degree from the Polytechnic University of Romania, Bucharest, Romania, in 1998 and "Engineer Doctor" degree from the University of Grenoble, Grenoble, France, in 2002.

She is currently an Assistant Professor with the Ecole Polytechnique of Montreal, Montreal, QC, Canada, where she teaches embedded systems design and real-time systems. Her research work is in the field of specification and validation of heterogeneous systems and multiprocessor system-on-chip design.