



HAL
open science

An Adaptive Message-Passing MPSoC Framework

Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, Michel Robert

► **To cite this version:**

Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, et al.. An Adaptive Message-Passing MPSoC Framework. *International Journal of Reconfigurable Computing*, 2009, 2009, pp.#242981. 10.1155/2009/242981 . lirmm-00373949

HAL Id: lirmm-00373949

<https://hal-lirmm.ccsd.cnrs.fr/lirmm-00373949>

Submitted on 25 May 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Distributed under a Creative Commons Attribution 4.0 International License

Research Article

An Adaptive Message Passing MPSoC Framework

Gabriel Marchesan Almeida, Gilles Sassatelli, Pascal Benoit, Nicolas Saint-Jean, Sameer Varyani, Lionel Torres, and Michel Robert

*Laboratoire d'Informatique, de Robotique et de Microélectronique de Montpellier (LIRMM),
Centre National de la Recherche Scientifique (CNRS), University of Montpellier 2, 161 rue Ada,
34392 Montpellier, France*

Correspondence should be addressed to Gabriel Marchesan Almeida, gabriel.marchesan@lirmm.fr

Received 19 December 2008; Accepted 14 April 2009

Recommended by J. Manuel Moreno

Multiprocessor Systems-on-Chips (MPSoCs) offer superior performance while maintaining flexibility and reusability thanks to software oriented personalization. While most MPSoCs are today heterogeneous for better meeting the targeted application requirements, homogeneous MPSoCs may become in a near future a viable alternative bringing other benefits such as run-time load balancing and task migration. The work presented in this paper relies on a homogeneous NoC-based MPSoC framework we developed for exploring scalable and adaptive on-line continuous mapping techniques. Each processor of this system is compact and runs a tiny preemptive operating system that monitors various metrics and is entitled to take remapping decisions through code migration techniques. This approach that endows the architecture with decisional capabilities permits refining application implementation at run-time according to various criteria. Experiments based on simple policies are presented on various applications that demonstrate the benefits of such an approach.

Copyright © 2009 Gabriel Marchesan Almeida et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

1. Introduction

The exponentially increasing number of transistors that can be placed on an integrated circuit is permitted by the dropping of technology feature sizes. This trend plays an important role at the economic level, although the price per transistor is rapidly dropping the NRE (Nonrecurring Engineering) costs, and fixed manufacturing costs increase significantly. This pushes the profitability threshold to higher production volumes opening a new market for flexible circuits which can be reused for several product lines or generations and scalable systems which can be designed more rapidly in order to decrease the Time-to-Market. Moreover, at a technological point of view, current variability issues could be compensated by more flexible and scalable designs. In this context, Multiprocessor Systems-on-Chips (MPSoCs) are becoming an increasingly popular solution that combines flexibility of software along with potentially significant speedups.

These complex systems usually integrate a few mid-range microprocessors for which an application is usually statically mapped at design-time. Those applications however tend

to increase in complexity and often exhibit time-changing workload which makes mapping decisions suboptimal in a number of scenarios. Additionally, such systems are designed in very deep-submicron technologies that bring a number of hardly predictable physical effects that, associated also to the increasing process variability, demonstrate the intrinsic and unavoidable unreliability of future nanoscale integrated systems.

These facts challenge the design techniques and methods that have been used for decades and push the community to research new approaches for achieving system adaptability and reliability (out of unreliable technology components).

This paper presents a hardware/software framework (HS-Scale platform) that is based on a set of adaptive principles which endows the architecture with some decisional capabilities. This approach helps continuously refining application mapping for optimizing various criteria such as performance or power consumption and should eventually enable fault tolerance.

This hardware/software framework is intended to permit the exploration of scalable solutions for future MPSoCs

in the context of massive on-chip parallelism with several hundreds of processing elements (PEs). Therefore, the proposed architecture relies on principles that do not imply resource sharing among processors in the broad sense of the term. The system is made of a regular arrangement of PEs that runs applications in a distributed way, exchanging messages that relate to both platform management and application data. The used programming model is derived from a popular message passing interface (MPI) system that has been augmented for supporting adaptive mechanisms.

This paper is organized as follows.

Section 2 presents the related works in the field of multiprocessor systems, programming models, and task migration techniques. Section 3 introduces the HS-Scale framework which covers the hardware, software, and the programming model used in this approach. Section 4 shows the validations in terms of both developed hardware and area utilization figures in the context of an SoC realization. Section 5 presents the results of various applications mapped on the framework emphasizing on the cost induced by the used migration techniques and the corresponding observed benefits. Section 6 draws some conclusions on the presented work and puts this in perspective with other upcoming challenges of the area.

2. Related Works

This section shortly introduces the different existing families of multiprocessor systems and puts focus on the relevant approaches that are found in the fields of scalable message passing architectures and task migration techniques.

2.1. Preliminary Considerations. Multiprocessor systems are increasingly considered as an attractive solution for accelerating computation. Parallel architectures have been studied intensively during the past 40 years; there is consequently a huge amount of books [1, 2] related to this topic, and we will therefore only focus on general concepts. The most common type of multiprocessor systems falls into the Multiple Instruction Multiple Data (MIMD) family as that defined by the Flynn's taxonomy [3]. There are two types of MIMD machine classified in accordance with their memory architecture:

- (i) shared memory architectures in which all processors share the same memory resources; therefore all changes done by a processor to a given memory location become visible to all other processors of the system;
- (ii) distributed memory architectures in which every processor has its own private memory; therefore one processor cannot read directly in the memory of another processor. Data transfers are implemented using message passing protocols.

From an architecture design point of view shared memory machines are poorly scalable because of the limited bandwidth of the memory. Existing realizations marginally use more than tenth of processors because of this reason.

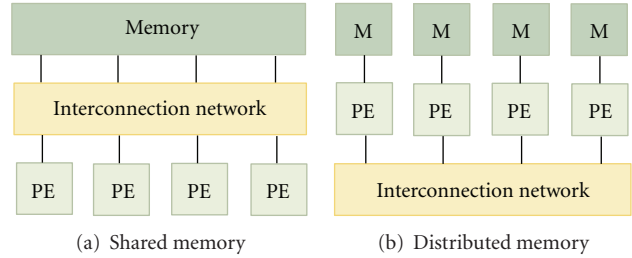


FIGURE 1: Shared x distributed memory.

Distributed memory machines are more scalable since only the communication medium may be shared among processors. There exist two families of programming models, each of which exhibits a better adequacy to one of the previously presented architecture families.

- (i) Shared memory systems require synchronization mechanisms such as semaphores, barriers, and locks since no explicit communication mechanism exists. POSIX threads [4] and OpenMP [5] are two popular implementations of the thread model on shared memory architectures.
- (ii) Distributed memory systems require mechanisms for supporting explicit communications between processes (that may be hosted on the same or different processors). Usually a library of primitives that allow writing in communication channels is used. The Message Passing Interface (MPI) [6] is the most popular standard that is used in High-Performance Computing (HPC) computer clusters for instance.

In a shared memory architecture (Figure 1(a)), processes (executed by different processors) can easily exchange information through shared variables; however it requires handling carefully synchronization and memory protection. In a distributed memory architecture (Figure 1(b)), a communication infrastructure is required in order to connect processing elements and their memories and allow exchanging information.

Since this work targets massively parallel on-chip Multiprocessor systems, scalability is a major concern in the approach. For this reason, we put focus on distributed memory machines and therefore choose a message passing programming model for it provides a natural mapping to such machines.

2.2. Message Passing for Embedded Systems. The Message passing model is based on explicit communication between tasks. This model is often used for architectures that do not provide global address space; here communications among tasks take place through messages and are implemented with functions allowing reading and writing to communication channels. CORBA, DCOM, SOAP, and MPI are examples of message passing models.

Message Passing Interface (MPI) is the most popular implementation of the message passing model, and only for this model some embedded implementations exist. The

Message Passing Interface is a specification for an API that allows many PEs to communicate through a communication network.

MPI provides a comprehensive number of primitives that relate to general-purpose distributed computing; a number of work have devised lightweight implementations supporting only a subset of the mechanisms of MPI for embedded processors and systems. This makes sense since the nature of applications for these systems is well defined, often limited to data flow applications for which Kahn Process Networks formalism offer a sufficient support that requires only blocking read operations [7]. Some MPI implementations are layered, and advanced communication synchronization primitives (such as collective, etc.) found in the upper layers make use of the simple point-to-point primitives such as `MPI_Send()` and `MPI_Receive()`. This enables using these collective mechanisms in an application-specific basis in case they prove necessary.

The specific requirements of embedded systems have led programmers to develop lightweight MPI implementations, basically built upon a subset of the original MPI mechanisms. In [8] the authors present TMD-MPI which is a lightweight MPI implementation for multiple processors across multiple FPGAs. It relies on a layered implementation which provides only 11 primitives. As no operating system is used, task mapping is static and done at design-time. Similarly, authors in [9] have also selected 11 primitives among which only 2 relate to point-to-point communications. Finally, in [10] authors present eMPI which also uses the simplest low-level point-to-point communication primitives in a layered style.

2.3. Task Migration Support. Task migration techniques have been mainly studied in contexts that fall in one of the following areas.

- (i) General purpose computing, involving usually a single computer made of several processors or processor cores. Such systems are usually built around shared memory architectures.
- (ii) High-Performance Computing (HPC) computer clusters. Such systems are usually of distributed memory type and therefore generally use message passing programming style.
- (iii) Multiprocessor embedded systems, which may make use of either shared or distributed memory architecture.

For shared memory systems such as today's multicore computers, task migration is facilitated by the fact that no data or code has to be moved across physical memories: since all processors are entitled to access any location in the shared memory, migrating a task comes down to electing a different processor for execution. There exist several efficient implementations on general purposes OS such as Windows or Linux [11].

In the case of multiprocessor-distributed memory/message passing architectures, both process code and state have to be migrated from a processor private memory to another, and synchronizations must be performed using

exchanged messages such as in [12] which targets Linux computer clusters. Some other approaches aimed at augmenting MPI for providing a support for process migration, such as [13, 14]. In [15] users present similar features based on a JAVA MPI framework that provides hardware independence; they show that despite migrating tasks imply overheads, which are in the order of seconds, significant speedups can be achieved. All these approaches target computer clusters with the typical resources of general-purpose computers and are therefore hardly applicable to MPSoCs.

Task migration has also been explored for MPSoCs, notably based on locality considerations [12] for decreasing communication overhead or power consumption [16]. In [17], authors present a migration case study for MPSoCs that relies on the Clinux operating system and a check pointing mechanism. The system uses the MPARM framework [18], and although several memories are used, the whole system supports data coherency through a shared memory view of the system.

In [19] authors present an architecture aiming at supporting task migration for a distributed memory multiprocessor-embedded system. The developed system is based on a number of 32-bit RISC processors without memory management unit (MMU). The used solution relies on the so-called "task replicas" technique; tasks that may undergo a migration are present on every processor of the system. Whenever a migration is triggered, the corresponding task is respectively inhibited from the initial processor and activated in the target processor. This solution induces a significant memory overhead for every additional task and therefore falls beyond the scope of this paper. Finally, to the best of our knowledge, no other work combines the use of a message passing programming model, on-chip multiprocessor system, and transparent decentralized automated task migration.

3. HS-Scale

The key motivations of our approach are scalability and self-adaptability; the system presented in the rest of this paper is built around a distributed memory/message passing system that provides efficient support for task migration. The decision-making policy that controls migration processes is also fully distributed for scalability reasons. This system therefore aims at achieving continuous, transparent, and decentralized run-time task placement on an array of processors for optimizing application mapping according to various potentially time-changing criteria.

3.1. System Overview. The system is based on an array of compact general-purpose PEs interconnected through a packet switching Network-on-Chip. The HS-scale system is a purely distributed memory system which is programmed using a simple message passing protocol. Contrary to MPI, processes must not be mapped to a given processor but shall freely move in the system according to user-definable policies that may aim at optimizing a given property in the system, such as performance or power consumption. Both hardware and software resources are intended to be minimalist for

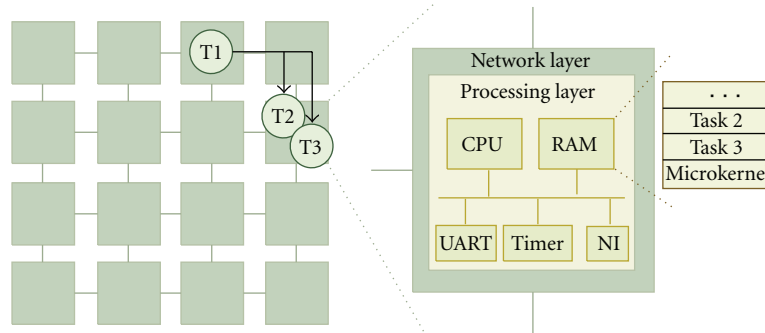


FIGURE 2: NPU structural description.

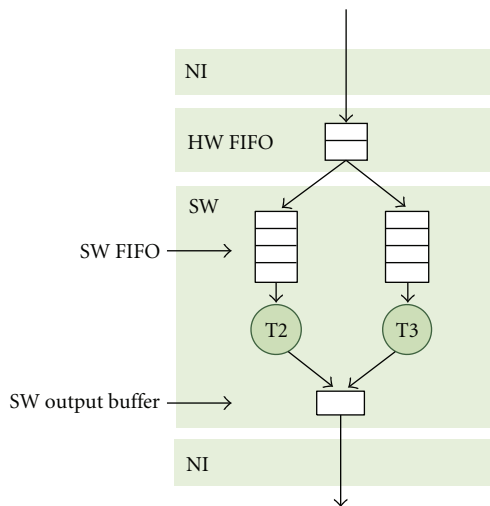


FIGURE 3: NPU functional description.

favoring compactness of processors and therefore encouraging massive parallelism. Also, for scalability reasons, there exists no master in the system unless a given application requires it.

3.2. Hardware Structure

3.2.1. Network Processing Unit. The architecture is made of a homogeneous array of Processing Elements (PEs) communicating through a packet-switching network. For this reason, the PE is called NPU, for Network Processing Unit. Each NPU, as detailed later, has multitasking capabilities which enable time-sliced execution of multiple tasks. This is implemented thanks to a tiny preemptive multitasking Operating System which runs on each NPU. The structural and functional views of the NPU are depicted in Figures 2 and 3, respectively.

The NPU is built around two main layers, the network layer and the processing layer. The Network layer is essentially a compact routing engine (XY routing). Packets are read from incoming physical ports, then forwarded to either outgoing ports or the processing layer. Whenever a packet header specifies the current NPU address, the packet is forwarded to the network interface (NI in Figure 3). The

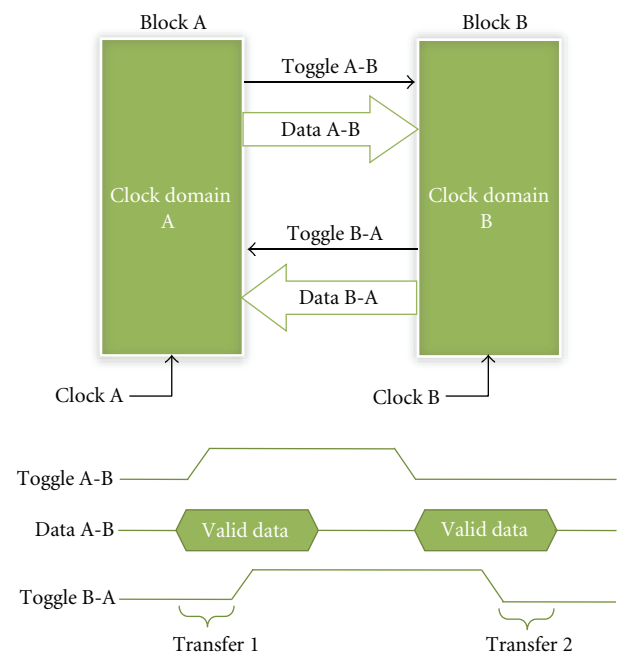


FIGURE 4: The asynchronous toggle protocol.

network interface buffers incoming data in a small hardware FIFO and simultaneously triggers an interrupt to the processing layer. The interrupt then activates data demultiplexing from the single hardware FIFO to the appropriate software FIFO as illustrated in Figure 4.

The processing layer is based on a simple and compact RISC microprocessor, its static memory, and a few peripherals (one timer, one interrupt controller, one UART) as shown in Figure 2. A multitasking microkernel implements the support for time-multiplexed execution of multiple tasks.

The processor used has a compact instruction set comparable to an MIPS-1 [20]. It has 3 pipelines stages, no cache, no Memory Management Unit (MMU), and no memory protection support in order to keep it as small as possible.

3.2.2. Communication Infrastructure. For technology-related concerns, a regular arrangement of processing elements (PEs) with only neighboring connections is favored. This helps in (a) preventing using any long lines and their associated undesirable physical effects in deep submicron CMOS

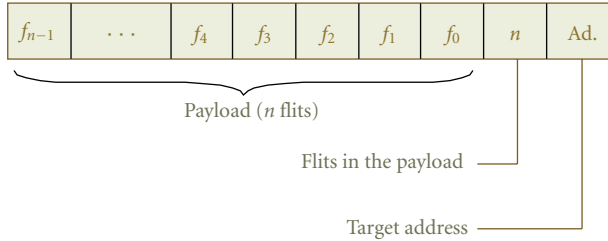


FIGURE 5: Packet format.

technologies and (b) synthesizing the clock distribution network since an asynchronous communication protocol between the PEs might be used. Also, from a communication point of view, the total aggregated bandwidth of the architecture should increase proportionally with the numbers of PEs it possesses, which is granted by the principle of abstracting the communications through routing data in space. The Network-on-Chip (NoC) paradigm enables that easily thanks to packet switching and adaptive routing.

The communication framework of HS-scale is derived from the Hermes Network-on-chip; refer to [21] for more details. The routing is of wormhole type, which means that a packet is made of an arbitrary number of flits which all follow the route taken by the first one which specifies the destination address. Figure 5 depicts the simple packet format used by the network framework constituted by the array of processing elements. Incoming flits are buffered in input buffers (one per port). Arbitration follows a round-robin policy giving alternatively priority to input ports. Once access to an output port is granted, the input buffer sends the buffered flits until the entire packet is transmitted (wormhole routing).

Inter-NPU communications are fully asynchronous and are based on the toggle-protocol. As depicted in Figure 4 this protocol uses two toggle signals for the synchronization, a given data being considered valid when a toggle is detected. When the data is latched, another toggle is sent back to the sender to notify the acceptance. This solution allows using completely unrelated clocks on each PE in the architecture.

3.3. Operating System. The lightweight operating system we use was designed for our specific needs. Despite being small (28 KB), this kernel does preemptive switching between tasks and also provides them with a set of communication primitives that are presented later. Figure 6 gives an overview of the operating system infrastructure and the services it provides.

Figure 7 presents the process state diagram each task follows depending on the events that may occur. This scheme answers to the general principles of operating systems in general although transition events have been specialized for this specific case.

The interrupts manager may receive interrupts from 3 hardware sources: UART, Timer, and network interface (NI). Whenever an interrupt occurs, other interrupts are disabled, and the processor context is saved in the system stack. Following the type of interrupt, it reads from the UART,

schedules another task (timer), receives data from other NPUs, or use a communication primitive (interrupt from the network interface FIFO_in). Afterwards processor context is restored, and interrupts are re-enabled. The scheduler is the core of the microkernel. Each time a timer interrupt occurs, it checks if there is a new task to run. In the positive case, it executes this new task. Otherwise, it has two possibilities: either there is no task to schedule then it just runs an idle task or there is at least one task to schedule. Tasks are scheduled periodically following a round robin policy (there is no priority management between tasks) as depicted in Figure 8.

Figure 9 presents the memory layout of an NPU running this operating system along with two tasks. Each task is located in a memory region that embeds code, data, and stack segments.

3.4. Dynamic Task Loading and Migration (PIC). One of the objectives of this work is to enable dynamic load balancing which implies the capability to migrate running tasks from processor to processor. Migrating tasks usually implies the following.

- (i) To dynamically load in memory and schedule a new process.
- (ii) To restore the context of the task that has been migrated.

3.4.1. Dynamic Process Loading. Both points are challenging for such microprocessor targets since, for density reasons, no Memory Management Unit (MMU) is available. An MMU, among other tasks, usually performs the translation between virtual and physical addresses and therefore permits to load and run a code in an arbitrary region of the physical memory. The code then performs read, write, and jump operations to virtual memory locations that are being translated into physical locations matching the memory layout decided by the operating system upon loading.

A possible alternative to overcome this problem relies in resolving all references of a given code at load-time; such a feature is partly supported in the ELF format [22] which lists the dynamic symbols of the code and enables the operating system loader to link the code for the newly decided memory location. Such mechanisms are memory consuming and imply a significant memory overhead which clearly puts this solution out of the scope of the approach.

Another solution for enabling the loading of processes without such mechanisms relies on a feature that is partly supported by the GCC compiler that enables to emit relocatable code (PIC: Position Independent Code). This feature generally used for shared libraries generates only relative jumps and accesses data locations and functions using a Global Offset Table (GOT) that is embedded into the generated ELF file. A specific postprocessing tool which operates on this format was used for reconstructing a completely relocatable executable. Experiments show that both memory and performance overheads remain under 5% for this solution which is clearly acceptable.

Figure 10(a) shows an example of relative jump with PIC compilation with code compiled for an execution at the

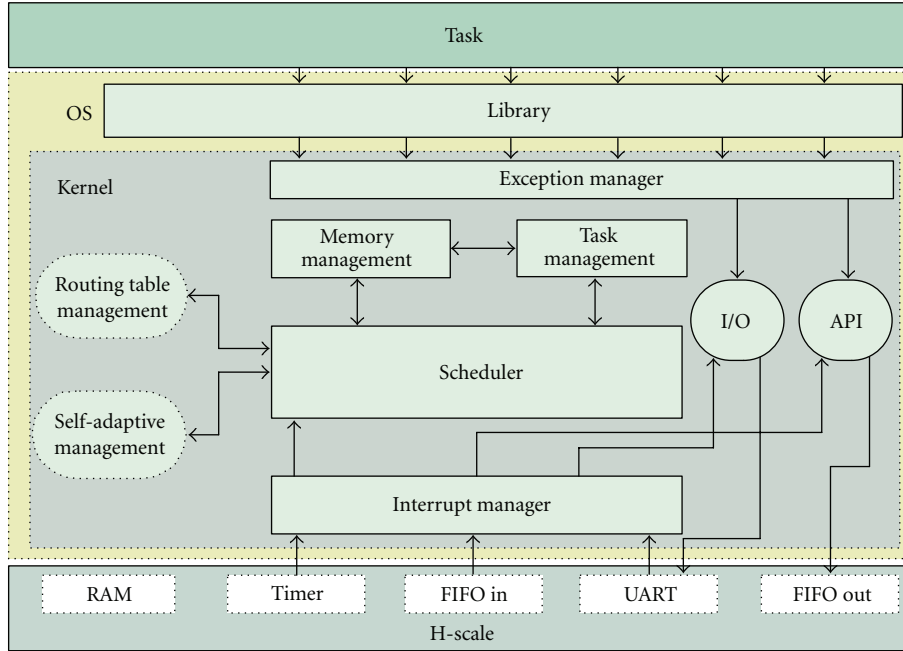


FIGURE 6: Operating system overview.

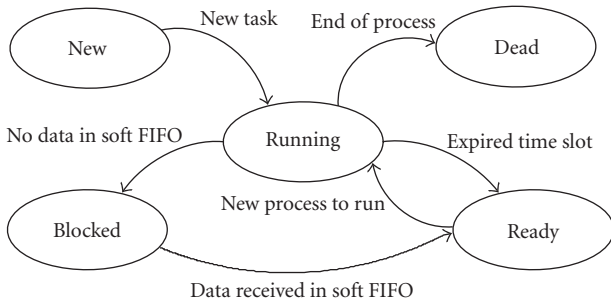


FIGURE 7: Task state diagram.

address $0x0000$; the address for the jump is referenced by the sum of current address and GOT entry reference. So if the code needs to be executed to another location, for example, 0×0100 (Figure 10(b)), the code is copied to the new location, and the only modification to perform is to add the same offset of the code (0×0100) to the GOT section entries.

3.4.2. Task Context Migration. Migrating a process implies not only instantiating a new executable into the memory but also restoring its context. Again, the lack of MMU makes this task difficult since the context of the process includes the stack which not only embeds data (such as return values of functions) but also returns addresses that are memory-location dependent. The solution we developed is based on defining migration points that are at specific locations in the code, namely, whenever a communication primitive is called. This method is restrictive since it assumes that the computation relies on a strict consumer/producer

model where no internal state is kept from iteration to iteration. This translates in the fact that there cannot be any dependencies between two adjacent computed data chunks.

When a task migration order is issued by the operating system, the following sequence of action is initiated between NPU1 which is current host for the task and NPU2 which is the future.

- (1) Task code is sent from NPU1 to NPU2. NPU2 then loads the task into memory, creates the necessary software FIFOs, and runs the task which is frozen when it reaches the first communication primitive call (`MPI.Send()` or `MPI.Receive()`).
- (2) NPU1 modifies routing tables (that embeds task and FIFO placements) and broadcasts this information to the other NPUs. Future messages for the task will be buffered in the newly created FIFO on NPU2.
- (3) Task execution on NPU1 continues until the next communication primitive call is reached which freezes application execution. Following this, the remaining task FIFO content on NPU1 is sent and reordered on NPU2.
- (4) Task execution is resumed on NPU2, and concurrently the task is removed from memory on NPU1.

Table 1 presents an example of migration time of one task (20 KB), between two NPUs with a distance of one. These results show that time migration is mainly due to the time to send executable through the network (with a frequency $f = 25$ MHz). $T_{Shutdown}$ refers to the time taken by the operating system for unregistering the task. T_{Send} represents the time taken for the operating system to perform the necessary actions for formatting and sending the task to

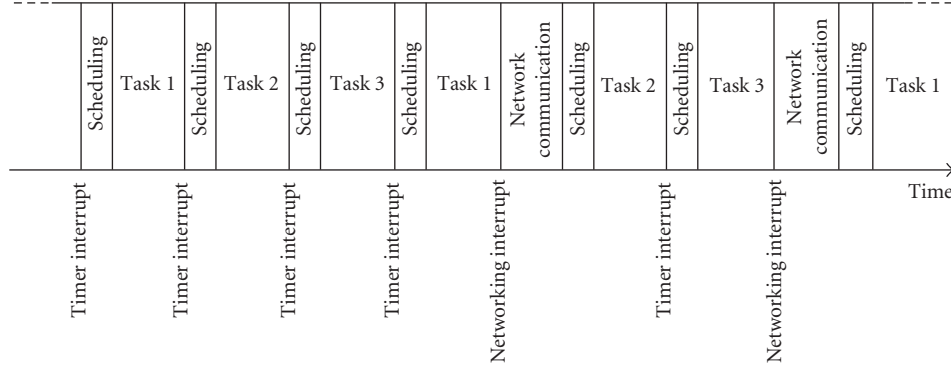


FIGURE 8: Scheduling diagram.

TABLE 1: Timeline of the migration mechanism.

	$T_{Shutdown}$	T_{Send}	$T_{Receive}$	$T_{Relight}$	T_{Reboot}	$T_{Migration}$
Time (ms)	3.067	13.970	13.968	3.110	0.107	34.222

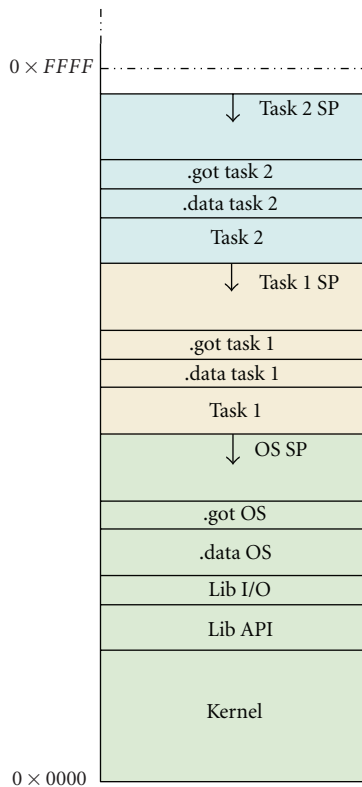


FIGURE 9: Memory layout.

the remote NPU, the time actually spent in sending the task through the network T_{Send} being negligible. Similarly, the operating system of the remote NPU requires a significant amount of time $T_{Receive}$ for receiving and instantiating the task in memory. Finally, the last action taken before running the task is updating both local and remote routing tables; this is realized in a time $T_{Relight}$. The task is then rapidly scheduled and executed (T_{Reboot}).

3.5. Programming Model. Programming takes place using a message passing interface. Hence, tasks are hosted on NPUs which provide through their operating system communication primitives that enable data exchanges between communicating tasks. The proposed model used only two communication primitives, $MPI_Send()$ and $MPI_Receive()$. This communication primitive is based on the synchronous MPI communication primitive (MPI_Send and $MPI_Receive$). Figure 11 depicts the layered view of the communication protocol we use. $MPI_Receive()$ blocks the task until the data is available while $MPI_Send()$ is blocking until the buffer is available on the sender side. In our implementation each call exhibits this behavior and is translated into a sequence of low-level $Send_Data()/Receive_Data()$ methods that set up a communication channel through a simple request/acknowledge protocol as depicted in Figure 12. This protocol ensures that the remote processor buffer has sufficient space before sending the message which helps lowering the contentions in the communication network and also prevents deadlocks.

Figure 12 depicts the communication stack that is used in our system. Although a hardware implementation could certainly help improving performance, for compactness reasons it is fully implemented in software down to packet assembling.

No explicit group synchronization primitives are provided; however this can be simply achieved in an ad hoc fashion through using $MPI_Send()$ and $MPI_Receive()$ for passing tokens. Broadcast, gather, and similar mechanisms can also be implemented in the same manner. Furthermore, although it could be easily implemented, no nonblocking receive method is provided since the targeted applications usually do not require it.

The prototypes of those functions are as in Algorithm 1.

The prototypes of these functions are self explanatory, a reference to the graph edge identifier, a constant void pointer, and data size expressed in bytes.

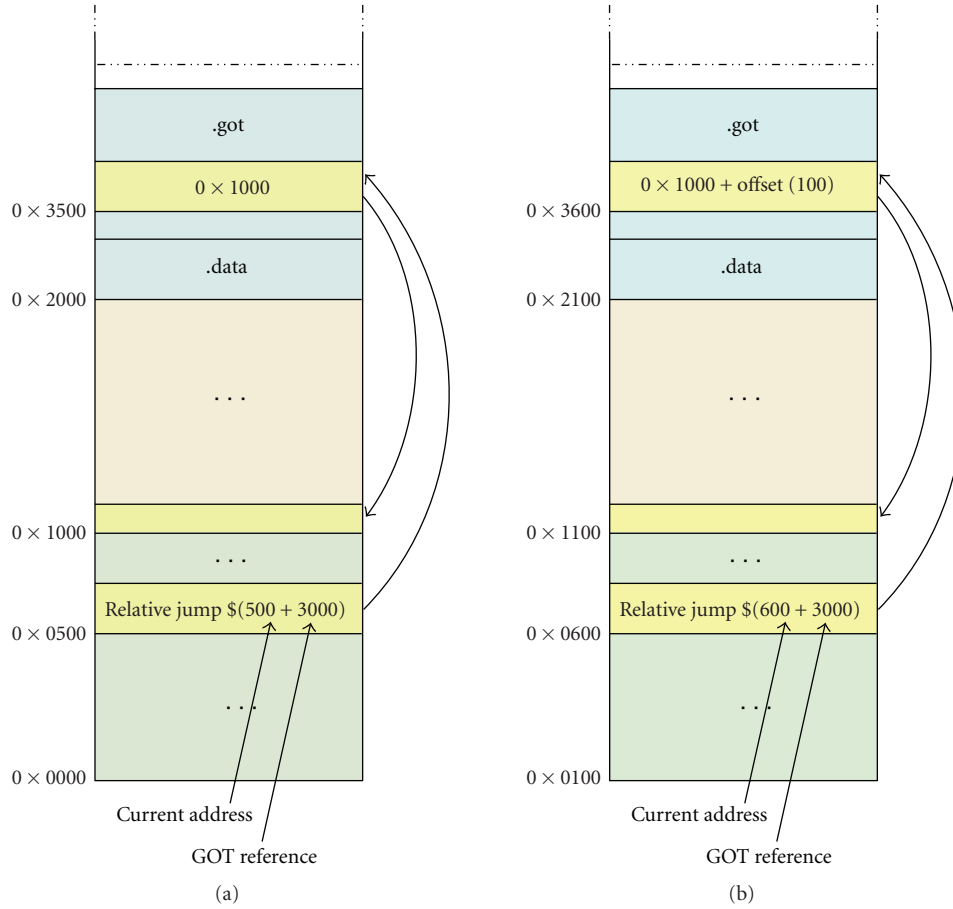


FIGURE 10: Relative jumps with GOT.

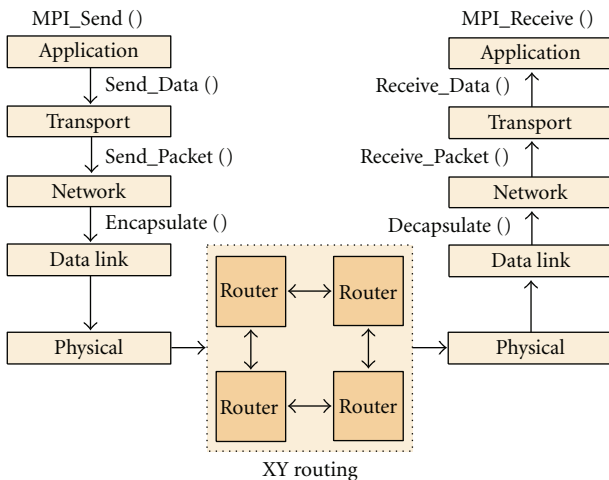


FIGURE 11: HS-Scale protocol stack.

Figure 13 shows an example of task graph where it can be seen that communication channels feature a (software) FIFO queue at the receiver side. Queues sizes can be parameterized, and their size can be tuned on-line as the operating system provides memory allocation and deallocation services.

```
MPI_Send(int edge, const void *data, int size)
MPI_Receive(int edge, void *data, int size)
```

ALGORITHM 1

3.6. *Self-Adaptive Mechanisms.* The platform is entitled to take decisions that relate to application implementation through task placement. These decisions are taken in a fully decentralized fashion as each NPU is endowed with equivalent decisional capabilities. Each NPU monitors a number of metrics that drive an application-specific mapping policy; based on these information an NPU may decide to push or attract tasks which results in, respectively, parallelizing or serializing the corresponding tasks execution, as several tasks running onto the same NPU are executed in a time-sliced manner.

Figure 14 shows an abstract example where it can be observed that upon application loading the entire task graph runs onto a single NPU; subsequent remapping decisions then tend to parallelize application implementation as the final step exhibits one task per NPU. Similarly, whenever a set of tasks become subcritical, the remapping could revert to situation 3 where T1, T2, and T3 are hosted on a single

NPU while the other supposedly more demanding do not share NPU processing resources with other tasks. These mechanisms help achieving continuous load-balancing in the architecture but can depending on the chosen mapping policy help refining placement for lowering contentions, latency, or power consumption.

Mapping decisions are specified on an application-specific basis in a dedicated operating system service. Although the policy may be focused on a single metric, composite policies are possible. Three metrics are available to the remapping policy for taking mapping decisions.

- (i) *NPU load*. The NPU operating system has the capability of evaluating the processing workload resulting from task execution.
- (ii) *FIFO queues filling level*. As depicted in Figure 13, every task has software input FIFO queues. Similarly to NPU load, the operating system can monitor the filling of each FIFO.
- (iii) *Task distance*. The distance that separates tasks is also a factor that impacts performance, contentions in the network, and power consumption. Each NPU microkernel knows the placement of other tasks of the platform and can calculate the Manhattan distance with the other tasks it communicates with.

Algorithm 2 shows an implementation of the microkernel service responsible of triggering task migrations. The presented policy simply triggers task migration in case one of the FIFO queues of a task is used over 80%.

The `request_task_migration()` call then sequentially emits requests to NPUs in proximity order; the migration function will migrate the task to the first NPU which has accepted the request; the migration process is started according to the protocol described previously in Section 3.4.2. This function can naturally be tuned on an application/task specific basis and select the target NPU taking into account not only the distance but also other parameters such as available memory and current load.

We have implemented also a migration policy based on the CPU load. The idea is very similar to the first one, and it consists of triggering a migration of a given task when the CPU load is lower or greater than a given threshold. This approach may be subdivided in two subsets.

- (1) Whenever the tasks time is greater than or equal `MAX_THRESHOLD`, it means that tasks are consuming more than or equal to the maximum acceptable usage of the CPU time.
- (2) Whenever the tasks time is less than `MIN_THRESHOLD`, it means that the tasks are consuming less than the minimum acceptable usage of the CPU time.

For both subsets, the number of tasks inside one NPU must be verified. For the first subset, it is necessary to have, at least, two tasks running in the same NPU. For the second subset, whenever there are one or more tasks in the same NPU, the migration process may occur.

In the same way the migration process occurs whenever the CPU load is less than `MIN_THRESHOLD` (20%). When

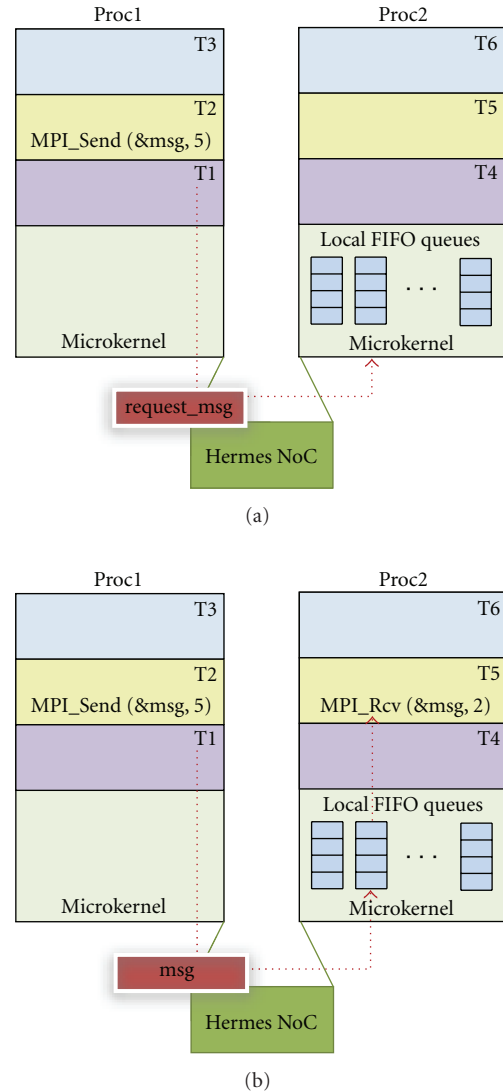


FIGURE 12: Proactive communication principle.

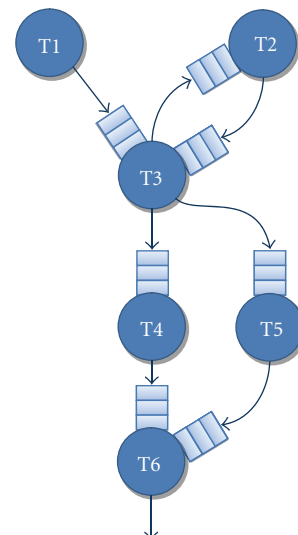


FIGURE 13: Example of task graph.

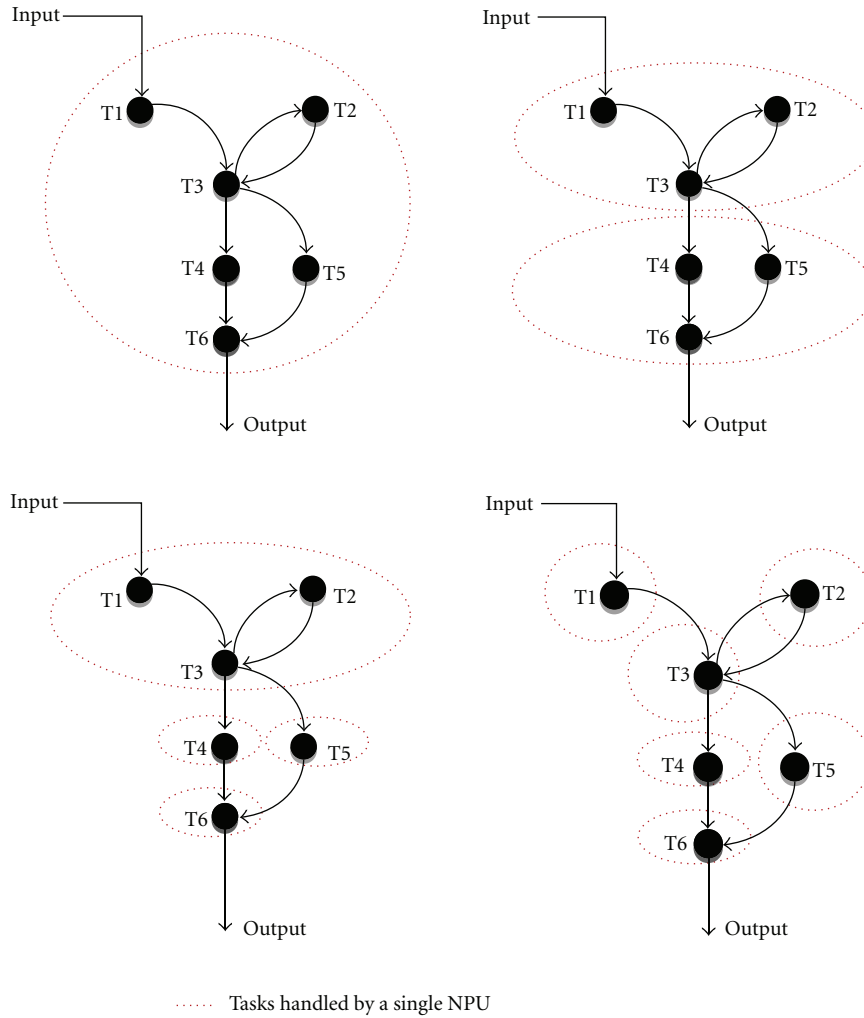


FIGURE 14: Task graph.

```

void improvement_service_routine(){
    int i, j;
    //Cycles through all NPU tasks
    for (i = 0; i < MAX_TASK; i++){
        //Deactivates policy for dead/newly instantiated tasks
        if (tcb[i].status != NEW && tcb[i].status != DEAD){
            //Cycles through all FIFOs
            for (j = 0; j < tcb[i].nb_socket; j++){
                //Verifies if FIFO usage > MAX_THRESHOLD
                if (tcb[i].fifo_in[j].average > MAX_THRESHOLD){
                    //Triggers migration procedure if task
                    //is not already alone on the NPU
                    if (num_task > 1)
                        request_task_migration(tcb[i].task_ID);
                }
            }
        }
    }
}

```

ALGORITHM 2

TABLE 2: Area scalability results.

Number of NPU	1	2	4 (2 × 2)	9 (3 × 3)	16 (4 × 4)
Area (mm ²)	1.14	2.29	4.60	10.33	18.40

this occurs, the migration function must look for a NPU that is using at given threshold of CPU usage, in this case, 60% of usage. To avoid the task with less than MIN_THRESHOLD keep migrating every time, we have inserted a delay to reduce the number of migrations.

4. Validations

4.1. Estimations of the Silicon Hardware Prototype. A complete synthesizable RTL level description (about 6000 lines of VHDL) of the H-Scale system has been designed. It has allowed us to validate our approach, to estimate areas (post place and route, with ST Microelectronics 90 nm design kit), and to improve the design. Any instance of the H-Scale MP-SOC system may be easily generated with generic parameters and then evaluated with any standard CAD tool flow (Encounter Cadence was used).

Table 2 summarizes these evaluations. The clock of the NPU has been constrained to 3 nanoseconds allowing a 300 MHz frequency. Table 2 clearly shows the area scalability of H-Scale hardware system (the very low overhead is due to the wires needed to interconnect the NPUs). These results let us easily extrapolate that we could design an HS-Scale system with 32 processors and 2 MB of embedded memory with less than 50 mm² of silicon area.

4.2. Multi-FPGA Prototype. The first validations of the systems were performed thanks to VHDL simulation. Obviously, this was far too slow for realistic application scenarios (about 4 minutes for a 10 milliseconds simulation with a 1.6 GHz processor). Although a SystemC prototype is also available, we chose to develop a scalable multi-FPGA prototype.

4.2.1. Platform Description. It is essentially based on a Spartan3 S1000 FPGA which 1920 configurable logic blocks (CLB). The board features several general purpose I/Os (8 slide switches, 4 pushbuttons, 8 LEDs, and 4-digit seven-segment display), 1 MB of fast asynchronous SRAM, several ports for debugging/monitoring purposes (one serial port, a VGA port, and PS2 mouse/keyboard port), and three 40-pin expansion connectors for the interconnections of boards.

As mentioned, one NPU is synthesized on a single FPGA board. The maximum frequency of the synthesized design on Spartan3 S1000 FPGA is 25 MHz. Figure 15 depicts the board with two of the 40-pin expansion connectors used for North, South, East, and West connections. Communications are taking place in an asynchronous fashion as described previously (toggle protocol).

Table 3 gives the device utilization figures for a single NPU hosted on a single board. The complete prototype is then composed of several instances of the prototyping boards connected through the 40-pin expansion connectors.

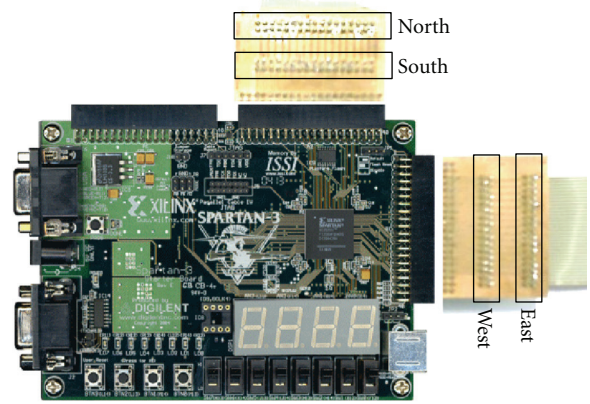


FIGURE 15: The prototyping board.

One board, that is, one UART of a single NPU, is directly connected to a PC as depicted in Figure 16. This PC is used as a human-machine interface for sending program data (i.e., task codes and microkernel code), the data to compute and to display debugging messages in the monitoring terminal.

Each NPU has originally a bootloader which performs upon power up the following operations in sequence.

- (1) It checks whether a PC is connected to the UART port. If so, the NPU initializes its XY coordinates to address (0 : 0). It then acts as a Dynamic Host Configuration Protocol (DHCP) server and proactively sends packets to the East and South ports informing that it has taken address (0 : 0).
- (2) If no UART connection is detected, an incoming network request is expected. Once the corresponding packet is received (that the interface NPU has initiated as described above), an address is calculated: East neighbor will take address (1 : 0) and South neighbor address (0 : 1).
- (3) This process is reiterated until the boundaries of the network are found. The X-axis and Y-axis boundaries are then broadcasted in the network in order to inform each NPU of the current network topology.
- (4) After the topology information update, the interface NPU bootloader downloads the microkernel code from the PC through the UART interface and broadcasts it to the other NPUs in the network. Each NPU starts up the operating system as soon as received. The microkernel is common for each NPU. Depending on the address of the router, the microkernel knows its location. For application code, again NPU0 receives the code since it is the only one connected to the UART. This code is forwarded only to the processor, where it is supposed to execute. Only a single copy of application code exists inside the system.

This method allows to easily scale up the prototype to any size and shape (form factor may be different than 1). It has

TABLE 3: FPGA synthesis result.

Device	#Slices	#FPGA resources used
NPU	2496	32,50%
Router	683	8,89%
MIPS R3000	1462	19,04%
Other	351	4,57%
Total used	4992	65%

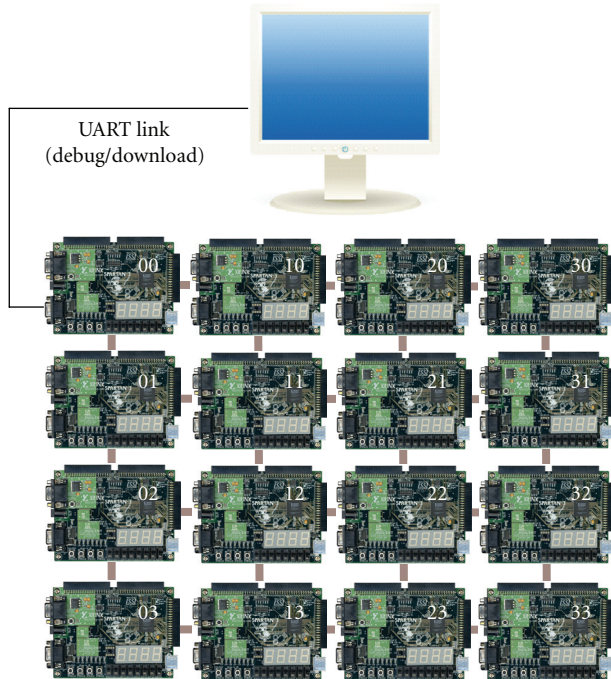


FIGURE 16: Array of 4 × 4 NPU multiboard.

TABLE 4: Operating system time and memory costs (KB) size.

Microkernel	10
Com. primitive	5,12
I/O primitive	3,12
DATA section	8,48
STACK section	0,56
OS Size	27,28

been designed for later exploring reliability issues for reconfiguring the system if an NPU becomes unreachable (defective hardware) for instance. In such a case, the faulty NPU can be removed from the table of available processing units.

4.2.2. Kernel Characteristics. Table 4 provides an overview of the memory footprint of our Operating System. In terms of time penalty, each time the OS is invoked (each time a timer interrupt happens), it requires 218 cycles to perform its job. In terms of memory overhead, it requires 27.28 KB. The communication primitives represent almost one fifth of the total memory required by our OS.

4.3. Description of the Experiments. The FPGA platform is the basis of our experiments. Those have been carried out on the HS-Scale system in order to study and characterize the strengths and the weaknesses of our approach.

4.3.1. Set of Applications. We have chosen 3 different applications: a 2-TAP Finite Impulse Response (FIR) filter, a Data Encryption Standard (DES) encoder, and an MJPEG decoder. The main motivation for using such applications was to cover a wide range of possible dataflow applications in terms of granularities and regularities of the tasks. The FIR filter is based on fine grain tasks with a task graph requiring multiple dependencies. Compared to FIR filter, the granularity of DES tasks and MJPEG tasks is coarser. DES tasks are regular and not data dependent, while MJPEG tasks are irregular and depend on the image characteristics. Some dummy applications have been created for better highlighting the capabilities of different policies.

4.3.2. Experimental Protocol. We have developed a set of self-adaptive features for the HS-Scale system: the purpose of this study is to evaluate and to measure the impact of the self-adaptability on application performance. The main metric presented in the next section is the Throughput (TP). It is computed as follows:

$$TP(\text{KB} \times \text{s}^{-1}) = \frac{\text{Number of Computed Data (KB)}}{\text{Number of Cycles}} \times f(\text{Hz}). \quad (1)$$

Our experiments were performed with each NPU running at $f = 50$ MHz. We have implemented different application scenarios as follows.

(i) *Monoprocessor Implementations.* Each application of our test set is programmed as a monolithic task (with or without the operating system) in order to calculate a reference throughput.

(ii) *Multiprocessor Implementations with Static Mappings.* Each application is described as a task graph application. Performing figures for various static mappings have been collected.

There is a certain degree of randomness in the execution of a scenario due to different reasons. Firstly, the communications between routers are asynchronous. Secondly, the execution of a task on a given NPU depends on several varying parameters such as the presence of data in its FIFO (may depend on other tasks placed on different NPUs communicating through the asynchronous network) and the timer interrupts regarding application start that can induce a different scheduling and a different timing in the decision making process. This is the reason why, for (1) and (2), the experiments were repeated 10 times in order to expose the average throughput and its standard deviation.

(i) *Multiprocessor Implementations with Dynamic Mappings (Migrations).* This case represents our main contribution with self-adaptability features. The studied scenario relies

TABLE 5: FIR, DES, and MJPEG monolithic implementations and OS cost.

		-OS ^(a)	+OS ^(b)
FIR	Average TP (KB/s)	315.92	313.08
	Standard deviation of TP (Kb/s)	0	0.09
DES	Average TP (KB/s)	6.56	6.54
	Standard deviation of TP (Kb/s)	0	0.06
MJPEG	Average TP (KB/s)	35.39	34.98
	Standard deviation of TP (Kb/s)	0	0.14

^(a)Without the operating system.

^(b)Without the operating system.

on an application that is sequentially injected on a single NPU which triggers remapping decisions. These remapping decisions are all based on nearest-free neighbor policy where every time the FIFO utilization reached the 80% threshold, a migration was triggered. We have monitored dynamically the throughput and the FIFO utilization ratio in order to plot these metrics as temporal functions.

5. Applications and Results

This section is devoted to the analysis of the self-adaptive results obtained on the FPGA prototype. Three classes of results are exposed: (i) monoprocessor implementations used as reference, (ii) multiprocessor static mappings, and (iii) self-adaptive implementation where tasks freely migrate from NPU to NPU.

5.1. Monoprocessor Study. Each application (FIR, DES, and MJPEG) was programmed as a monolithic task with or without the Operating System. The aim of this study is to evaluate the impact of the use of our OS on performance and also to provide a reference performance for further implementations.

Table 5 summarizes the results. The important information from these results are (1) that the impact of the Operating System on the throughput performance is relatively low (less than 0.95% for each application of our test set) and (2) that it introduces a certain level of randomness (shown by the standard deviation). As a conclusion, the OS provides low-cost multitasking capabilities and implies a very little reduced quality-of-service as the throughput is not deterministic anymore.

5.2. Static Placement Study. Each application of our test set was partitioned into several tasks. Our objective was to distribute the computations of a given application onto several processors in order to evaluate the impact on the throughput. This distribution was hand-made (static placement), and no migration was allowed for discarding the influence of transient phenomena. All references to task placements made throughout this section rely on the addressing mode presented in Figure 16.

5.2.1. Data Encryption Standard. The Data Encryption Standard (DES) algorithm is composed of several computational

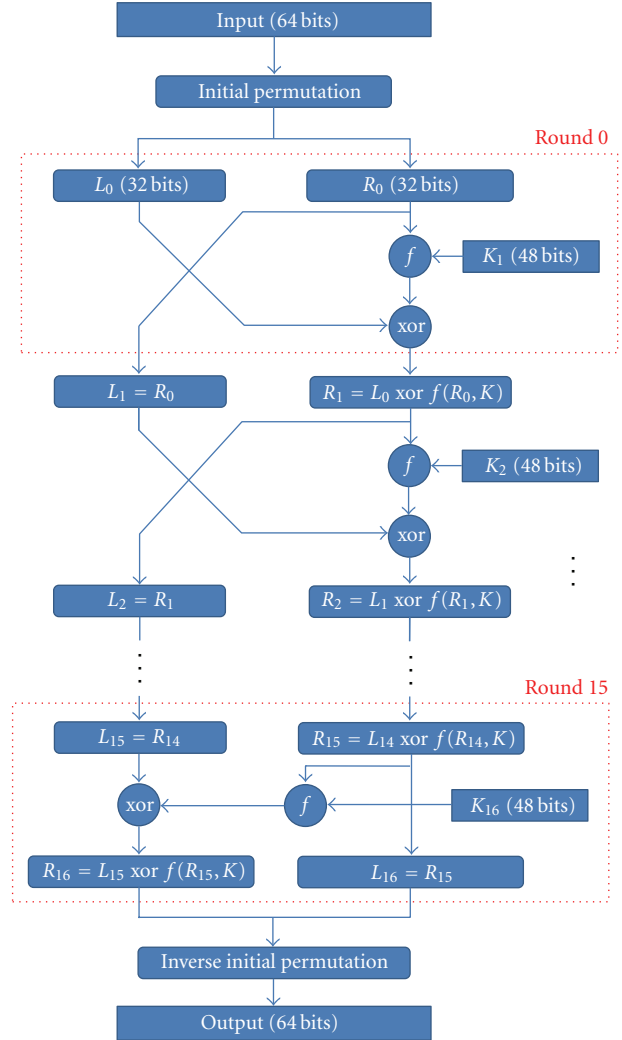


FIGURE 17: The DES algorithm.

steps as depicted in Figure 17. All 16 rounds are functionally equivalent but operate with different keys ($K_1 \dots K_{16}$).

We have chosen to implement it in a pipeline fashion, by decomposing the rounds (16 rounds). Figure 18 shows the DES performance results for different partitioning of the DES algorithm. We devised 4 different pipelines, with 2, 4, 6, and 8 tasks which therefore correspond to tasks embedding 8 to 2 rounds. Due the communication overhead introduced by the task partitioning, we observed that the performance is decreased when running all tasks on the same NPU. Then, when expanding the task graph to other NPUs, we observe that the throughput increases rapidly until it reaches its maximum value when n tasks are mapped to n NPUs. The OS overhead is generally hidden when the n tasks can be mapped to $n/2$ NPU. Finally, the performance improvement of n task partitioning corresponds to n stage pipeline, that is, the reference throughput is at the maximum approximately multiplied by n .

5.2.2. MJPEG Decoder. Figure 19 shows the processing pipeline of a JPEG encoder operating on grey-coded images.

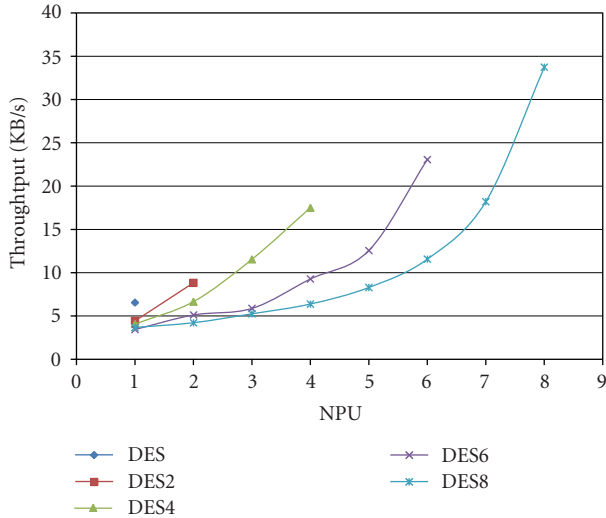


FIGURE 18: DES performances results with 1, 2, 4, 6, and 8 tasks.

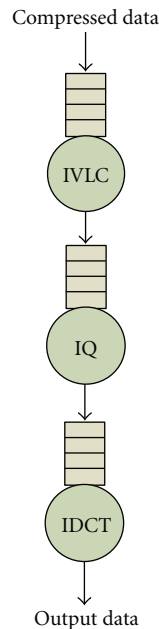


FIGURE 19: MJPEG data flow.

The first step of this application is the inverse variable length coding (IVLC) which relies on a Huffman decoder. This processing time for that task is data dependent. The two last tasks of the processing pipeline are, respectively, the inverse quantization (IQ) and the inverse discrete cosine transform (IDCT). The atomic data transmitted from task to task is an 8×8 pixel block which has a size of 256 bit. We naturally chose to use a traditional task partitioning as depicted in Figure 19 with both a task-level dataflow description.

Similarly to the FIR and DES applications, the operating system communication primitives induce a performance overhead when the decoder is splitted into 3 tasks (Table 6, column 1) compared to the performance shown in Table 6, column 2 (39.11 KB/second). Distributing the processing of

TABLE 6: Throughput evolution with task graph expansion.

#NPU	1	2	3
Task placement	3L ^(a)	2L, 1R	3R
Average TP (KB/s)	29.21	39.11	39.05
Standard deviation of TP (Kb/s)	0.21	0.52	0.40

^(a)The notations L and R refer to “Local” and “Remote” executions.

IVLC on a remote NPU (Table 6, column 3) immediately pays with a significant increase in the throughput. The fully distributed implementation exhibits a very small performance improvement when comparing to a local implementation, which is due, as we will show in the next section, to the fact that a critical task in the processing pipeline already fully employs the processing resources of a given NPU. The standard deviation, as previously observed for DES and FIR applications, increases with task partitioning and distribution.

5.3. *Dynamic Placement Study (with Migrations)*. The aim of this section is to analyze the dynamic behavior of HS-Scale. We will study on the different remapping policies of the transient phenomenon in time, and the impact on performance will be measured.

5.3.1. *Diagnostic and Decision Based on Communication Load*. The first migration policy corresponds to a percentage of the FIFO utilization threshold (80% in our experiments). In this case, when the software monitor detects that the FIFO is filled over 80% for a given task, this task is automatically moved to another NPU according to a first neighbor policy.

Two scenarios DES4 and MJPEG applications are exposed in this section to prove the validity of this policy, starting from the neighbor on the east.

5.3.2. *DES4*. In this scenario, we used the DES application partitioned into 4 tasks: task 1 (T1) corresponds to the rounds 1 to 4, task 2 (T2) corresponds to the rounds 5 to 8, task 3 (T3) corresponds to the rounds 9 to 12, and task 4 (T4) to rounds 13 to 16. Figure 20 depicts the measured throughput of the application with the normal policy.

During the first seconds, the tasks (T1, T2, T3, T4) are manually sent from the external PC and placed on the NPU(1,1). At $t_1 = 25.43$ seconds, the DES starts the computation: the four tasks are running sequentially on the same NPU. Three migrations are performed by the policy in less than 200 milliseconds; the FIFO levels of T1, T3, and T4 are above the threshold at times $t = 26.21$ seconds, $t = 27.02$ seconds, and $t = 27.78$ seconds, respectively. After the last migration, we can see an increase of the throughput that also causes a decrease T3 FIFO filling. The throughput then stabilizes around 17 KB/s.

Comparing the performance of the last placement with the static mapping presented previously, we observe identical figures; this policy has rapidly found (1.59 seconds) one of the best placement. In order to better observe the evolution of performance in the different steps, results presented in

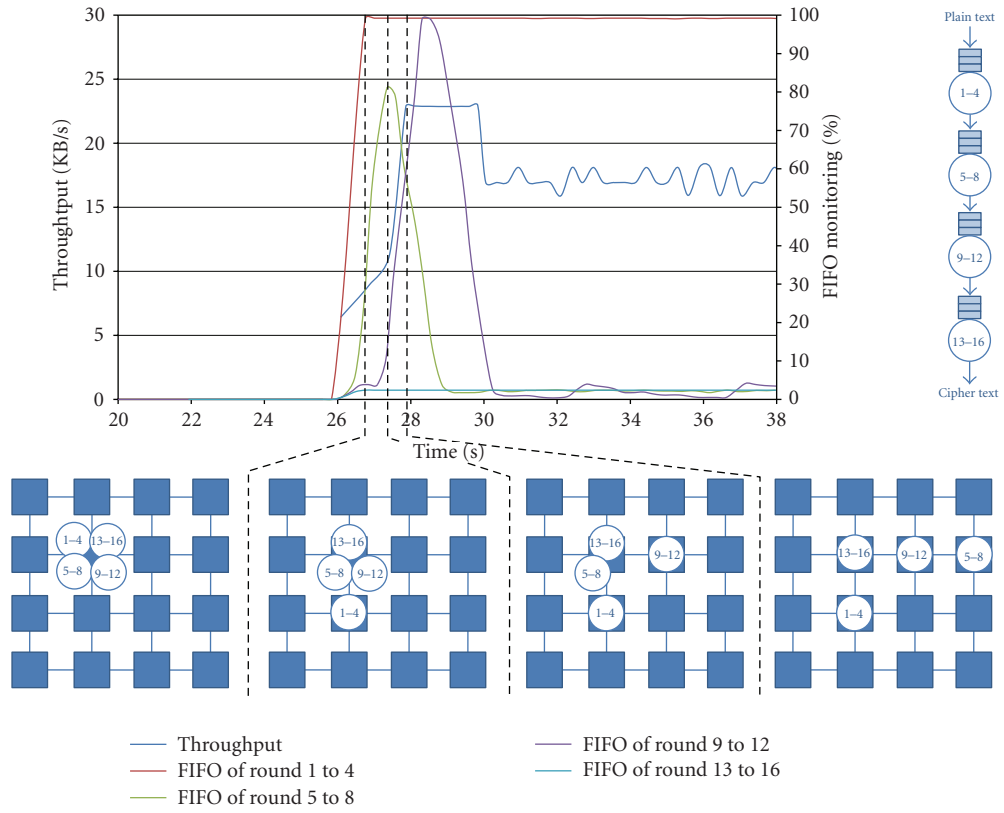


FIGURE 20: DES4 execution in time without delay.

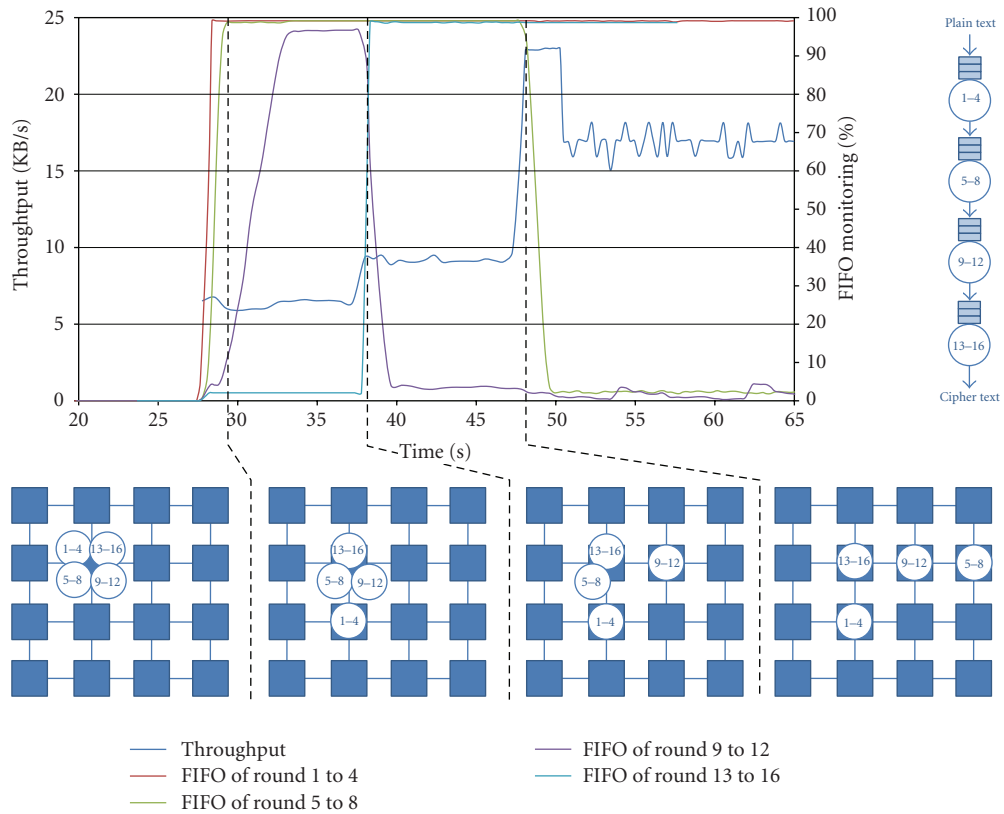


FIGURE 21: DES4 execution in time with delay.

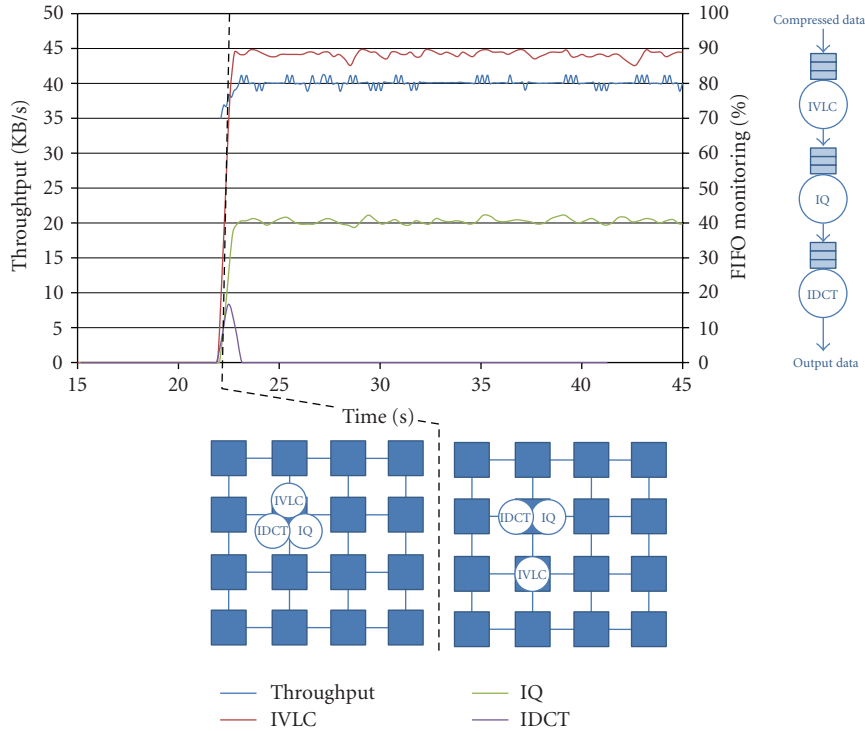


FIGURE 22: MJPEG execution in time.

Figure 21 were implemented using a policy that allows one migration every 10 seconds at most.

At the beginning after the start of the DES application at $t_1 = 28.71$ seconds, all tasks are running in the same NPU, and the throughput of the application is around 4 KB/s. At $t_2 = 29.02$ seconds, the FIFO monitor of T1 indicates a FIFO usage greater than 80%: it then begins automatically to move the task (i.e., (1) waiting a migration point, (2) looking for a free NPU, (3) migrating the task, and (4) restoring the context). Due to the migration policy, the task is placed on the NPU(1,2). Then, the throughput stabilizes around 6.5 KB/s. After this migration, FIFO fillings of T2 and T3 increase quickly above 80% because T1 shows a higher throughput due to the fact it benefits from an entire NPU. As soon as the policy re-enables migrations ($t_3 = 39.03$ seconds), T3 migrates on NPU(2,1). FIFO usage of T4 then increases because of the time-multiplexing execution which results ten seconds later into another migration ($t_4 = 49.04$ seconds).

The same final state is observed as previously with an average throughput of 17 KB/s. This clearly demonstrates for the DES application that such a simple policy is capable of rapidly converging to a best placement.

5.3.3. MJPEG. In this scenario, we use the MJPEG decoder application partitioned into three tasks: IVLC, IQ, and IDCT. Figure 22 depicts the application throughput and the FIFO usage of each task in the pipeline.

During the very first seconds, all tasks are instantiated manually on the NPU(1,1). From $t_1 = 21.35$ seconds to $t_2 = 22.26$ seconds, these tasks are executed sequentially on the same NPU which provides an average throughput of 30 KB/s.

At t_2 , the IVLC FIFO reaches a value greater than 80%: this leads to a migration process that involves the following steps.

- (i) Freezing task execution and search for a free NPU (3.11 milliseconds).
- (ii) Migrating the task (14.05 milliseconds).
- (iii) Restoring the context (3.32 milliseconds).

Due to the migration policy, the task is moved from NPU(1,1) to NPU(1,2): it takes 20.48 milliseconds for the whole task migration process. During this time, the OS consumes CPU time for the migration process which decreases the application throughput. After the migration completion, the average throughput reaches 40 KB/s: it takes 153.6 milliseconds until a performance benefit is observed.

We then observe the following behavior of the system: on one hand, the IQ FIFO utilization remains stable meaning that it has just enough CPU time to process its data, and on the other hand the IDCT FIFO decreases meaning that it has enough CPU time to process all the data in its FIFO. In this situation, the mapping is stable from the migration policy point of view.

This clearly suggests that the policy is adequate with respect to the optimization of performance: as seen previously in Table 6 since IVLC proves the most time consuming task, therefore any further migration would not help improving performance.

5.3.4. Diagnostic and Decision Based on CPU Workload. The example on Figure 23 shows the results for this migration policy based on the CPU workload. The experimental

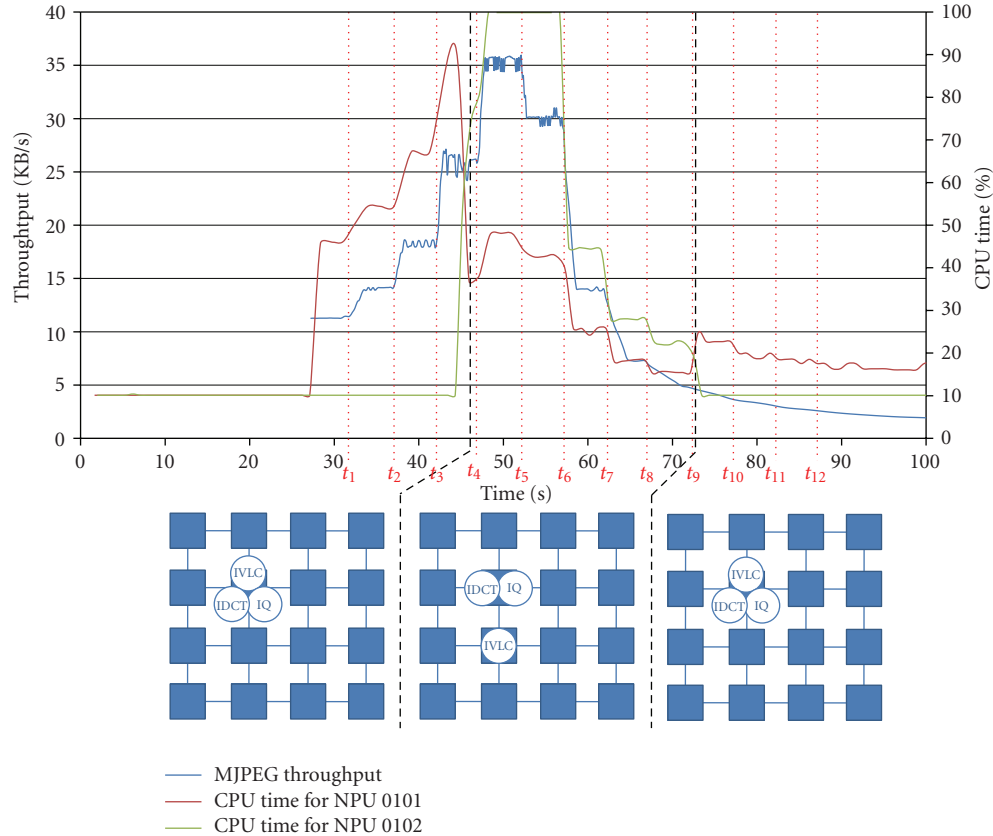


FIGURE 23: MJPEG decoder throughput based on CPU workload.

protocol used for these results relies on varying the input data rate for observing how the system adapts.

At the beginning all tasks (IVLC, IQ, and IDCT) are running on the same NPU(1,1), but the input throughput on the MJPEG application is lower so the CPU time consumed is around 47%. At each step t_1 , t_2 , and t_3 the input throughput is increased, so we can see an increase of the CPU time consumed step by step. When the CPU time used exceed the threshold (i.e., 80%), the operating system detects that the NPU(1,1) is overloaded (at 45 seconds) so it decided to migrate the task which uses the most of CPU time on a neighboring NPU. In this example IVLC task migrates on NPU(1,2) which decreases the CPU time used by NPU(1,1) around 35% and an increase of CPU used by NPU(1,2) around 80%. At t_4 the input throughput increases more which leads to an MJPEG throughput increase around 35 KB/s and overloaded the NPU(1,2) at 100% but migration is not triggered because just one task is compute.

From t_5 to t_{12} the input throughput of the MJPEG application is decreased step by step, and when the CPU time of NPU(1,2) is less than 20% (at 72 seconds), the operating system decides to closer task on the same NPU (the NPU(1,1)). We can see after this migration a decrease of CPU time used by the NPU(1,2) and an increase of CPU time used by NPU(1,1) but without saturate it.

We can observe that MJPEG application performances are lower than in the static mode; this is because the

operating system uses more CPU time (around 10%) to monitor CPU time and average sample.

5.3.5. Diagnostic and Decision Based on Locality. The third migration policy corresponds to optimizing placement for decreasing Manhattan distance [23] (number of hops) between communication tasks: whenever the software monitor detects a closer placement of a given task, this task is automatically migrated to this NPU.

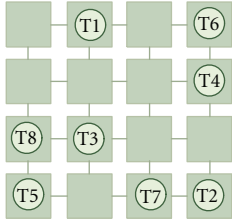
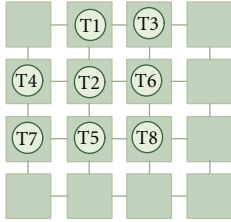

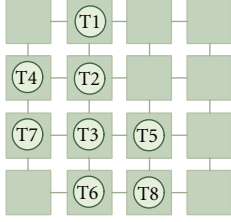

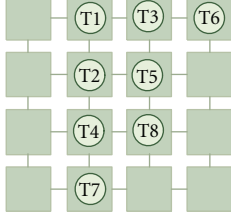

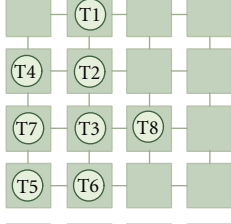

To prove the validity of this policy, we will expose results on one scenario with a dummy application that is made of numerous communicating tasks.

In this scenario, we use a dummy application which generates intensive traffic between communicating tasks. Each task consumes a different value of CPU time to compute this data. The graph task of this dummy application is presented on Figure 24.

Table 7 presents the results obtained with the policy based on the closest placement. For four initial placements, we have executed five simulations and observed the final placements obtained. This table summarizes these experiments and gives the initial placement, the final placements as well as the cumulated distance for these.

The best obtained placements exhibit 13 hops which is a satisfying result with respect to the best placement presented previously. These placements were furthermore obtained with a limited number of migrations (around 10) which suggests that deriving policies which would take into account

TABLE 7: Dummy application with closest placement policy.

Initial placement		Number of migrations	Final placement							
Number of hops between tasks	Graph		Number of hops between tasks	Graph						
31		10	13							
				7	14					
						7	13			
								6	18	
										10

more parameters or even take sub-optimal decisions for avoiding local minima could help further improving these results.

6. Conclusions

Microelectronics currently undergo profound changes due to several factors such as the approaching limits of silicon CMOS technology as well as the inadequacy of the machine models that have been used until now. These challenges imply to devise new approaches to the design and programming of future integrated circuits. Hence, parallelism appears

as the only solution for coping with the ever increasing demand in term of performance. Together with this, issues such as reliability and power consumption are yet to be addressed. The solutions that are suggested in literature often rely on the capability of the system to take online decisions for coping with these issues, such as scaling supply voltage and frequency for increasing energy efficiency, or testing the circuit for identifying faulty components and discarding them from the functionality.

MPSoCs are certainly the natural target for bringing these techniques into practice: provided they comply with

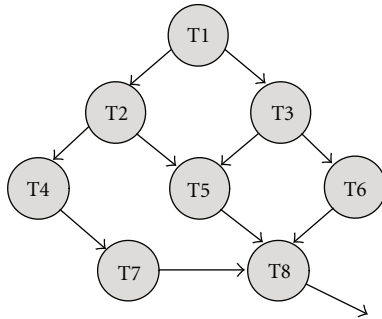


FIGURE 24: Dummy task graph.

some design rules they may prove scalable from a performance point of view, and since they are in essence distributed architectures, they are well suited to locally monitoring and controlling system parameters.

From the software point of view, the system presented in this paper relies on a tiny operating system that is used on every processing element. This decision certainly helps in improving system scalability as the adaptability policies that were proposed rely on a purely decentralized decisionmaking approach. The three presented policies suggest that although distributed and operating on either local or possibly non up-to-date system information, performance versus static scenarios are comparable. Finally we also demonstrated that such a solution could be viable even considering economic constraints such as silicon cost: the overhead incurred by this approach has been quantified in terms of logic and memory thanks to the realization of the 16 processors prototype which is fully functional.

This paper describes an adaptive MPSoC framework that utilizes a message passing programming model. Based on information implemented in the form of distributed software monitors, the user can specify migration policies that enable the architecture to refine task placement. The conducted experiments show the following.

- (i) Migration helps better balancing processor load at run-time for achieving better performance.
- (ii) Task migration incurs a minimal performance overhead.

The important characteristics that have been considered are mostly flexibility, scalability, and adaptability. The remapping policies adopted show that it is possible to have a very flexible, scalable, and self-adaptable MPSoC by using monitoring systems not complex which does not affect the overhead of the system.

The proposed architecture relies on the following design decisions.

- (i) Decentralized control.
- (ii) Homogeneous array of processing elements.
- (iii) Distributed memory.
- (iv) Scalable NoC-style communication network.

Although such principles have so far demonstrated the interest in the sole context of performance, it suggests that

other benefits could be achieved such as power management where highly communicating task could be mapped to neighbor processors whereas this constraint is relaxed for others. Future work aims at exploring these functionalities in connection with voltage and frequency scaling techniques since these approaches combined could reveal useful for better balancing performance and power consumption.

Another promising perspective of this work relies on task replication which helps further taking advantage of processing resources whenever needed. Although this is applicable for certain applications only, our first experiment on the MJPEG application shows that only critical task gets replicated with similar mapping/replication policies as the one presented in this paper. This should prove particularly useful for tasks which data dependant processing load such as the IVLC in our case.

Although this has not yet been demonstrated, the structure also intrinsically supports fault tolerance to a certain degree since each processor is aware of all others in the system. A faulty processor identified as such using mutual testing techniques could be discarded from the list of functional units, and further mapping decisions would therefore target the remaining processors only. Extending the control to some different system parameters such as processing elements frequency and supply voltage would certainly help further improving observed performance. Among the considered possible strategies, task replication is certainly of significant interest as it would help better matching the number of processing resources to the performance demand. This technique would prove applicable in some restricted domains (mostly for dataflow applications) but could probably be extended.

Similarly, for the growing concern of fault tolerance the developed techniques could here constitute a viable solution; since the system is entirely distributed, faulty units could be identified by others and therefore functionality would rely on the remaining processing units.

Finally, we think that adaptability is an approach that will in the near future be widely adopted in the area of MPSoC. Not only because of the here mentioned limitations such as technology shrinking, power consumption, and reliability but also because computing undoubtedly goes pervasive. Pervasive or ambient computing is a research area on its own and in essence implies using architectures that are capable of self-adapting to many time-changing execution scenarios. Examples of such applications range from ad-hoc networks of mobile terminals such as mobile phones to sensor networks systems aimed at monitoring geographical or seismic activity. In such application fields, power efficiency, interoperability, communication and scalability are primary concerns such systems have to cope with many limitations such as limited power budget, interoperability, communication issues, and finally, scalability.

References

- [1] D. E. Culler, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann, San Francisco, Calif, USA, 1st edition, 1998.

- [2] W. M. Collier, *Reasoning about Parallel Architectures*, Prentice-Hall, Englewood Cliffs, NJ, USA, 1992.
- [3] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, vol. 21, no. 9, pp. 948–960, 1972.
- [4] B. Nichols, D. Buttler, and J. P. Farrell, *Pthreads Programming*, O'Reilly, Sebastopol, Calif, USA, 1996.
- [5] "The OpenMP API specification for parallel programming," <http://openmp.org/wp>.
- [6] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, Scientific and Engineering Computation Series, MIT Press, Cambridge, Mass, USA.
- [7] G. Kahn, "The semantics of a simple language for parallel programming," in *IPIP Congress*, pp. 471–475, 1974.
- [8] M. Saldana and P. Chow, "TDM-MPI: an MPI implementation for multiple processors across multiple FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '06)*, pp. 1–6, Madrid, Spain, August 2006.
- [9] J. Kohout and A. D. George, *A High-Performance Communication Service for Parallel Computing on Distributed DSP Systems*, Elsevier Science, Amsterdam, The Netherlands, 2003.
- [10] J. J. Murillo, D. Castells-Rufas, and J. C. Bordoll, "HW-SW framework for distributed parallel computing on programmable chips," in *Proceedings of the Conference on Design of Circuits and Integrated Systems (DCIS '06)*, 2006.
- [11] "MPCore Linux 2.6 SMP kernel and tools," ARM Limited, http://www.arm.com/products/os/linux_download.html.
- [12] A. Barak, O. La'adan, and A. Shiloh, "Scalable cluster computing with MOSIX for Linux," in *Proceedings of the Linux Expo*, pp. 95–100, Raleigh, NC, USA, May 1999.
- [13] J. Robinson, S. Russ, B. Heckel, and B. Flachs, "A task migration implementation of the message-passing interface," in *Proceedings of the IEEE International Symposium on High Performance Distributed Computing (HPDC '96)*, p. 61, 1996.
- [14] A. R. Dantas and E. J. Zaluska, "Improving load balancing in an MPI environment with resource management," in *High-Performance Computing and Networking*, pp. 959–960, Springer, Berlin, Germany, 1996.
- [15] L. Chen, C.-L. Wang, F. C. M. Lau, and K. K. Ricky, "A grid middleware for distributed Java computing with MPI binding and process migration supports," *Journal of Computer Science and Technology*, vol. 18, no. 4, pp. 505–514, 2003.
- [16] S. Carta, M. Pittau, A. Acquaviva, et al., "Multi-processor operating system emulation framework with thermal feedback for systems-on-chip," in *Proceedings of the 17th Great Lakes Symposium on VLSI (GLSVLSI '07)*, pp. 311–316, Stresa-Lago Maggiore, Italy, March 2007.
- [17] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali, "Supporting task migration in multi-processor systems-on-chip: a feasibility study," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '06)*, vol. 1, pp. 1–6, Munich, Germany, March 2006.
- [18] L. Benini, D. Bertozzi, A. Bogliolo, F. Menichelli, and M. Olivieri, "MPARM: exploring the multi-processor SoC design space with systemC," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 41, no. 2, pp. 169–182, 2005.
- [19] M. Pittau, A. Alimonda, S. Carta, and A. Acquaviva, "Impact of task migration on streaming multimedia for embedded multiprocessors: a quantitative evaluation," in *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia '07)*, pp. 59–64, October 2007.
- [20] MIPS corp., <http://www.mips.com>.
- [21] F. Moraes, et al., "Hermes: an infrastructure for low area overhead packet-switching networks on chip integration," *VLSI Journal*, vol. 38, pp. 69–93, 2004.
- [22] J. R. Levine, *Linkers and Loaders*, Morgan Kaufmann, San Francisco, Calif, USA, 1999.
- [23] P. E. Black, "Manhattan distance," <http://www.itl.nist.gov/div897/sqg/dads>.



Hindawi

Submit your manuscripts at
<http://www.hindawi.com>

