CRESTA

SEVENTH FRAMEWORK PROGRAMME

D2.5.3 – Proactive Fault Tolerance

WP2: Underpinning and Cross-Cutting Technologies

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exa-scale computing, software and simulation

Due date:	PM 39
Submission date:	31/12/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organisation	KTH
Version:	1.0
Status	Final
Author(s):	Gilbert Netzer (KTH), Luis Cebamanos (UEDIN), Stefano Markidis (KTH)
Reviewer(s)	Jens Doleschal (TUD), David Lecomber (ASC)

Dissemination level	
<PU/PP/RE/CO>	*PU*

© CRESTA Consortium

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	25/06/2014	TOC of the deliverable	KTH
0.2	10/08/2014	Added a literature study and incomplete reference list	KTH
0.3	01/12/2014	Version for internal review	KTH, UEDIN
0.4	10/12/2014	Incorporated feedback from review	KTH
1.0	15/12/2014	Final fixes	UEDIN

Table of Contents

1 INTRODUCTION **6**

2 BACKGROUND **7**

2.1 Exascale System Architecture 7
2.2 Exascale Operating Systems 8
2.3 Virtualisation at Exascale 8

3 FAULT TOLERANCE STRATEGIES **10**

3.1 Checkpoint-Restart Fault Tolerance 11
3.2 Migration for Proactive Fault Tolerance 11

4 PROCESS MIGRATION FOR HETEROGENEOUS SYSTEMS **12**

4.1 System Call Forwarding and Application Process Migration 12
4.2 Hardware Set-up for System Call Forwarding Demonstration 13
4.3 System Call Forwarding Demonstration on DSP Hardware 14

5 VIRTUALISATION-BASED FAULT TOLERANCE ON HPC SYSTEMS **15**

5.1 Test environment 15
5.2 Virtualisation Overheads 16
5.3 Live and Offline Migration 18
5.4 Pro-active Migration 21
5.5 Limitations of the Approach 21

6 DISCUSSION AND CONCLUSIONS **22**

6.1 Impact on CRESTA Applications 22
6.2 Impact on CRESTA Software-stack 22
6.3 Impact on Hardware 22

REFERENCES **24**

GLOSSARY OF ACRONYMS **26**

Index of Figures

Figure 1 - Overarching abstract machine model from [3]. 7
Figure 2 - Overview of system call forwarding and process migration mechanism. 12
Figure 3 - Functional block diagram of the Texas Instruments 66AK2H14 SoC [30]. .. 13
Figure 4 - Detailed setup of the system call forwarding demonstration. 14
Figure 5 - System Components 15
Figure 6 - Diagram of KVM layers 16
Figure 7 - Nek5000 performance on 4 cores 17
Figure 8 - Nek5000 performance on 12 cores 17
Figure 9 – Relative performance of running Nek5000 in virtual machines 18
Figure 10 - Live migration duration 19
Figure 11 - Offline migration duration 19
Figure 12 - Application execution time with one live migration. 20
Figure 13 - Application execution time with one offline migration. 20
Figure 14 - Speedup of total execution time of live migration over offline migration. ... 20
Figure 15 - Latency between virtual machines and physical inter-host latency............ 21

Executive Summary

Due to the massive scale of envisioned exascale computing systems, hardware and software faults are expected to be the rule rather than the exception, making it necessary to improve the resiliency of exascale applications. However, current approaches to tolerate faults in HPC applications are limited to transparent hardware mechanisms that exhibit low overheads such as ECC protection of memories or network links, or global checkpoint/restart mechanisms to deal with errors at the application level. Due to the limited I/O bandwidth of exascale systems, the latter mechanism is expected to be unusable without significant improvements in scalability.

Proactive approaches to fault tolerance are based on predicting future failures and taking preventive action to avoid the impact of these failures onto running applications. These methods offer the possibility to reduce apparent fault rates seen by the application and thus lower the cost of reactive approaches such as checkpoint/restart mechanisms.

Large-scale parallel applications are almost universally composed of many independent processes executing on a distributed-memory machine. This allows all application processes to be evacuated from a node using process migration schemes to prevent the failure of the node from affecting the application. We investigated two different approaches to accomplishing this migration: Migrating bare-metal application processes in a heterogeneous system, and using industry standard virtualisation techniques to migrate complete virtual machines.

Proposed exascale system architectures use heterogeneous mixes of processing elements such as general-purpose CPU cores and GPGPU style accelerator cores to improve performance and energy efficiency, a trend already manifested in today's high-end HPC systems. Since GPGPU and other thin cores are likely to lack features needed for efficient execution of OS services, programs on such cores generally delegate the task to general-purpose CPU or fat cores. By extending this mechanism it is possible to decouple application processes from the underlying OS kernel and apply different resiliency strategies to both parts, which is the inspiration to the first approach we examined.

Virtualisation techniques are on the other hand at the core of the emerging cloud computing infrastructures that already today deploy data-centres with 100,000 servers, which is in the range of the expected node count for exascale systems. General-purpose CPU vendors have therefore included effective hardware support for virtualisation into their processor designs, which may be exploitable by HPC systems using the same hardware.

Our experiments indicate that both approaches are feasible and can be carried out without modification to the application code, which protects investments into application software. This flexibility comes at the price of some performance loss, which can be minimised by improvement of the underlying hardware and software. Further performance improvements could be possible by minor modifications to the application code to take problem-specific knowledge into account when migrating application processes. On the other hand, our findings indicate that unrelated application mechanisms, such as load-balancing or check-pointing, could also be exploited to implement process migration.

1 Introduction

For more than a decade, growth in computing capacity of HPC systems has primarily been accomplished by a matching increase in the number of concurrently executed operations, as opposed to an increase in throughput of a single functional unit due to higher clock speeds. For instance, the number one system on the Top500 list from November 2014, the Chinese Tinahe-2 computer, consists of 16,000 nodes containing in total more than 3.2 million independent cores [1]. This trend is expected to continue, leading to exascale computers consisting of about 100,000 individual nodes and causing renewed concerns regarding the reliability of such massive systems [2].

While hardware already makes extensive use of error correction technology to tolerate faults in both data transmission networks and data storage, in both on- and off-chip memory and in persistent secondary storage, HPC software is lagging behind. Many programs still silently ignore the issue, or employ a global strategy, most often checkpoint-restart, to deal with any errors. Both approaches are predicted to be prohibitively expensive at exascale.

In contrast to reactive fault tolerance, which attempts to repair the damage caused by failing components, the proactive approach to fault tolerance attempts to avoid damage altogether by taking advance action. At software level the most common approach is to abandon the component that is predicted to fail to allow it to be repaired or replaced without interrupting the on-going computation. This strategy requires the ability to rearrange the computation and also relocate any data, which can be accomplished by migrating some of the processes of a distributed application to unaffected, possibly spare, parts of the system. This process migration strategy is the focus of the presented investigation.

Proactive fault tolerance alone may not be sufficient to provide the required resiliency for exascale application; it can, however, be used to reduce the rate of application-visible errors which need to be handled by a reactive strategy. This may allow more resource-hungry fault handling strategies to be useable at larger scale, possibly allowing existing applications to be used reliably at exascale.

Forward-error-correction, which can be considered as a proactive fault tolerance schema, is already widely used in data communication networks to prevent costly, both in time and energy, retransmissions of corrupted messages or packets. The resulting increase in reliability allows for instance to reduce transmitter output power, allowing a reduction in the average energy per bit for the used communication channel. Since power and hence energy efficiency is considered another major challenge at exascale, improved fault tolerance may also be necessary to reach the exascale power limit of 20 MW.

Section 2 gives some background on relevant exascale hardware architecture trends, operating system characteristics and virtualisation techniques for HPC. Section 3 contains background on the approaches to fault tolerance relevant for the experiments. Section 4 presents the findings for bare-metal process migration on future thin cores. Section 5 shows the results for using virtualisation to realise process migration. Section 6 closes with our overall conclusions.

© CRESTA Consortium

2 Background

2.1 Exascale System Architecture

Projections of the hardware and system architecture features of future exascale computing systems were presented in the DARPA Exascale Study [2], which resulted in the presentation of an abstract architecture in a US Department of Energy report [3]. As illustrated in Figure 1, future exascale systems are envisioned to be highly heterogeneous with specialised processing elements that provide most of the computational capacity and a highly complex memory hierarchy incorporating non-volatile storage. Furthermore, cache-coherency even at a chip level is suggested to be too expensive at exascale. A survey of future directions from the project's perspective given in [4] and [5] indicates similar trends.

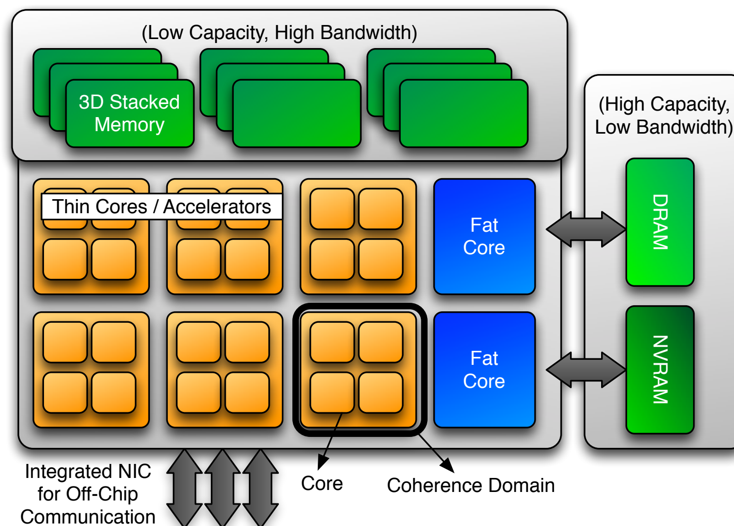

Figure 1 - Overarching abstract machine model from [3].

Increase in clock-frequency has significantly slowed down since 2004, and in the case of accelerators has even reversed. As a consequence, exascale systems are projected to concurrently execute several hundred millions to billions of operations. The amount of data processed in SIMD fashion is expected to increase with a Japanese system proposing a data width of 1024 to keep the number of execution threads at manageable levels [6]. At the same time advances in semiconductor technology will allow greater integration, which will in turn allow significantly more cores to be packed on a single chip, as illustrated by a recent research proposal of Nvidia to integrate 256+8 cores on a single chip manufactured in 7nm technology [7]. Despite this increase in per-chip computational power, the number of nodes in an exascale system is projected to be about 100,000. In contrast to computational power, memory capacity and bandwidth are not likely to increase proportionally, as already reflected in current trends.

The increase in system complexity, caused by the sheer number of components in an exascale system, is expected to severely decrease system reliability with extreme projections pointing at failures occurring roughly twice per hour [2]. While these expectations should be moderated by the fact that similar predictions for Petascale systems far exceeded actual error rates, it is likely that future systems will have to be more tolerant towards failures during operation. Stringent restrictions on total system power are also likely to reduce design margin and hence impact overall system reliability, for instance by heavily relying on error correction to allow reduction of link transmit power or supply voltages.

The relatively limited amount of global I/O bandwidth of likely exascale systems will require a more limited response towards single failures, with today's common global checkpoint-restart strategies becoming unfeasible.

2.2 Exascale Operating Systems

Following the hardware trend towards commodity products, operating systems used for HPC systems have shifted from closed-source products developed in-house (e.g. Cray UNICOS) towards open-source commodity server operating systems. This is illustrated by the dominance of UNIX-like Linux operating system among the current systems on the Top500 list (485 out of 500 systems, representing over 98% of the aggregated performance) [8].

Almost 4 decades ago, the design of the UNIX operating system was based around the concept of many users sharing access to a single computer that is equipped with a single CPU [9]. Modern UNIX-like Operating Systems such as FreeBSD [10] and Linux [11] have extended the model to include symmetric and non-uniform memory access multiprocessor machines, and make special provisions to handle multi-core and hyper-thread extensions in current hardware. Yet, the fundamental design to time-share the CPUs to handle kernel services remained fixed, and so did the definition of the kernel interface in the form of C-callable wrapper functions, syscalls, that enter the kernel via special hardware traps.

Unfortunately, the rich services offered by server-optimised operating systems require significant resources, both computationally and memory-wise, that can cause performance degradation when used at scale for HPC applications due to interference with the user-level application, an effect summarily called operating system noise [12]. Measures to reduce this interference for large-scale HPC machines can be classified according to the base operating system used to derive the scalable counterpart. In the full-weight kernel (FWK) approach, unnecessary services and system activities are as far as possible removed from an existing standard operating system. In contrast the light-weight kernel (LWK) approach aims at building a minimal operating system from scratch providing only necessary, but limited, functionality. A third option is to use co-operating FWK and LWK instances to provide richer functionality by delegating complex operations from the light-weight OS running on the application nodes to the full-weight OS running on specialised system nodes via system-call forwarding.

Separating the application from the full-weight OS has the additional benefit of providing rich services on architecturally-limited cores, for instance accelerator or power-optimised cores, as demonstrated by the FusedOS project [13]. A similar approach to off-load I/O operations was implemented on the BlueGene/P computer system [14]. The idea to intercept system calls to implement a distributed UNIX system was already implemented in 1987 [15].

In terms of reliability the heterogeneous approach allows to consolidate complex FWK-based OS images and provide a different fault tolerance strategy, perhaps also at a hardware level, from that used for the bulk application cores.

2.3 Virtualisation at Exascale

Virtualisation has successfully been used to isolate details of the hardware implementation from the user applications. For instance the now almost ubiquitous virtual memory allows the operating system to present a contiguous flat address space to the application while distributing the backing storage amongst available physical memory, greatly simplifying application programs. At the operating system level, the Hypervisor concept [16] allows whole operating system images to coexist on a single large computer as well as providing the basis for transparent management of these instances, as exploited by recent warehouse-scale Cloud infrastructure deployments.

Unfortunately for HPC, the associated overhead of virtualisation has caused scalability problems, as perhaps best exhibited by the TLB miss penalties triggered by large working set size applications [17]. OS level virtualisation has also been largely ignored

© CRESTA Consortium

due to the perceived overhead of virtualizing network I/O in current x86 hardware. However it has been shown that using optimised OS bypass techniques allowed to reduce the overhead for HPC benchmarks to around 3% [18].

Another interesting example is the Kitten lightweight operating system developed by Sandia National Laboratories which uses the Palacios virtual machine monitor jointly developed by Northwestern University and the University of New Mexico specifically for use in large-scale HPC [19]. Initial results indicate an overhead of less than 5% caused by the virtual machine layer, which suggests that otherwise observed overheads are due to software design issues rather than the virtualisation itself.

Datacenters for large-scale Cloud infrastructures, like those operated by Google, Facebook or Amazon, house up to 100 000 servers and use virtualisation techniques for ease-of-management. This indicates the possibility to successfully deploy virtualised HPC infrastructures of similar scale, which would translate to exascale machines. One interesting feature of virtualisation is the ability to migrate a virtual machine from one host to another host without the co-operation of the guest operating system. This capability could be used for providing fault tolerance.

Virtualisation can also increase node reliability by better isolating the different application domains from each other, thus potentially reducing the impact of software faults especially at operating system level.

3 Fault Tolerance Strategies

Fault tolerance is the ability of a system to continue to function correctly despite the presence of some malfunctioning or faulted parts of it. This feature is generally accomplished by exploiting redundant resources, which can either be extra computations, memory or time [20]. The level of tolerance is often specified in terms of the number of independent faults that can occur without impacting correct operation. Faults can be classified according to the behaviour into benign, or fail-stop, where a module stops producing output, and malicious, or byzantine, where a unit can produce arbitrary, but possibly reasonable, output. This later type of fault behaviour is much harder to detect and correct. Faults can exist both in hard- and software, with the latter often referred to as bugs.

A classic strategy to tolerate single faults is to carry out the computation at least three times on independent hardware and possibly also software, and use a majority vote to determine if the results were error-free, and further to select the correct result in the presence of a single fault. In this scheme, which is generally regarded as too expensive for HPC applications, detection of an error (a fault that is observable) is implicitly included; other schemes use separate error detection logic. Another popular technique is the use of extra storage to detect and correct single-bit errors using ECC codes. Since the overhead for these codes is much smaller, about one extra bit per byte, they are commonly used in HPC systems to protect system memory.

Reactive schemes attempt to detect and correct a fault after it has caused an error, while proactive schemes try to anticipate a likely future fault and take action while the system is still healthy. Therefore proactive schemas need to have a separate health checking system that can detect deteriorating system components before they cause errors. Often this is accomplished by monitoring the frequency of correctable errors (e.g. single-bit memory errors) [21]. Node failures can also be anticipated in a large number of situations [22], [23].

Algorithmic approaches to fault-tolerance can generally be classified as reactive. They differ from other techniques in the fact that they are self-healing and do not require separate detection: For instance a fault during the iterative approximation of a solution may be self-correcting and only result in extra iterations. Algorithms of this kind are for example studied in [24] and [25].

Fault tolerance implementations can also be classified according to the level of application visibility or involvement. Here transparent implementations, either system- or hardware-based, are completely invisible to the application. Common ECC memory protection schemes can be categorised as such, automatically correcting single-bit errors. User level implementations expose the errors towards the application but rely on lower-level software, for instance a library, to attempt correction. Many communication libraries use this approach. Finally application-level approaches expose the faults to the application software. This is often used by simpler operating system and library interfaces which for instance return an error condition in case an operation could not be performed. Unfortunately, applications are notoriously bad at handling errors and often completely ignore such conditions. The process migration approach that we focused on in this report can be implemented without changes to the application code but can be improved by incorporating application-specific knowledge.

Fault-tolerant implementations at any level require some basic guarantees from any underlying hardware or software. Most of the MPI library implementations that are a vital underpinning for many HPC applications make no guarantees whatsoever in the presence of any faults, effectively preventing any repair of the running application. Several efforts were made to remedy the situation in the past which would allow higher-level software to continue past a fault impacting the MPI layer and possibly even consider full-scale repair, for instance by incorporating new spare ranks to replace faulted ones [26]. So far these mechanisms have not been incorporated into the MPI standard.

3.1 Checkpoint-Restart Fault Tolerance

We want to single out checkpoint-restart approaches since those are popular in HPC. Here redundant storage is used to save the state of a long-running computational task at certain points in time, typically periodic. In case an error is detected, the last known good checkpoint is used to roll back the computation to an error-free state from where it is continued.

While this strategy can be simple to implement and can require relatively little in the way of extra resources, or in some cases none at all, it suffers from a major drawback: if the time to write a checkpoint plus the time to restart the computation approaches the mean time to fault, little or no progress is made. For exascale systems, this is the currently-predicted scenario when using global checkpoint-restart and efforts are undertaken to reduce the resources, both time and storage, needed to create and store checkpoint data to address the issue [27].

The node-level checkpoint-restart software developed by Berkeley Laboratories (BLCR) and popular in HPC applications is based on a cross-cutting architecture: a kernel module inside the Linux kernel is used to aid a user-level library to perform both checkpoint creation and application restarts. Since this combination can only handle a very limited set of services offered by the Linux kernel and user-level libraries, an interface is provided to allow extensions to be added to the mechanism that for instance can handle TCP connections [28]. OpenMPI up to version 1.6 uses the interface to support checkpoint-restart [29].

3.2 Migration for Proactive Fault Tolerance

In this work we consider migration to implement proactive fault tolerance for HPC applications. The principle of operation is that a node health checker indicates a likely future fault. Based on this indication the node is evacuated by a migration mechanism so that the computation can continue without interruption. It is possible to realise the migration using mechanisms already present in the application for other purposes.

If a checkpoint-restart mechanism is present, this can be used to create a checkpoint of the application state. The application can then be restarted on a new set of nodes thus avoiding the failing node. Two important optimisations are possible: First, since the node is still in working condition, checkpoint creation can be delayed for a short while to reduce complexity or checkpoint size. Second, only a partial checkpoint is needed since most of the nodes can simply pause computation during the migration operation.

In case a load-balancing mechanism exists, it can simply be used to assign all the work of the failing node to other nodes. Here the requirement is that the load-balancing implementation also moves the application state along with the work to fully evacuate the node. Furthermore some support from the communications subsystem may be necessary to exclude the node from message passing to prevent possible deadlock.

The third approach, which is also used by the experiments, is to pre-empt the computation to create a stable snapshot of the node's state and then replicate the state to a spare node. After this operation the computation can be resumed on the new nodes and terminated on the evacuated node. The main challenge of this approach consists in the generation of a consistent snapshot of the node's state. Simply halting the computation may cause in-transit messages to be lost and these messages need to be captured and replayed for correct operation. Furthermore it is desirable to halt only the local computation and for as short a period as possible.

4 Process Migration for Heterogeneous Systems

This approach takes advantage of the fact that future exascale nodes will likely incorporate a large number of thin cores or accelerators that delegate OS service requests to accompanying fat or general-purpose cores better suited for the task. The necessary infrastructure can be used to further separate the application from the OS kernel allowing different resiliency strategies to be used to protect the application and OS portions.

4.1 System Call Forwarding and Application Process Migration

The principal components used to forward system calls are shown in Figure 2. The application is distributed on several application nodes or cores that may use a different hardware architecture than the system nodes that execute the full weight kernel. On the application nodes, application processes are in full control of the complete node without interference from a kernel.

To access system services the application calls the runtime library included in the same process context. The runtime library then forwards the request to the system node responsible for servicing this application process via a user-level based bidirectional communications channel and waits for the corresponding response. This emulates the synchronous behaviour of UNIX system calls, however the runtime could also allow asynchronous calls by returning to the application without waiting for a response.

On the system node a proxy process waits for requests from the connected application process. This process provides the necessary context for the OS kernel to allow correct execution of the system calls. The user-mode runtime system executing on the system node receives the request and services it by extracting the parameters and executing the actual system call that transfers control to the kernel, which services the call.

To migrate the application process to a spare node, the process is first notified to block its execution. After that, the process image is copied to the new location and the communications channel is re-routed. Since the application process does not contain any in-kernel state, it can be migrated independent from the OS kernel.

Since application processes are expected to directly communicate with each other via a fast scalable interconnect, it is necessary to inform the other application nodes of the new location. These requests are expected to be handled by the MPI library layer.

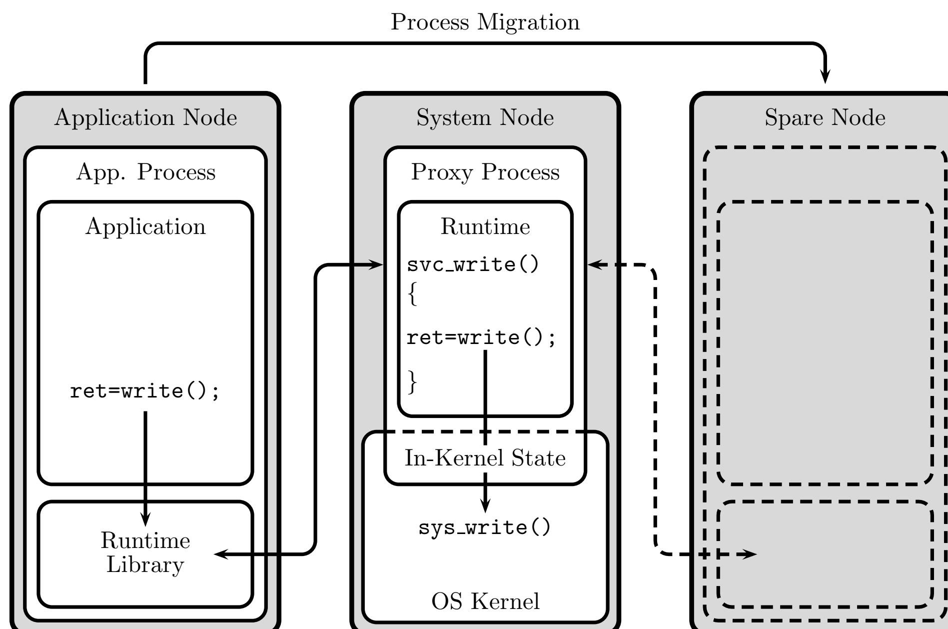

Figure 2 - Overview of system call forwarding and process migration mechanism.

© CRESTA Consortium

4.2 Hardware Set-up for System Call Forwarding Demonstration

We use the Texas Instruments 66AK2H14 SoC for our system call forwarding demonstrations. The chip, as shown in Figure 3, integrates four ARM Cortex-A15 cores, eight Texas Instruments C66x cores, 6 MB of on-chip shared memory and a large number of peripheral devices. External DDR3 memory can be attached via two independent single-channel memory controllers. Multiple SoCs can be connected via the two point-to-point HyperLink interfaces that permit access to the memory spaces of the link partner.

In our setup we use Revision 3.0 Advantech EVMK2H evaluation modules to support the SoC. The modules also house a 2 GB DDR3 SO-DIMM that is used as main system memory.

The SoC boots a modified Linux 3.17.0-rc3 mainline kernel obtained from the kernel.org Git repository (git.kernel.org). The modifications consist of the device driver kernel module developed in-house. The user-space file system is held in the initramfs and based on the minimal console image created by the Arago project and supplied via the Texas Instruments MCSDK version 3.00.03.15.

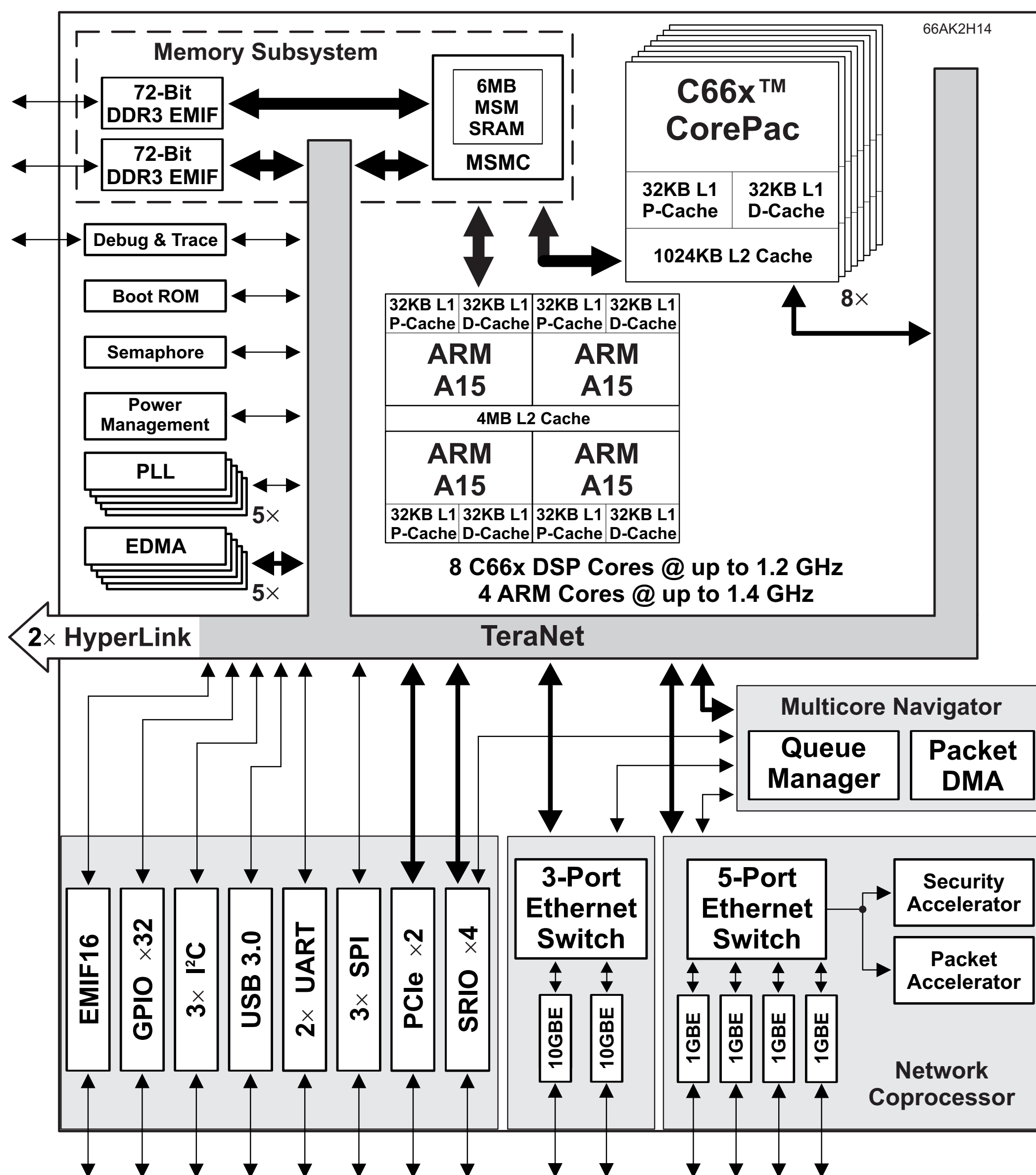

Figure 3 - Functional block diagram of the Texas Instruments 66AK2H14 SoC [30].

4.3 System Call Forwarding Demonstration on DSP Hardware

The detailed setup of the experiment to demonstrate the system call forwarding is shown in Figure 4. In this experiment the application process executes on a DSP core while the proxy process is running on an ARM core on top of the Linux kernel. Both processes execute on the same SoC and hence share the common system memory.

A custom developed device driver module (/dev/coproc) is loaded into the Linux kernel to allow management of the DSP cores using inter-processor interrupts. The module also allows allocation of system memory that can be shared between a Linux process and a DSP core using the mmap() system call.

To start a new process on the DSP core, the proxy process is started via standard UNIX methods (fork() via shell). This process starts executing ARM code (ARM-only .text in the figure), which opens the device file (/dev/coproc) to establish the DSP processor context. It then examines the DSP executable to be loaded and establishes the necessary memory segments shared with the DSP (DSP .text, .data, .stack). Since these are also mapped into the proxy process address space, they can be populated with program code and static data read from the DSP executable from the Linux managed file system. The proxy process also sets up a shared-memory queue pair to provide the communications channel for system call forwarding. Finally it starts the DSP process using an I/O control, ioctl(), call (start) to the device driver module that in turn takes the DSP core out of reset and directs code execution to the program entry point in the DSP .text section.

During normal operation the proxy process sleeps, waiting for requests from the DSP process via the wait ioctl(). The DSP initiates a system call by placing the necessary header and parameters onto the shared memory queue and signals the ARM core by raising an inter-processor-interrupt (IPI). This interrupt is handled by the coproc device driver in the Linux kernel that wakes up any processes sleeping in the wait ioctl(). Once the processes are woken up, they check if the IPI they were waiting for was raised and, if so, return to user mode signalling the new condition. The user-mode ARM code examines the shared memory queue and executes the requested system call (write() in the demonstration). After completion it writes the return value and eventual data onto the outgoing queue and uses the raise ioctl() to signal completion to the waiting DSP core again using IPIs.

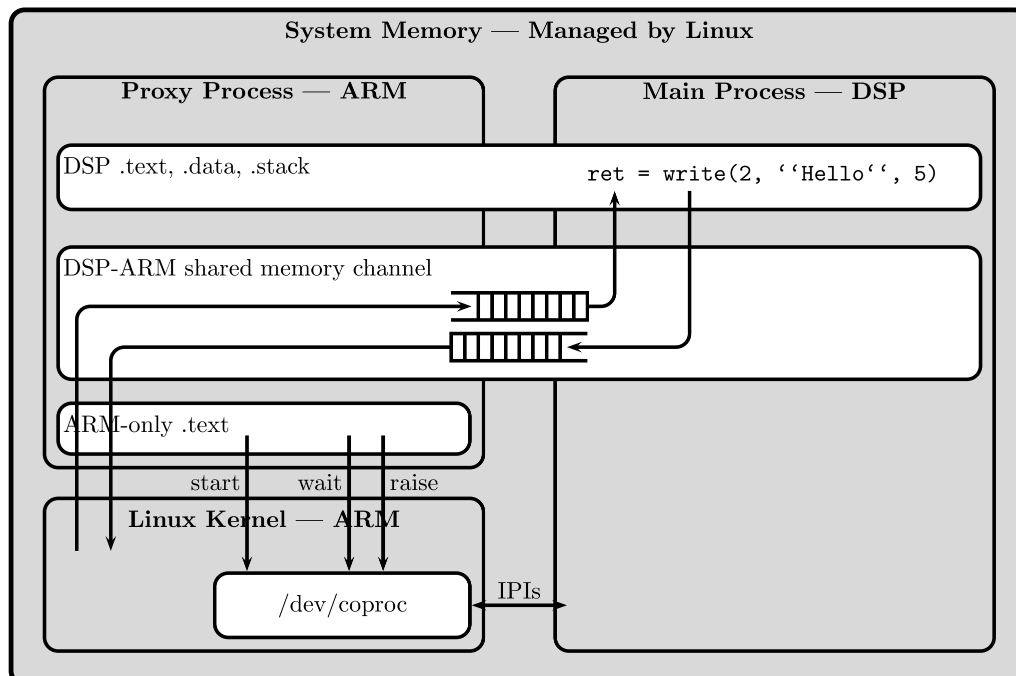

Figure 4 - Detailed setup of the system call forwarding demonstration.

5 Virtualisation-based Fault Tolerance on HPC Systems

Modern virtualisation environments often provide the ability to transparently migrate virtual machines between hosts, either off-line by pausing the VM or even “live” with just a minimal pause.

Here, we investigate the advantages of and capacity for using migration techniques in a virtualised HPC environment as a fault tolerance mechanism. In this investigation we have used Nek5000, a highly scalable HPC application. Nek5000 is an open-source computational fluid dynamics solver that employs MPI as the method to exploit parallelism [31]. Furthermore, Nek5000 is one of the CRESTA co-design applications with strong scientific need for exascale performance.

5.1 Test environment

An overview of the system elements employed to create a virtualised environment can be seen in Figure 5. We have used a front-end system that serves the shared file system (NFS in this case) across all nodes and virtual machines hosted on computing nodes. The virtualised system is equipped with two different compute nodes, a 24 core Intel Xeon X5650 at 2.67GHz and 48GB of memory and a 48 core AMD Opteron 6168 at 1.9GHz and 64GB of memory. Both nodes run Scientific Linux 6.5 and are interconnected by 10-gigabit Ethernet.

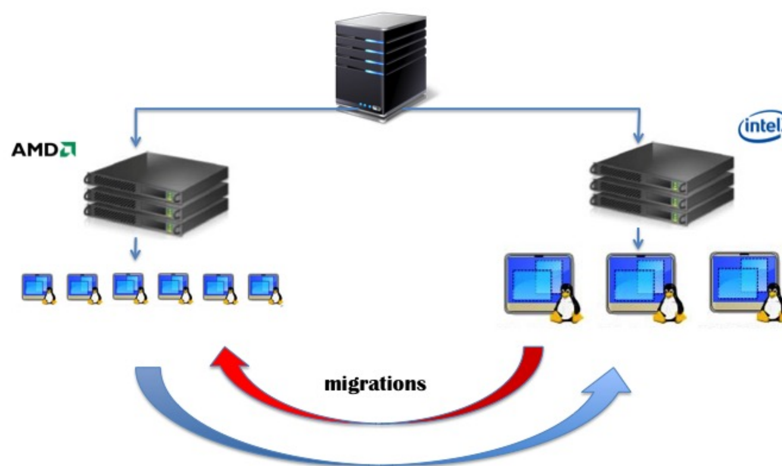

Figure 5 - System Components

The virtualised environment has been created through KVM version 0.12.1.2, kernel 2.6.32-431.29.2 which has been installed on both compute nodes. KVM is a full virtualisation solution for Linux on x86 hardware containing virtualisation extensions (Intel VT or AMD-V) [32]. It consists of a loadable kernel and processor module that provides the core virtualisation infrastructure. Through KVM, each virtual machine has its own private virtualised hardware. The diagram of layers in a regular KVM installation can be seen in Figure 6.

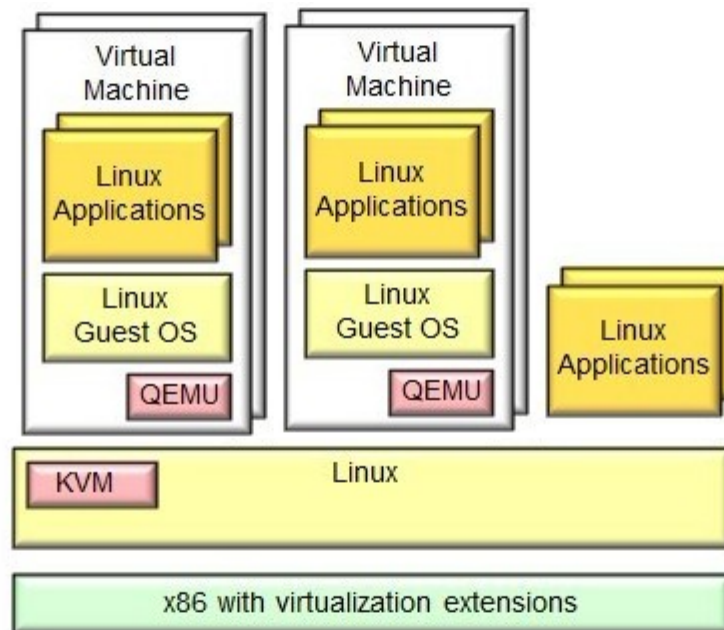

Figure 6 - Diagram of KVM layers

KVM was selected among other virtualisation solutions because it is capable of performing off-line and "live" virtual machine migrations. Furthermore, KVM is part of Linux and uses the regular Linux scheduler and memory management whereas other solutions are external hypervisors [32].

The virtual machines created are also configured to run the same version of Linux, Scientific Linux 6.5, and contain 512 MB of main memory. Compute nodes and virtual machines run MPICH 3.1 as main MPI package.

5.2 Virtualisation Overheads

While our main aim is to test the performance of Nek5000 in a pro-active fault tolerance virtualised environment, we first conducted a study comparing the performance of Nek5000 on a pure virtual environment and directly onto the physical platforms that virtualise the system. This will help us to understand how much our chosen application is influenced by running on a virtualised environment compared to a regular physical environment.

Therefore, we carried out a number of different tests executing the Nek5000 application on our virtualised environment and directly on one of the physical machines. We have also varied the number of MPI tasks per virtual machine (VM) to be able to understand how the number of MPI tasks per VM influences the performance of Nek5000.

In our first experiment, we run Nek5000 using 4 MPI tasks. Those MPI tasks have been launched on a) 4 VMs of 1 core each and b) 4 physical cores of the host machine, in this case on the AMD node. To assess the performance of our systems, we obtained the average time/timestep that Nek5000 provides when the simulation has finished.

In Figure 7 it is possible to see how the time/timestep varies depending on the system environment chosen. The bar on the left represents the performance result of running on 4 different VMs of 1 core each whereas the graph on the right shows the performance on the physical host employing 4 MPI tasks (one per core). The average time/timestep of running Nek5000 on this virtualised environment is around 0.59 seconds with a 3.84% of relative standard deviation (RSD) against 0.36 seconds and 0.89% RSD when running directly on the physical platform.

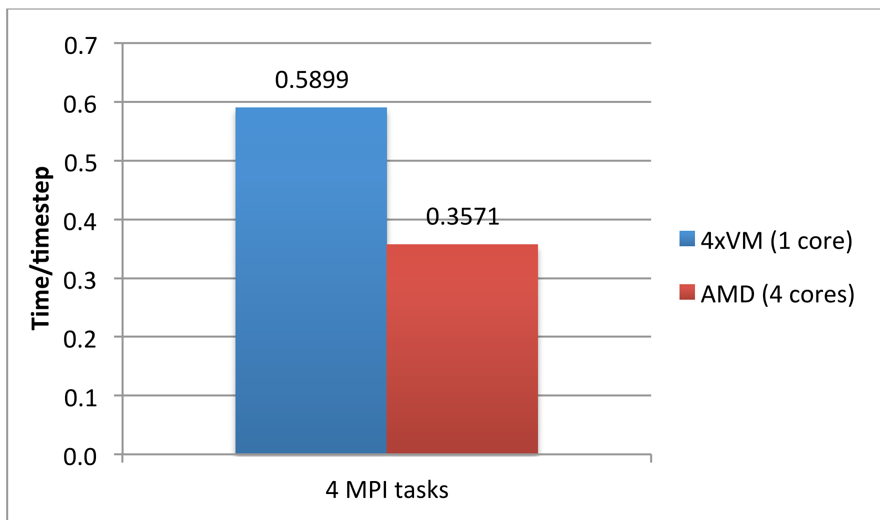

Figure 7 - Nek5000 performance on 4 cores

Although the performance of the application is reduced by almost 40% when running on a virtualised environment, it is perhaps the higher variability shown in the RSD that is more surprising.

In our second test, we have repeated the previous experiment increasing the number of MPI tasks to 12. The graph in Figure 8 illustrates the time/timestep of running Nek5000 on a VM composed of 12 cores and directly to the physical platform employing 12 MPI tasks, one per core. In this case the relative standard deviation is down to 0.12% on the virtualised environment and 0.67% on the physical machine.

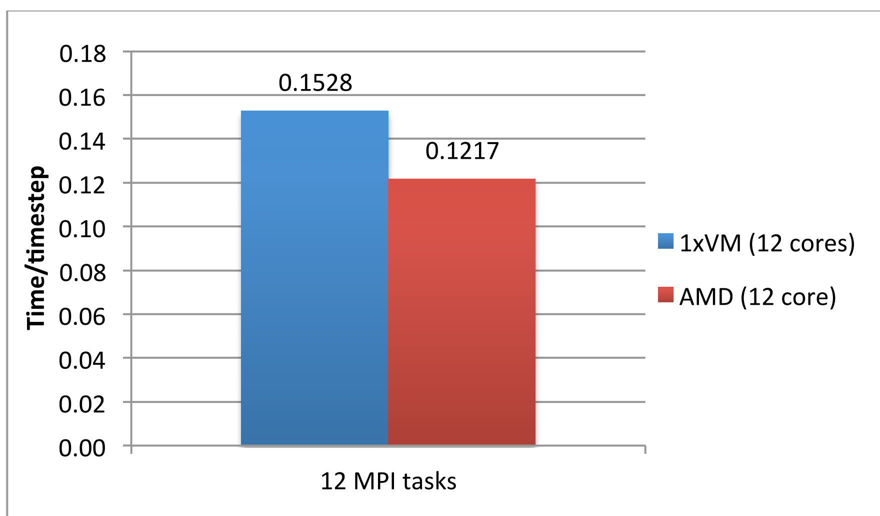

Figure 8 - Nek5000 performance on 12 cores

Figure 9 displays the relative performance of running Nek5000 inside a virtual machine versus the performance of running directly on the underlying hardware. The performance of running 4 MPI tasks inside of 4 single core VMs on the same node is about 39% lower than running the same tasks directly on the physical machine. This large overhead is most likely caused by the virtualisation of the MPI interconnect which prevents the use of shared memory. When instead running 12 MPI tasks in a 12-core virtual machine the overhead drops to about 20% compared to running directly on the physical hardware.

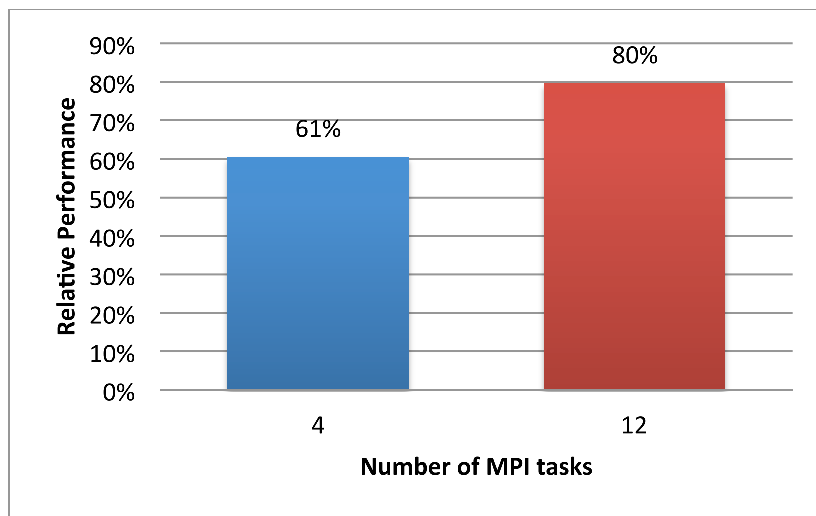

Figure 9 – Relative performance of running Nek5000 in virtual machines

5.3 Live and Offline Migration

One of the main features of virtualisation is that virtual machines can be moved or migrated from one physical host to another. KVM supports two varieties of migrations – live and offline. Offline migration moves a virtual machine from one host to another by pausing it, transferring its memory, and then resuming it on the destination host. Live migration does a similar operation, however the transfer process occurs without pausing the virtual machine. When performing a live migration, the applications running on the virtual machine continue with their execution.

The next set of experiments aims to obtain and estimate the overhead induced due to the migration (live and offline) of virtual machines. This would be the equivalent of having one imminent node failure. Using the Nek5000 application, we have forced the migration of one of the virtual machines from the Intel to the AMD computing node. Migrations were initiated after 100 timesteps for a simulation of 1000 timesteps. We used three different virtual machine configurations each with a total of 24 usable cores: one machine with 24 cores, two machines with 12 cores and four machines with six cores. For each of these configurations, a single virtual machine was migrated, thus for the latter two configurations inter-host traffic was directed over the Ethernet network. For each of the configurations we run Nek5000 using 4, 8 and 16 MPI tasks, leaving at least two cores on each virtual machine free for OS use.

We first consider the actual time it takes to perform the migration. During this time we would expect some impact on the performance of the application executing in the guest VMs. Furthermore this is the minimum advance warning time necessary to successfully evacuate a node before a failure. The time was measured as the wall-clock execution time of the virtual machine management command, virsh in our case, performing the migration. Figure 10 shows the time needed to perform the live migration. In this case, using a single large VM leads to a significant increase in migration duration. In fact, the time for migrating the 16 MPI task experiment was longer than the total simulation time. In contrast, Figure 11 shows the time used to perform an offline migration. Here execution time stays below 8.5 seconds. For the two smaller VMs, the offline migration is only slightly faster than the live migration. The difference in behaviour may be attributable to the fact that the large VM configuration did not have to use any inter-VM messaging and instead relied on shared memory to implement the MPI communication. Another interesting trend is that the migration time for the two larger VMs decreased as the number of MPI tasks increased, in contrast to the smallest VM configuration. Further experiments would be needed to explain this.

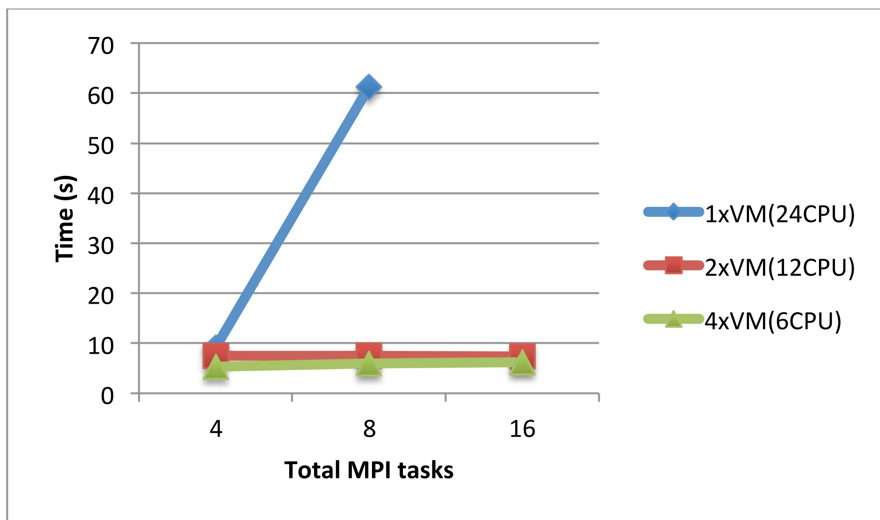

Figure 10 - Live migration duration

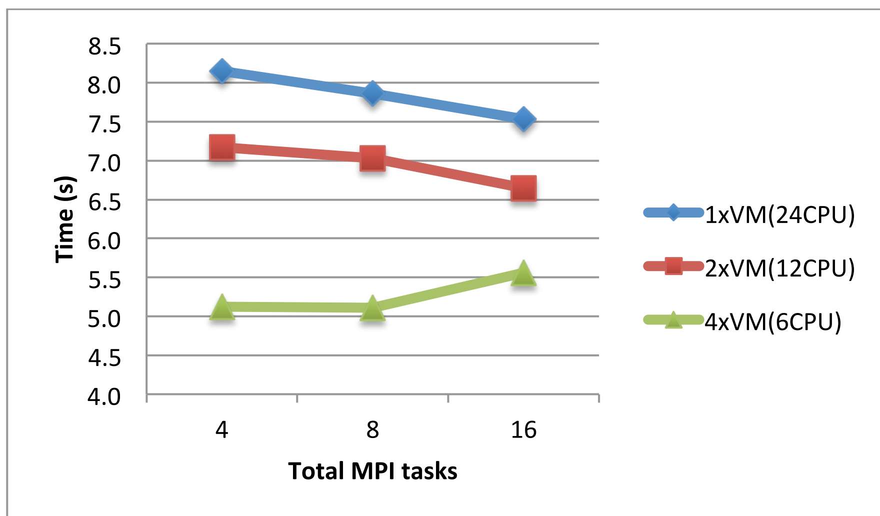

Figure 11 - Offline migration duration

The migrations also affect the overall wall-clock execution time of the application. We display this as the average wall-clock execution time normalised to a single timestep of the simulation. As mentioned all experiments ran for 1000 timesteps. Here we first must mention that the two nodes have different hardware with different performance characteristics affecting overall execution speed. Since only one VM is moved, we would expect this effect to be most pronounced for the single VM case, whereas the other two cases will show less effect due to the synchronizing nature of the messages exchanged between the MPI tasks on the different VMs. Since all migrations were triggered after 100 timesteps, the performance effects of the difference in node hardware should be roughly equal when comparing the live and offline cases, with the exception that live migrations for large VMs take significantly longer. Thus for instance the performance for the 16 MPI tasks shown in Figure 12 is not affected by the performance on the destination node since migration only completed after the simulation ended. In contrast to Figure 12, which shows the situation when performing live migration, Figure 13 shows the impact of the offline migration onto the application execution.

Figure 14, where we show the total speedup of performing offline migrations over live migrations, compares the two previous situations. Although there are no significant differences in the total performance of the application, it seems that live migrations are more beneficial in cases where more than one virtual machine has been employed. Also, the number of MPI tasks plays an important role in the total performance where live migration runs obtain higher speedups.

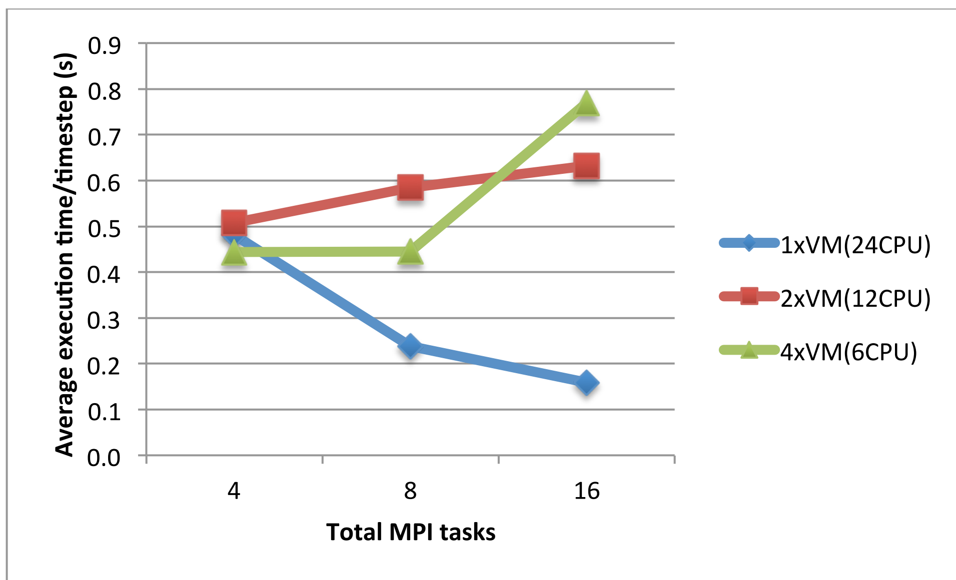

Figure 12 - Application execution time with one live migration.

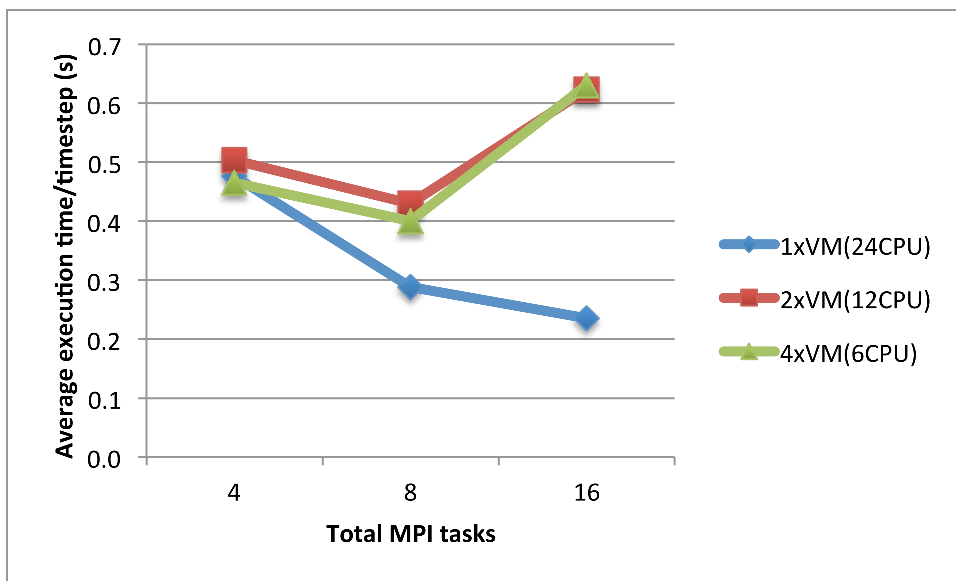

Figure 13 - Application execution time with one offline migration.

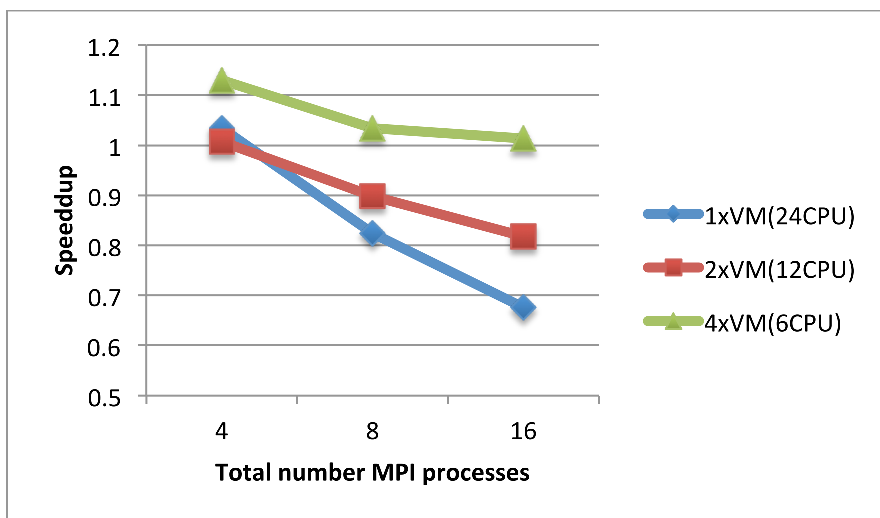

Figure 14 - Speedup of total execution time of live migration over offline migration.

5.4 Pro-active Migration

Monitoring health and detecting degrading status can often anticipate node failures on new hardware. These techniques, in addition to live and offline migrations would help to exploit a pro-active mechanism for fault tolerance. Instead of reactive schemes where applications are stopped and continued after an occurred failure, here we propose a system which would detect nodes' health and would migrate from deteriorating nodes to healthy ones before the failure has already occurred.

5.5 Limitations of the Approach

One of the main obstacles we faced during the execution of all test experiments was that root access was required in a number of situations. For instance, migrations and virtual machine configurations required becoming root. Only system administrators are granted those privileges on production HPC systems, and thus testing the approach on large-scale systems becomes a difficult task.

Although KVM allows migrations from an AMD host to an Intel host and back, we experienced a number of problems performing live migrations from an AMD host to an Intel host. Virtual machines often hung and a reboot was required after attempting live migration. The KVM error seemed to be related to a synchronisation error between hosts, however clocks were properly synchronised. Since these problems have not been the case when migrating from an Intel host to an AMD host, we continued testing only this direction.

In addition to the actual migration overhead, the total time involved in the application execution largely depends on the application itself and the network latency. Although Nek5000 is a highly-scaling application it is unlikely it scales well in a virtualised environment and latency is one of the main problems. As depicted in Figure 15 latency has played an important role in our experiments and that is suspected to be the main reason for the increased total execution time on Nek5000 as we increased the number of MPI tasks involved in the simulation.

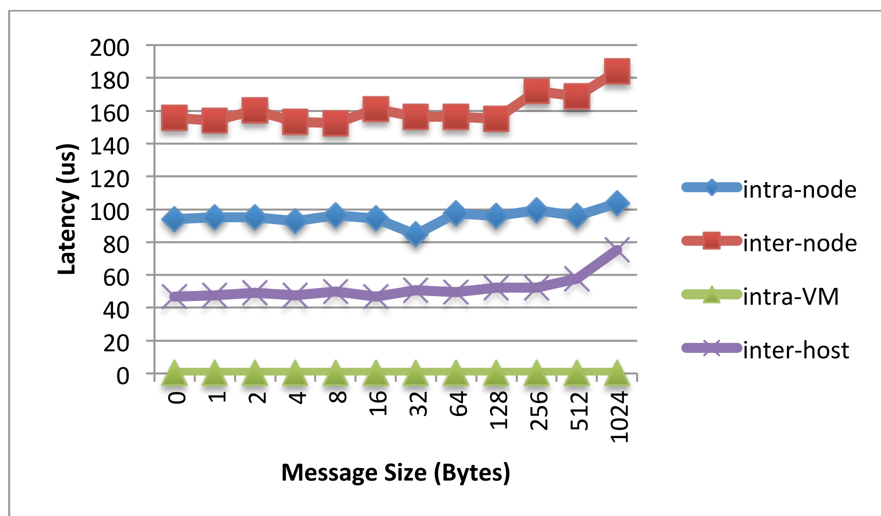

Figure 15 - Latency between virtual machines and physical inter-host latency.

Using the Ohio State University micro-benchmark [33] we have measured the latency in three different scenarios: Between virtual machines hosted in different nodes (inter-node), between two virtual machines on the same node (intra-node) and between two cores within the same virtual machine (intra-VM) as shown in Figure 15. For comparison the latency between the non-virtualised hosts is also included (inter-host). We observe about 100 microseconds of increase in the latency time between intra-VM and intra-node. That large increase of latency time is probably responsible for the lack of scalability of Nek5000 in this virtualised environment.

© CRESTA Consortium

6 Discussion and Conclusions

Current projections for exascale systems indicate that faults are likely to happen even during the execution of a single application job, and that current global fault tolerance approaches require too much extra time and I/O resource to scale to expected levels of concurrency. Proactive approaches to fault tolerance can reduce the rate of errors seen by the application and therefore reduce the cost of existing reactive schemes. This could extend the scalability and lifetime of already implemented fault tolerance measures as well as providing a richer design space for novel solutions.

6.1 Impact on CRESTA Applications

Literature and our experiments indicate that process migration can be carried out without application involvement, which allows the implementation of a migration-based scheme without changes to the application. However, our virtualisation experiments show that application performance is sensitive to the details of the VM setup and it may thus be necessary to tune application and virtualisation layer together to achieve optimal performance.

Application-specific knowledge can generally be used to reduce the overhead of fault tolerance techniques, for instance by reducing the amount of state to save. For process migration, already-existing application features, such as load-balancing or check-pointing, could be used to improve efficiency, which may require changes to the application code. For advanced applications that adapt to the topology of the underlying network hardware, more extensive two-way interactions between the migration engine and the application may be needed for instance to migrate even unaffected processes to maintain nearest neighbour relationships. These interactions could also be integrated into suitable application frameworks or libraries.

The GROMACS, HemeLB and OpenFOAM applications already support checkpoint-restart allowing periodic creation of checkpoints. They could take advantage of information from a health-checking system to tailor checkpoint frequency, or the implementations could be used to write a checkpoint to implement process migration.

6.2 Impact on CRESTA Software-stack

While it may be possible to completely implement process migration in user-space, it is likely and indeed indicated by our experiments that some assistance from the OS or hypervisor layer is required for high-performance implementations. Furthermore user-space libraries may be needed to adopt to new conditions and possibly to prepare the system before migration. This is especially likely in the case of direct user-level access to hardware such as network interfaces or accelerators.

Comparison with the Kitten lightweight OS suggests that optimisation of system software and device drivers to increase performance should be feasible.

The absence of even the most basic guarantees in the presence of faults in many popular MPI implementations may at least be partially responsible for the popularity of global checkpoint-restart approaches. The lack of defined behaviour forces applications to treat any fault as catastrophic and abort the whole computation as soon as possible and restart from a last known good state. Proactive approaches are compatible with this type of implementation since they react while the library and application are still in a working state, but more subtle reactive approaches need more clearly-defined behaviour. Several efforts have been undertaken in the context of the MPI libraries, but standardisation and implementation seem to be stalled at the moment.

6.3 Impact on Hardware

Current x86 processor implementations, both from AMD and Intel, have hardware support for virtualisation; for general device hardware, such as network interfaces and accelerators, more support may be necessary to improve performance. This could however also be a software or device driver issue.

© CRESTA Consortium

Hardware support for remote memory access could be used to implement zero-copy techniques for transmitting bulk data from the application to the system nodes. This could significantly lower the overhead of data-intensive (I/O) system calls. Similar network techniques could also be used to increase the performance of state-replication between the system nodes if desired. Furthermore the system node in particular could benefit from a highly scalable system to deliver events to the waiting proxy processes.

References

[1] Jack Dongarra, "Visit to the National University of Defense Technology Changsha, China," Oak Ridge National Laboratory, June 3, 2013.

[2] Peter Kogge et al., "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," DARPA Information Processing Techniques Office and Air Force Research Laboratory, September 28, 2008.

[3] J.A. Ang et al., "Abstract Machine Models and Proxy Architectures for Exascale Computing Rev. 1.1," Sandia National Laboratories and Lawrence Berkeley National Laboratory, May 16, 2014.

[4] Stephen Booth, "D2.1.1 – Architectural Developments Towards Exascale," CRESTA, 2012.

[5] Jeremy Nowell et al., "D2.1.2 – Architectural Developments Towards Exascale," CRESTA, 2013.

[6] Mitsuhisa Sato. (2014, April) A report on Feasibility Study on Future HPC Infrastructure. [Online]. http://www.ccs.tsukuba.ac.jp/eng/wordpress/wp-content/uploads/2014/04/CCS-MitsuhisaSato.pdf

[7] Oreste Villa et al., "Scaling the Power Wall: A Path to Exascale," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, 2014, pp. 830-841.

[8] (2014, November) Top500. [Online]. http://www.top500.org/lists/2014/11/

[9] Maurice J. Bach, *The Design of the UNIX Operating System*. Upper Saddle River, NJ, USA: Prentice Hall, 1986.

[10] Marshall Kirk McKusick and George V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Reading, MA, USA: Addison-Wesley, 2005.

[11] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2006.

[12] Dan Holmes, "D2.3.1 – Operating systems at the extreme scale," CRESTA, 2013.

[13] Yoonho et al. Park, "FusedOS: Fusing LWK Performance with FWK Functionality in a Heterogeneous Environment," in *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, New York, NY, 24-26 October 2012, pp. 211-218.

[14] Venekatram Vishwanath et al., "Accelerating I/O Forwarding in IBM BlueGene/P Systems," in *Proceedings of the 2010 ACM/IEEE Conference for High Performace Computing, Networking, Storage and Analysis SC'10*, Washington, DC, 2010, pp. 1-10.

[15] J.P. Black, L.F. Marchall, and B.Randell, "The Architecture of UNIX United," *Proceedings of the IEEE*, vol. 75, no. 5, pp. 709-718, May 1987.

[16] Robert P. Goldberg, "Survey of virtual machine research," *Computer*, vol. 7, no. 6, pp. 34-45, June 1974.

[17] Alessandro Morari et al., "Evaluating the Impact of TLB Misses on Future HPC Systems," in *2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai, 21-25 May 2012, pp. 1010-1021.

[18] Arun Babu Nagarajan, Frank Mueller, Christian Engelmann, and Stephen L Scott,

"Proactive Fault Tolerance for HPC with Xen Virtualisation," in *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*, Seattle, WA, 18-20 June 2007, pp. 23-32.

[19] John Lange et al., "Palacios and Kitten: New High Performance Operating Systems For Scalable Virtualized and Native Supercomputing," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, Atlanta, GA, 19-23 April 2010, pp. 1-12.

[20] Israel Koren and C. Mani Krishna, *Fault-Tolerant Systems*.: Morgan Kaufmann, 2010.

[21] Carlos H. A. Costa, Yoonho Park, Bryan S. Rosenburg, Chen-Yong Cher, and Kyung Dong Ryu, "A System Software Approach to Proactive Memory-Error Avoidance," in *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, New Orleans, LA, 2014, pp. 707-718.

[22] H. Song, C.B. Leangsuksun, R. Nassar, N.R. Gottumukkala, and S. Scott, "Availability modeling and analysis on high performance cluster computing systems," in *The First International Conference on Availability, Reliability and Security (ARES)*, 20-22 April 2006, p. 8.

[23] C. Leangsuksun, A. Tikotekar, V. Rampure, S. Scott S. Rani, "Toward efficient failure detection and recovery in HPC," in *High Availability and Performance Computing Workshop*, 2006.

[24] Mark Bull and Jeremy Nowell, "D2.5.1 –Fault Agnostic and Asynchronous Algorithms at Exascale," CRESTA, 2013.

[25] Piyush Sao and Richard Vuduc, "Self-stabilizing Iterative Solvers," in *ScalA '13 Proceedings of the Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, Denver, CO, pp. 4:1-4:8.

[26] Wesley Bland et. al, "An evaluation of User-Level Failure Mitigation support in MPI," *Computing*, vol. 95, pp. 1171-1184, 2013.

[27] Sheng Di, M.S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimisation of Multi-level Checkpoint Model for Large Scale HPC Applications," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, Phoenix, AZ, 19-23 May 2014, pp. 1181-1190.

[28] Jason Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Lawrence Berkeley National Laboratory, LBNL-54941, December 2002.

[29] OpenMPI. [Online]. https://www.open-mpi.org/faq/?category=ft

[30] Texas Instruments, "66AK2H14/12/06 Multicore DSP+ARM KeyStone II System-on-Chip (SoC)," Texas Instruments, SPRS866E, November 2013.

[31] James W. Lottes and Stefan G. Kerkemeier Paul F. Fischer. (2008) nek5000 Web page. [Online]. http://nek5000.mcs.anl.gov

[32] KVM - Kernel Based Virtual Machine. [Online]. http://www.linux-kvm.org/page/Main_Page

[33] MVAPITCH Benchmarks. [Online]. http://mvapich.cse.ohio-state.edu/benchmarks/

Glossary of Acronyms

AMD	Advanced Micro Devices Incorporated
ARM	ARM Holdings
BLCR	Berkeley Laboratories Checkpoint-Restart
CPU	Central Processing Unit
D	Deliverable
DARPA	Defense Advanced Research Projects Agency
DDR3	Double Data Rate 3 (memory technology)
DSP	Digital Signal Processor
EC	European Commission
ECC	Error Correction Code
FWK	Full Weight Kernel
GPGPU	General Purpose Graphics Processing Unit
HPC	High Performance Computing
IPI	Inter Processor Interrupt
I/O	Input/Output
KVM	Kernel Virtual Machine
LWK	Light Weight Kernel
MCDSK	MultiCore Software Development Kit
MPI	Message Passing Interface
NFS	Network File System
OpenMPI	Open Message Passing Interface (MPI)
OS	Operating System
PM	Project Manager
RSD	Relative Standard Deviation
SIMD	Single Instruction Multiple Data
SoC	System-on-Chip
SO-DIMM	Small Outline Dual Inline Memory Module
TCP	Transmission Control Protocol
TLB	Translation Look-aside Buffer
VM	Virtual Machine
WP	Work Package

© CRESTA Consortium