LEGaTO

Low Energy Toolset for Heterogeneous Computing

D3.4 "REPORT ON EVALUATION AND OPTIMIZATIONS IN THE RUNTIME STACK"

Version 1.0

Document Information

Contract Number	780681
Project Website	https://legato-project.eu/
Contractual Deadline	02 Nov 2020
Dissemination Level	Public
Nature	Report
Author	Miquel Pericàs (CHALMERS)
Contributors	Do Le Quoc (TUD), Xavier Martorell (BSC), Leonardo Bautista-Gomez (BSC), Behzad Salami (BSC), Miquel Pericàs (CHALMERS), Mustafa Abduljabbar (CHALMERS), Pirah Noor Soomro (CHALMERS), Jing Chen (CHALMERS), Gunnar Billung-Meyer (CHR), Saber Nabavi (BSC), Paul Carpenter (BSC)
Reviewers	Valerio Schiavoni (UNINE), Nils Voss (Maxeler)

The LEGaTO project has received funding from the European Union's Horizon 2020 research and innovation programme under the Grant Agreement No 780681.

Change Log

Version	Description of Change
1539	2020-10-23, File created
1551	2020-10-27, Intro to FPGA checkpointing
1553	2020-10-27, FPGA undervolting results
1569	2020-10-28, SGX framework comparison added
1583	2020-10-29, Add XiTAO energy and pipeline parallelism results
1589	2020-10-30, Add XiTAO topologies evaluation
1620	2020-11-03, Add execute summary
1646	2020-11-06, List Valerio as a reviewer and update exec summary
1672	2020-11-10, Add an introduction
1679	2020-11-11, Update the introduction
1680	2020-11-11, Conclude the final deliverable
1689	2020-11-12, Include the SLRUM-RECS integration
1691	2020-11-12, Update introduction and summary to reflect changes
1831	2020-11-30, Apply internal edits by Osman

This log reflects actual revision numbers from SVN (version control software used).

Index

List of Figures

List of Tables

1. Executive Summary

In this deliverable, we highlight the evaluations and optimizations on top of the final release of the LEGaTO backend and middleware.

On the middleware side, we have integrated the Slurm workload manager with the RECS Master. This approach allows users to request resources using Slurm for jobs to be run in the cluster. Also, more resilience against data loss and enhanced reconfiguration capabilities have been promoted in the case of a power shutdown of the RECs box server.

In the backend, XiTAO software topologies have been validated using SparseLU and Heat diffusion benchmarks, and have shown adaptivity and consistent speedup compared to traditional locality-aware and work-stealing schemes. Thorough energy-efficient scheduling evaluation is presented for VGG16, Heat and SparseLU. The evaluation is carried out on both asymmetric and symmetric platforms, and is compared to energy-efficient and work-stealing schedulers. Overall, from 31% to 75% energy reduction is achieved in comparison to the baseline schedulers.

Further optimizations are developed on top of OmpSs backend. The performance of OmpSs@FPGA and CUDA is showcased for dense matrix multiply that appears in applications such as Convolutional Neural Networks (CNNs). Comparison of energy/performance gains using the Xilinx ZCU102 development kit have shown an efficiency of up to 7.32 GFlops/W using 2xIP core instances. The distributed variant of the OmpSs, *i.e.*, OmpSs@Cluster is evaluated using the Smart-Mirror use-case. Both the single and OmpSs-2@Cluster implementations hit the limit frame rate of the camera, which is 30 FPS. The energy consumption was 25W for the single node implementation, using the OmpSs-2@Cluster this dropped to 13W for the first node of which 10W is used for the processing of the raw captured image (*i.e.* scaling), and total of 17W for the second Xavier node.

Several evaluations of fault tolerance techniques are presented. Undervolting is demonstrated as an effective energy saving technique. Previously we had focused on the SRAM-based memories of FPGAs, while in this deliverable we extend this work to include High-Bandwidth Memory (HBM). Experimental results show a 2.3× energy saving for HBM at all bandwidth utilizations. The checkpointing library is now capable of checkpointing CPU, GPU and FPGAs under the same API in a completely transparent fashion. It has demonstrated the low-overhead of our approach, making heterogeneous computing more resilient and fault-tolerant. Last but not least, we provide an extensive performance comparison between state-of-the-art Intel SGX frameworks using our monitoring tool TEEMon, the first continuous performance monitoring and analysis tool for TEE-based applications.

By combining the various techniques introduced at the runtime and firmware levels in deliverables D3.3 and D3.4, we can achieve an order of magnitude reduction in energy consumption. Hence, we are confident that the energy efficiency goals of LEGaTO have been achieved.

2. Introduction

In the previous deliverables of work package 3 "Tool-Chain Back-End", we have motivated and described the implementations behind the LEGaTO task-based runtime environments on heterogeneous platforms featuring asymmetric CPUs, GPUs and FPGAs. These implementations target achieving up to 10× energy efficiency on the next generation IoT, Edge and HPC applications, as well as addressing security and fault-tolerance concerns. Thanks to the enhanced middleware interface, the reconfigurable hardware resources provided by Maxeler and Christmann are available to validate the backend contributions. Therefore in this final deliverable, we report on evaluations and optimizations for energy-efficiency, security, and resilience on top of the final release of the task-based runtime (D3.3 [25]). In general, this document highlights

1. Optimizations added to techniques described in D3.3,
2. Detailed evaluation of techniques from D3.3 or the new optimizations developed after D3.3, and
3. Incremental proposals that pertain to the parts in D3.3.

In Section 3.1, we describe and showcase the Slurm-RECS integration, which makes the heterogeneous LEGaTO hardware resources available to the de facto scalable cluster management and job scheduling utility. Section 3.2 demonstrates the robustness and usability of the RECS_Master management software and RedFish API to facilitate integration with the LEGaTO backend.

Section 4 focuses on the two runtimes researched in LEGaTO: the experimental XiTAO runtime and the production OmpSs runtime. We start by listing the optimizations and evaluations of the XiTAO backend in Section 4.1. These include an evaluation of the XiTAO topologies made available through dynamic locality-aware scheduling. Further, we include a thorough analysis of the energy-aware scheduler (EAS) as well as the pipeline parallelism. The techniques have been applied on representative HPC and machine learning benchmarks, running on an edge NVIDIA TX2 development board, one of the target platforms of the LEGaTO project.

The energy efficiency of the OmpSs backend is evaluated in Section 4.2. Starting with section 4.2.1, we evaluate OmpSs@FPGA & CUDA by executing a set of matrix multiplication experiments on the Xilinx ZCU102 development kit, with 4 ARM A53 cores and an integrated XCZU9EG FPGA. Each matrix multiplication experiment consists of 20 executions of matrix multiply operations, on a matrix of 2816x2816 single precision floating point elements. The matrix is blocked in tiles of 256x256 values. Insights are made about the performance/energy tradeoffs when tuning the amount of IP and CPU resources allocated to the described workload. The OmpSs-2@Cluster implementation is analyzed using the Smart Mirror use case in Section 4.2.2. OmpSs-2@cluster allows the CPU and GPU compute to be divided among two nodes to increase the frame rate with relatively lower increase in power consumption. This was possible using the annotations described in Section 4.2.3.2 at minimal development cost.

Section 5.1 gives more detail on the methodology followed to implement FPGA checkpointing. The work is separated into two different main approaches when it comes to implementing FTI functionality: host-based checkpointing and checkpointing of partial data from the FPGA task. We perform the experiments on a cluster of 4 Xilinx Zynq-7000 SoC nodes via the OmpSs@FPGA

environment. Comparisons by enabling and disabling the Fault Tolerance Interface (FTI) library are performed for YUV filter, Jacobi Solver and K-Means Clustering.

Then, in Section 5.2, we focus on undervolting High-Bandwidth Memory (HBM). It is known that modern high-performance devices employ HBM in order to meet high memory bandwidth requirements, but consume a substantial portion of the chip's power budget. By means of undervolting, our measurements show that the guard-band region in our HBM modules is 19% of the nominal voltage; bringing the supply voltage down to that region provides a 1.5× power saving gain for all bandwidth utilizations. Pushing voltage down further by 11% provides 2.3× power saving at the cost of unwanted bit flips. We explore and characterize the rate and type of these bit flips and show that the majority of them are clustered together in less than 16% of the memory's entire address space.

Finally, Section 5.3 analyzes the overheads by the developed security mechanisms based on TEEs, *e.g.*, Intel SGX or ARM TrustZone. The performance monitoring framework TEEMon [16], which has been reported in Deliverable D4.2, is used to measure the overhead of Intel SGX frameworks such as SGX-LKL [26], and Graphene-SGX [29] and as well as our own toolchain SCONE [4]. The depicted performance metrics are helpful for developers using the SGX framework in order to identify performance bottlenecks especially considering scarce resources such as EPC (Enclave Page Cache) memory and the expensive enclave exit and enter operations (due to system calls).

3. Middleware integration and improvement

3.1. SLURM integration with the middleware

We have integrated the Slurm workload manager [32] with the RECS Master service. This approach allows users to request resources using Slurm for jobs to run in the cluster. The Slurm manager selects the nodes where jobs can be run, starts the nodes, run the jobs, and stops the nodes upon completion. After that, the nodes can be used by the next job submitted.

3.1.1. Slurm configuration

The configuration of Slurm is provided in the `etc/slurm.conf` file. It provides the list of nodes available in each baseboard, and their hardware characteristics, that will be used in user jobs to decide which nodes should be selected. A baseboard is one of the slots available in the hardware system to allocate each one of the computing nodes. This information can be obtained at installation time, by using the Redfish API to obtain the hardware inventory information available in the target system.

For example, in a common Slurm file, we have:

- NodeName=BB_1_[0,2-15] Sockets=1 CoresPerSocket=4 ThreadsPerCore=1 Feature=ARM,bigLITTLE,hasGPU
- NodeName=BB_2_[0-7] Sockets=2 CoresPerSocket=2 ThreadsPerCore=1 Feature=x86_64

It describes a collection of 15 nodes, named BB_1_0 to BB_1_15, with the exception of the non-existing BB_1_1. Those nodes have 1 socket (chip), with 4 ARM cores, and a single thread per core. Additionally, each core has several big cores and additional LITTLE cores, and hosts a GPU. The Slurm features are arbitrarily set per organization, so they are adapted to the target environment.

The second collection of nodes (BB_2_0 to BB_2_7) have 2 sockets with 2 cores per socket, and their hardware architecture is x86_64.

The Slurm configuration file also groups nodes in partitions:

- PartitionName=debug Nodes=BB_1_0,BB_1_[2-15],BB_[13-18]_[0-7] OverSubscribe=EXCLUSIVE Default=YES MaxTime=INFINITE

This describes the *debug* partition, with the collections of nodes BB_1_*, BB_13_* to BB_18_*, not allowing oversubscribing, and with no execution time limit.

3.1.2. Job descriptions

Slurm jobs are annotated with the node characteristics requested. In order to do this we use the Slurm node features specified above in the node definitions, through the *constraint* field:

```
#!/bin/bash
#SBATCH -N 4
#SBATCH --constraint=ARM
#SBATCH -o test-%j.out
#SBATCH -e test-%j.err
... job commands ...
```

3.1.3. Job flow

Nodes are turned off by default. When a job is submitted, Slurm takes care of selecting the nodes to be awakened and powers them up using the Redfish API. Once they are up and ready (running the slurm daemon), the job is launched. At job termination, and after a short idle period, the nodes are shutdown, and stay off till they receive a new job assignment.

This is implemented with a series of scrips executed by slurm, that contact the RECS|Box management system through its webservice. Figure 3.1 shows the high-level view of this environment.

The user submits slurm jobs using batch job scripts specifying the desired conditions that the nodes must accomplish. In the example, the user requests nodes that are described as ARM, bigLITTLE, and hasGPU. Slurm contacts the RECS Master server to power up and run the jobs. The user can follow the job execution using the Slurm tools. *squeue* shows the jobs enqueued, some of which may be running, others waiting for resources. Finally, *sinfo* that shows the partitions available, and the state of the nodes belonging to them.

3.2. Improvement of hardware management and APIs

In the course of the middleware evaluation, its robustness and usability were continuously improved to allow a seamless integration with the rest of the stack.

The RECS_Master management software is the main authority for controlling the hardware resources and communication infrastructure within the RECS|Box and t.RECS servers. Therefore, it is also responsible for configuring the Ethernet and PCIe switches to separate the network between the microservers with VLANs and compose nodes by managing PCIe functions, respectively. In contrast to all other data held by the RECS_Master, the VLAN and PCIe configuration are no physical property of the hardware and thus volatile in terms of power loss. Hence, the configuration data has to be persisted to be recovered after the power is back again. While the obvious

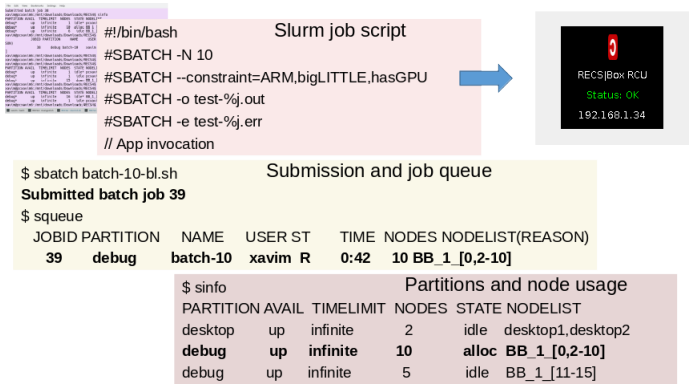


Figure 3.1. Job description, submission to slurm, and interaction with RECS Master

and common reason for power-loss is a reboot or shutdown of the RECS|Box or t.RECS platforms, the former system has a second cause. Due to its blade-like architecture, the RECS|Box allows hot-plugging whole baseboards with multiple microservers and dedicated communication infrastructure on it. Both scenarios are now protected against data loss by persisting volatile information. When the power is on again, the RECS_Master reconfigures all Ethernet and PCIe switches of a baseboard or the whole system by controlling the firmware layer. That way, the VLAN configuration and composed nodes are in the same state as before the power loss. In addition, the whole Ethernet management architecture within the RECS_Master and firmware was reworked, by building a fabric over all Ethernet switches within the RECS|Box and only presenting one virtual switch to the upper layers through the Redfish API and WebGUI.

The Redfish API was also revised by changing the internal generation of endpoint URLs. This was done with regard to an easier maintenance and testability of the implementation within the RECS_Master and a more consistent presentation to the upper layers and users of the API. Before this rework, some resources had more than one endpoint URL depending on the context it was in. One example is the high-speed/low-latency interface, which is the physical part of a microserver, PCIe device or PCIe port, where it connects to the high-speed/low-latency communication infrastructure, analogue to an Ethernet interface. Within the node composition process, those interfaces can also be the endpoints of a configured connection. Before the rework of the URL handling, such an interface could be called by two different URLs. This was now changed by assigning only one unique URL to a resource, which corresponds to its physical affiliation. The complete documentation of the Redfish API implementation of the RECS_Master can be accessed online at Github [12].

Furthermore, the whole Web-GUI of the RECS_Master was re-designed, now supporting recent Java versions. The main framework for the web interface, Vaadin[1], was also upgraded from version 7 to 17, because version 7 was end-of-life. The new Web-GUI now conforms to more modern web standards and features responsive design, for a smoother user experience (see 3.2).

4. Energy-efficient task-based runtime

This chapter describes our efforts to develop runtime technologies targeting scalability and high energy-efficiency. We present the approach taken using the XiTAO runtime first, and then the developments and evaluation using OmpSs.

[1]https://vaadin.com/

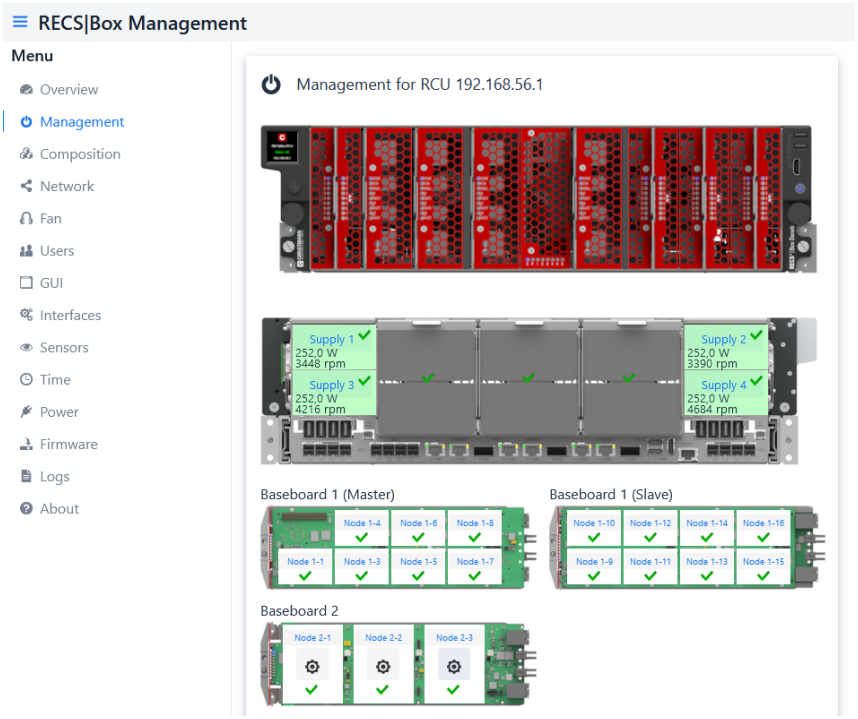


Figure 3.2. Web-GUI of the RECS_Master in new design

Table 4.1. Hardware structure of the used machines. In parenthesis: number of hardware threads that share memory/cache.

Architecture	Sockets	Cores/socket	Threads/core	Memory(GB)	L3(MB)	L2(KB)	L1d(KB)
`AMD Abu Dhabi`	4 - 2 nodes each	6	2	8(6)	6(6)	2048(2)	16(1)
`Intel Broadwell`	4	18	1	768(18)	45(18)	256(1)	32(1)
`Intel Haswell`	2	10	1	256(10)	24(10)	256(1)	32(1)

4.1. XiTAO

XiTAO is a lightweight layer that provides a task-parallel and data-parallel interface using modern C++ features. It serves as a backend runtime along side OmpSs in the LEGaTO toolchain. As mentioned in D3.3 [25], the design goals of XiTAO are to be low-overhead and to serve as a development platform for testing scheduling and resource management algorithms.

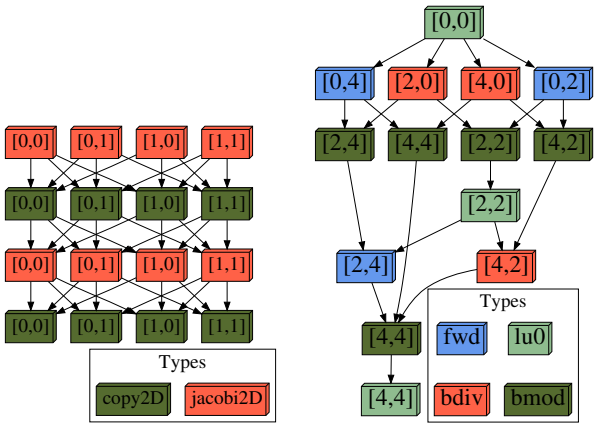
4.1.1. Evaluation of XiTAO Software Topologies

4.1.1.1. Methodology

This section describes the experimental methodology used to evaluate the contributions of of the XiTAO topologies presented in D3.3. LAGRES is integrated into XiTAO[1], a DAG runtime system implemented on top of C++11. XiTAO is designed to flexibly evaluate scheduling policies and already features moldable tasks, i.e., those that can be mapped to a single or multiple threads. This facilities the mapping of N (coarsened tasks) to M (elastic resources) [24]. However, LAGRES is decoupled from the runtime implementation.

Platforms:
Experiments are performed on three architectures: `AMD Abu Dhabi`, `Intel Broadwell` and `Intel Haswell` with the structure described in Table 4.1. The `AMD Abu Dhabi` machine has 4 AMD Opteron processors (6348), each having 2 chips, with 6 cores each, for a total of 48 cores. The

[1] https://github.com/mpericas/xitao.git



(a) 2 iterations of a Stencil 2D DAG

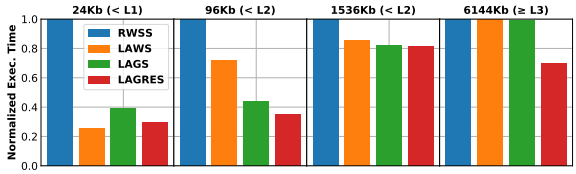
(b) SparseLU DAG of a 4x4 block matrix

Figure 4.1. Sample dependency graphs for the analyzed applications.

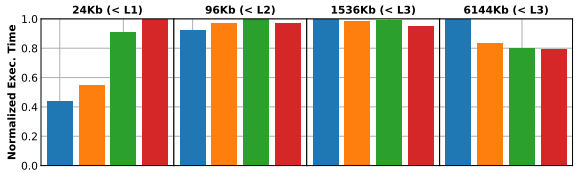
other two machines belong to academic research computing centers. One node on each machine is used exclusively (i.e. with no other jobs co-scheduled on the node during the experiments). The nodes of the `Intel Haswell` cluster have 2 Xeon E5-2650v3 processors, with 10 cores each, for a total of 20 cores. Also, we use a "large memory" `Intel Broadwell` cluster, where each node contains 4x18 Xeon E7-8860v4 cores, for a total of 72 cores.

Baseline Schedulers:
In the following, we describe and justify the baseline schedulers used.
Random Work-Stealing Scheduler (RWSS): it is based on distributed work-stealing, where each worker greedily tries to reduce idleness by fetching work from victim queues. It has been formally introduced in [8] and has appeared in several parallel task-based libraries such as Cilk [7], MassiveThreads [21] and Intel TBB [17].
Locality-Aware Work-Stealing Scheduler (LAWS): this scheduler is introduced in [11]. It is helpful in this evaluation as it is an incremental improvement over RWSS. Also, it provides work-stealing within the NUMA node of task's initialization, and tries to optimize cache misses within the node. Using ideas from LAWS, we allow work-stealing within the inclusive partitions set where the task has been initialized, and we greedily fill in the resource partitions to construct cache-friendly subtrees of the DAG.
Locality-AGnostic Scheduler (LAGS): we also evaluate the proposed scheduler without the effect of moldability. Partition widths are persistently set to 1. Tasks are initialized and executed in the NUMA node mapped by their STA. Local stealing is preferred, then global stealing requests are fulfilled when the stealing thread reaches idleness threshold and the steal that reduces the cost function (as per the model) is chosen.
LAGRES: the adaptive locality scheduler based on XiTAO topologies introduced in D3.3.

Applications:
Iterative DAG - HEAT: we leverage a DAG implementation to compute heat diffusion on a 2D grid. One of the iterative numerical methods to achieve this is to use 2D Jacobi stencil [15]. We use a 5-point stencil and create dependencies between the neighbor nodes (see Figure 4.1a). The approach involves computing the stencil in a compute task, and copying out the update in a copy task. The DAG is iteratively executed for a predefined number of iterations. For STA specification, we use the coordinates of block of mesh points involved in a task.
Recursive DAG - SparseLU: we port a SparseLU benchmark from the Barcelona OpenMP Tasks



(a) AMD - Abu Dhabi



(b) Intel - Broadwell

Figure 4.2. SparseLU parallel execution time on 8x6-core AMD - Abu Dhabi and 4x18-core Intel Broadwell as you increase the task input size in bytes. Matrix size = 128 x 128 blocks.

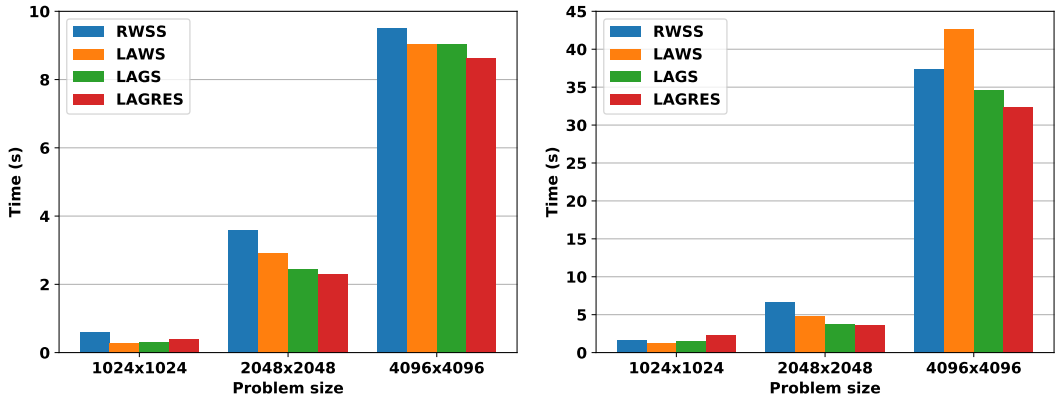
Suite [14]. This benchmark computes an LU matrix factorization over sparse matrices. The matrix is composed of NxN blocks, each of which has a pointer to a sub-matrix of size MxM. Load-imbalance is evident due to the sparsity of the matrix. For each phase of the LU algorithm, a task is spawned for non-empty blocks (see Figure 4.1b) . For STA specification, the matrix block indices are used.

4.1.1.2. Evaluation

In this section, we share the obtained empirical results comparing LAGRES to the baseline schedulers. Then, we analyze the achieved performance gains by showing the schedule map for LAGRES pertaining to a specific task type and location, and how locality is preserved adaptively. Also, we explain the gains by relating them to the proportion of the cumulative work time to the overall scheduling time.

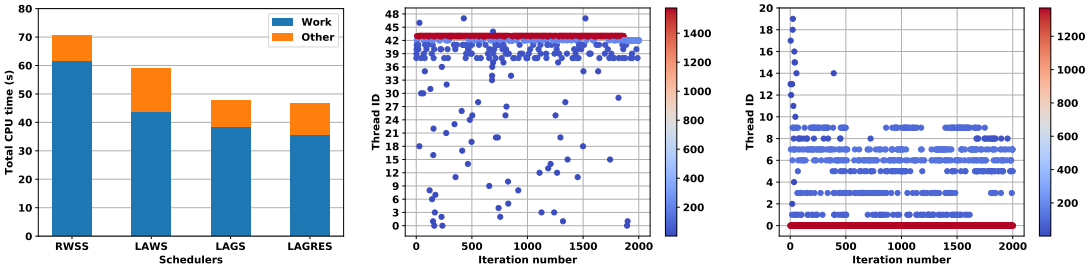
Empirical Results:

Figure 4.2 depicts the SparseLU normalized (to max) execution time of the evaluated schedulers factorizing a double-precision 512x512 matrix, block dimension size of 32, 64, 256 and 512, respectively. The block sizes are varied to span different cache-levels in the system . The experiments are run on 48 threads (`AMD Abu Dhabi`) and 72 threads - (`Intel Broadwell`). The layout description corresponds to the sharing levels of the underlying architecture, and the sharing is enabled to the socket level. In Figure 4.2a, we use $1, 2, 6, 12$, which maps to L2, L3 and socket-level sharing. With the exception of the smallest case ($< L1$), it is noted that the proposed scheduler in both its variants (LAGS and LAGRES) performs better than the baselines. Enabling elasticity provides a considerable improvement when L3 cache size is exceeded. According to our data, the adaptive socket level sharing provides higher streaming bandwidth due to interleaving over 2 NUMA nodes for the tasks that benefit from this (one with low arithmetic intensity), otherwise, the reduction in idle-time due to lower cost of molding to larger widths in the cases of tasks with higher arithmetic intensity. LAGS has an almost similar behavior to LAWS, as the model does not help much in improving over LAWS when resources are not properly utilized with moldability. The `Intel Broadwell` exhibits less cache/memory access heterogeneity due to less sharing. This limits the analysis of LAGRES to a few partitions, and yields to local/global work-stealing scheduler. Hence, the performance is bounded by LAWS with the exception of the larger cases when streaming provides an advantage. Note that the slight improvements seen for task sizes of (1536Kb and 6144Kb) are factors of 10s and 100s of seconds.



(a) `AMD Abu Dhabi` *8x6 threads* *(b)* `Intel Haswell` *2x10 threads*

Figure 4.3. Time to solution for different mesh sizes to compute 5-point stencil on two architectures. Iteration count = 2000



(a) Stencil - `Intel Haswell` *(b) Stencil (Compute)* - `AMD Abu Dhabi`- *tasks data pinned to Node 6 - leader thread = 42* *(c) Stencil (Copy)* - `Intel Haswell`- *tasks data pinned to Node 0 - leader thread = 0*

Figure 4.4. Comparison of the cumulative work/other distribution, and the scheduling trace for a Stencil (compute and copy) tasks. The denoted tasks have a single software topology address. Mesh resolution = 2048×2048.

It is also important to stress test LAGRES by conducting a scalability analysis to assess whether checking the model incurs an overhead that hinders performance gains. Figure 4.3 shows the execution time of the stencil application comparing the different schedulers as the resolution of the mesh is increased (resulting in more smaller tasks). The case of 1024×1024 yields around $128x128$ mesh points per tasks after decomposition. In essence, this is too fine grain not to observe a degradation compared to LAGS and LAWS due to the slight overhead that can diminish the benefits for smaller-sized tasks. This is also in line with the results obtained in SparseLU (Figure 4.2). The effect of resource elasticity contributed by LAGRES becomes especially clear as the problem size is increased across the 2 architectures. In the case of `AMD Abu Dhabi` depicted by Figure 4.3a, we observe improvements of at least 30% over LAWS and RWSS in the 2048×2048 case. The behavior is sustained on `Intel Haswell` shown by Figure 4.3b.

Analysis:

One of the important attributes to be analyzed in task-based runtime scheduling is the proportion of work-time, that is time spent executing the tasks, to other scheduling activity, which includes queue management, scheduling logic, dependency checks, idleness, etc. In Figure 4.4a, we show the cumulative work time, that is the thread total time x the number of threads vs. the total other time on `Intel Haswell` for the stencil application 2048×2048. The reduction in time-to-solution incurred by LAGRES mostly manifests as reduction in work time. According to our profiling data, the adaptation to locality constrains by LAWS explains the increase of "other" time. To see LAGRES in action, we display the scheduling trace for the compute and copy tasks

of the stencil. We pick a sequence of tasks dependencies that are initialized on the nodes highlighted by the Figures 4.4b and 4.4c. To reduce clutter, we omit the selected resource width and just show the leader thread for the selected resource partition. For the compute task executed on AMD Abu Dhabi, we can see that in some instances, LAGRES still chooses to schedule away from the data location, which appears to be an effective technique that reflects in reduction of time-to-solution even though data locality is not strictly preserved. However, this is not the case in the stencil copy task, where streaming bandwidth, and reduction of cache latency are crucial to achieve higher throughput. Hence, LAGRES is able to discover these attributes (i.e. the computational requirements) adaptively and without prior assumption about the nature of the workload.

4.1.2. The XiTAO heterogeneous scheduler

4.1.2.1. Energy Efficient Task Scheduler

Based on the proposal of energy efficient task scheduler runtime in D3.3, we conduct more real applications evaluation of energy savings on two different platforms. The details of platforms and applications are introduced as following:

Asymmetric platform. NVIDIA Jetson TX2 features a dual-core NVIDIA Denver 2 64-bit CPU and a quad-core ARM A57 Complex (each with 2 MB L2 cache). The board is set to MAX-N nvpmodel mode. The Denver and the A57 cores implement the ARMv8 64-bit instruction set and are cache coherent. The two clusters, i.e. Denver and A57, support the same range of frequencies and the platform only allows cluster-level frequency changes. Note that the A57 cluster cannot be powered off.

Symmetric platform. It features a dual-socket 10-core Intel 2650v3 (code-named "Haswell") node and runs Linux version 3.10.0. RAPL is used to build the power profiles and measure the energy consumption on the platform.

Image Classification (Darknet-VGG-16 CNN) is a 16-layered deep neural network. It is implemented as a fork-join DAG that spans all the layers. Each task of the DAG performs GEneral Matrix Multiplication (GEMM) operations on a sub-partition. No task criticality assignment exists in VGG-16 since all the tasks within a layer have the same properties and dependencies.

Heat Diffusion is implemented on a 2D grid by using one of the iterative numerical methods: 2D Jacobi stencil. It involves computing the stencil as compute-bound tasks, and copying out the update as memory-bound tasks. The DAG is iteratively executed for a predefined number of iterations (1000 in this work). The resolution is set to 10240 and the external DAG is constructed via slab (1D) decomposition into four domains. We also experiment with other decompositions, such as pencil (2D), without change in conclusions. The Heat DAG is symmetrical, thus no task criticality assignment is performed.

Sparse LU Factorization is selected from the Barcelona OpenMP Tasks Suite. This benchmark computes an LU matrix factorization over sparse matrices. The matrix is composed of N×N blocks, each of which has a pointer to a sub-matrix of size M×M. The application includes four kernels: LU0, FWD, BDIV and BMOD. Among of them, LU0, FWD, BDIV are set as critical tasks.

Evaluated Scheduling Policies. In order to evaluate proposed energy efficient task scheduling techniques, we utilize random work-stealing (abbr. RWS) as a baseline scheduler. RWS is a typical and widely used scheduling scheme for task-DAGs, which still works well in asymmetric environment. It is a greedy scheduler that tries to keep all cores busy. This results in more tasks

Table 4.2. Summary of evaluated runtime strategies

Name	Acronym	Notion
Random Work Stealing (with Sleep)	RWS (+Sleep)	Typical greedy scheduling (enhanced with “Sleep”)
Fastest Cores with Criticality (with Sleep)	FCC (+Sleep)	Performance-oriented criticality task scheduling (enhanced with “Sleep”)
Lowest Costs with Criticality (with Sleep)	LCC (+Sleep)	Lowest cost-oriented criticality task scheduling (enhanced with “Sleep”)
Lowest Energy without Criticality	LENC	The goal of scheduling each task targets lowest energy
Lowest Energy with Criticality	LEC	The goal of scheduling critical tasks targets performance

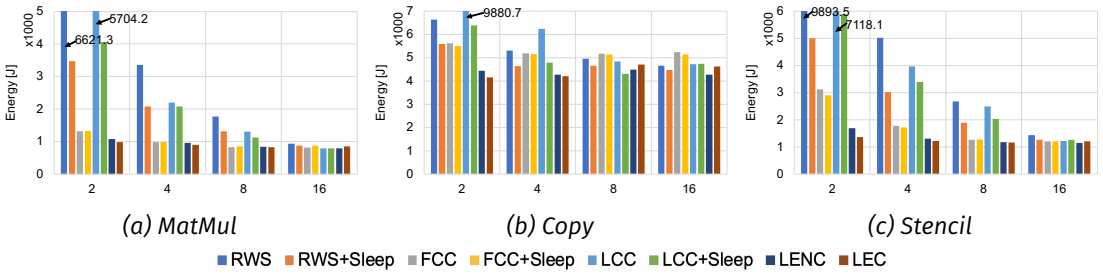


Figure 4.5. Energy comparison of three synthetic benchmarks on Intel Haswell.

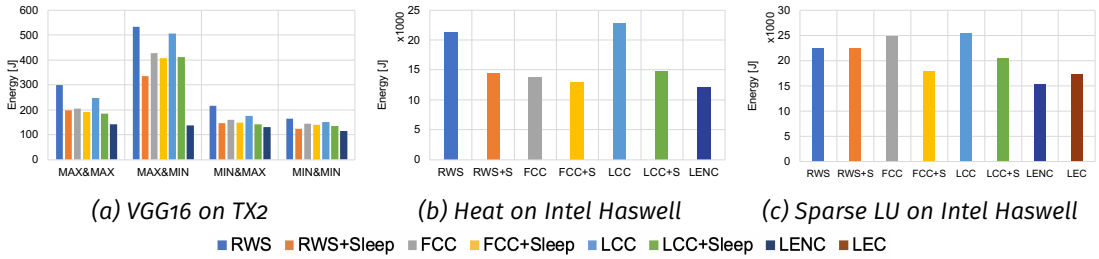
being scheduled to powerful cores, which achieves execution time balancing between asymmetric cores. Each task is executed on a single core in the RWS baseline.

In addition, we compare with a performance-oriented criticality aware scheduler (abbr. FCC). FCC is inspired by related work criticality aware task scheduler CATS [13], but instead of using one core for each task in CATS, FCC features task moldability, i.e. each task can be executed on multiple cores, which could show the benefit of task moldability by comparing to RWS. In comparison to RWS, FCC has criticality awareness and critical tasks in FCC are dynamically scheduled to the fastest configuration pair (leader core, resource width) by globally checking all possible configurations in the dynamic performance modeling table described in D3.3, while non-critical tasks only search locally for the fastest resource width (i.e. task moldability) without changing the launching core. We also develop a lowest cost-oriented criticality aware scheduler (abbr. LCC). The difference between FCC and LCC is that LCC attempts to minimize the parallel costs, i.e. the product: execution time $\times$ the resource width. We also enhance these three schedulers with the same exponential backoff sleep strategy to study this energy conservation technique.

The energy efficient scheduler (LENC) is a variant that steers all tasks for the lowest energy, independent of task criticalities. Finally, to learn the impact of critical tasks on energy efficiency, we devise a lowest energy with criticality scheduler (LEC) which assigns critical tasks to the fastest configuration pair, similar to FCC. All evaluated schedulers and their variants are described in Table 4.2.

4.1.2.2. Evaluation Results

Figure 4.5 shows energy consumption of different scheduling policies when running three synthetic benchmarks with different DAG parallelisms ranging from 2 to 16. Note that we evaluate a wider DAG parallelism spectrum (i.e. we include p=16) since the symmetric platform comprises more cores. The results show that LENC consumes less energy than other scheduling policies and the energy reduction is significant especially when DAG parallelism is low. In the case of MatMul with low DAG parallelism, RWS only utilizes a single core for executing a task, while under-utilized cores consistently attempt work stealing resulting in a considerable increase in



(a) VGG16 on TX2 *(b) Heat on Intel Haswell* *(c) Sparse LU on Intel Haswell*

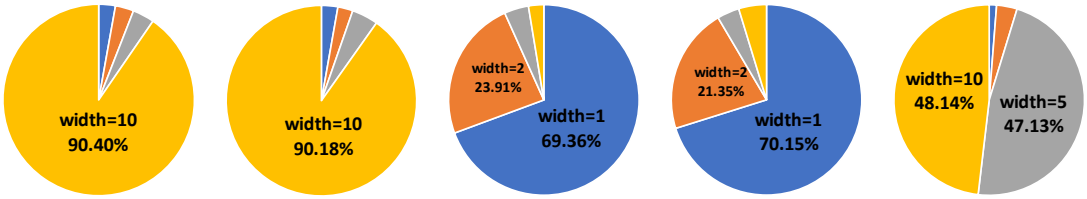
Figure 4.6. Energy comparison results of three applications on two platforms. MAX & MIN in x-axis of Figure 4.6a means that on TX2 Denver cluster's frequency is maximum and A57 cluster's frequency is minimum.

energy consumption. On the contrary, LENC is able to achieve high energy efficiency through moldable execution of tasks. Wider width selection aids in effectively reducing the energy consumption by limiting periods of under-utilization. With FCC, 95.5% tasks execute with a resource width of 10 since it is the fastest configuration and 1.6% execute with a resource width of 5. In comparison, LENC executes 96.1% tasks with a resource width of 10 and 3% tasks with a resource width of 5, which further reduces under-utilization. The gap between LENC and FCC reduces with increasing DAG parallelism since there is a decline in the under-utilization period.

In this section, we analyze the impact of the scheduling features using three real applications. VGG-16 is an inference workload that is typical of mobile and edge devices. We use the Nvidia TX2 platform as it features characteristics typical from this domain (asymmetry, low core counts). The Heat and Sparse LU benchmarks are common HPC workloads that we evaluate on the dual-socket Intel multicore node (symmetric, high core count). We also evaluated the opposite configurations which leads to the same conclusions and is not further discussed here.

Asymmetric Platform. Figure 4.6a shows energy consumption when running VGG-16 application on TX2. The results show that the LENC scheduler is the most energy efficient scheduler across different frequency settings. This scheduler achieves 31% to 75% energy reduction in comparison to RWS, and from 19% to 68% energy reduction compared to FCC and between 25% to 73% energy reduction compared to LCC. This can be attributed to the benefits by using task-type aware execution and task moldability of LENC when running on asymmetric platforms.

Symmetric Platform. In order to show the genericity and portability of our proposal, we also evaluate two applications i.e. Heat and Sparse LU, on one symmetric platform Intel Haswell. Figure 4.6b and 4.6c show the energy consumption comparison between different scheduling strategies. The results show again that LENC achieves better results than other scheduling policies. To understand the effectiveness of LENC, we analyze the resource width distribution across different scheduling policies for Heat. The result is shown in Figure 4.7. The figure shows that the majority of tasks in FCC execute with a resource width of 10. In contrast, in the case of LCC 70% of tasks execute with a resource width of 1 due to reduced parallel cost associated with the execution. In the case of LENC, the number of tasks that execute with a resource width of 5 and 10 are almost equal. This happens because in Heat there are two different types of kernels with different scalability: *jacobi* and *copy*. The *jacobi* kernel is compute-bound and while *copy* is memory-bound. LENC is able to determine an optimal width for each kernel. During the run, 98% of *jacobi* tasks execute with a resource width of 10 as it the most energy efficient configuration while 97% of the *copy* tasks execute with a resource width of 5.



(a) FCC *(b) FCC+Sleep* *(c) LCC* *(d) LCC+Sleep* *(e) LENC*

Figure 4.7. Heat task distribution with different resource width on Intel Haswell.

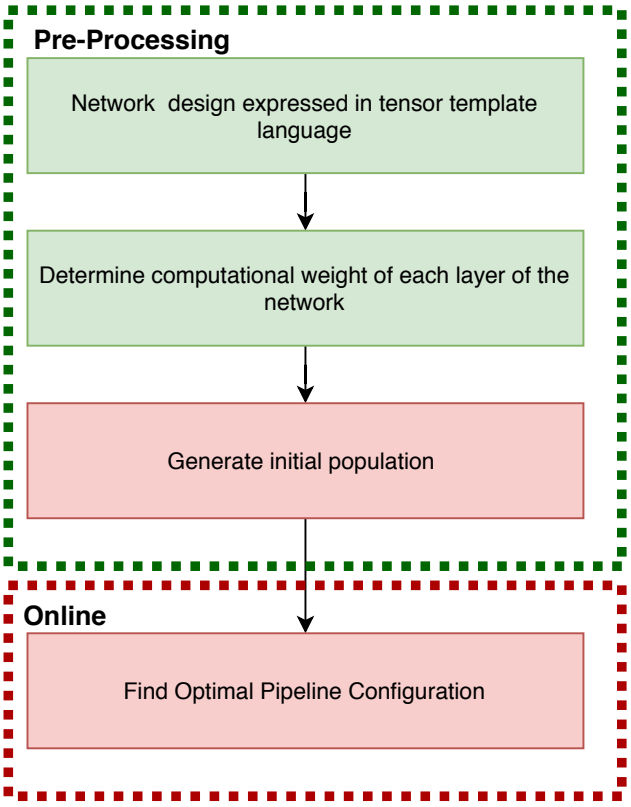


Figure 4.8. Overview of Pipelined CNN in XiTAO.

4.1.3. Support for Pipeline Parallelism

XiTAO now supports pipeline parallel execution of Convolutions neural networks. The system overview is depicted in figure 4.8. The Framework consists of 2 modules.

1. In Pre-processing module, We determine the computational hints from network descriptor. The hints provide a notion of computational weights of each layer based on which we model the initial partition of network layers to generate a pipeline stage.
2. In the online module, we measure the configurations which are highly expected to be a good candidate for a balanced pipeline on a given platform. The training finishes when algorithm has found the optimal solution for mapping. Rest of the input data is then processed in pipelined fashion.

4.1.4. Pipelined execution of VGG-16

The figure 4.9 shows two phases of execution:

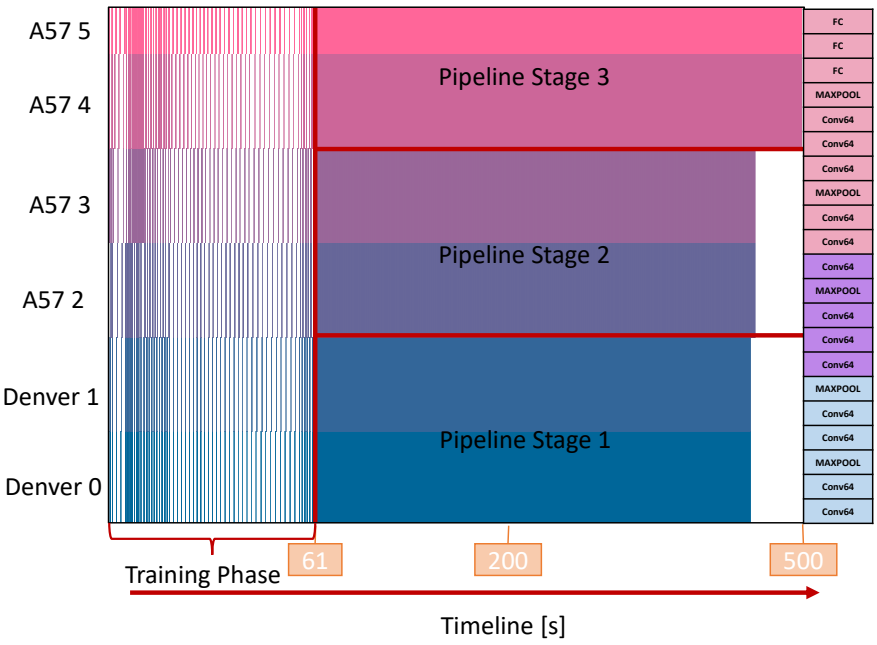


Figure 4.9. Execution timeline of VGG 16 on Nvidia Jetson TX2.

1. The training phase of the execution.
2. The normal phase of the pipeline processing.

During the training, after 61 input frames, the algorithm converged to a 3-Staged pipeline. The core partitioning is static and pre-decided for each pipeline stage count. Pipeline Stage 1 comprises of first 6 layers and executes on 2 Denver cores. Pipeline Stage 2 comprises of next 5 layers and executes on 2 A57 cores. Pipeline Stage 3 comprises of the last 10 layers and executes on 2 A57 cores.

As throughput maximizing pipeline is the one which has smallest bubble size and the slowest stage takes minimum possible time.The bubble size is referred to as the latency gap between the execution time of pipeline stages, zero bubble size ideally means a fully balanced pipeline. Thus the configuration is chosen based on the fact that it minimizes the bottleneck of the pipeline. As we can also observe in the figure 4.9, the resulting pipeline has a negligibly small bubble size.

4.2. OmpSs

4.2.1. OmpSs@FPGA, OpenCL and CUDA

In this section we show the evaluation of the OmpSs support for Intel FPGAs (including environments with FPGAs and Nvidia GPUs), the evaluation of OmpSs@FPGA on Xilinx FPGAs, using the ZCU102 development kit, and the development and evaluation of the OmpSs-2 support for Xilinx FPGAs.

4.2.1.1. OmpSs support for CUDA and OpenCL

We have evaluated the OmpSs support for OpenCL FPGAs and Nvidia GPUs, with two different environments:

- Intel Arria 10 FPGA: single chip 4 core Intel Core(TM) i7-6700 @ 4GHz, with hyperthreading, Terasic de5net a7 Arria 10 FPGA, and Nvidia GeForce GTX Titan X GPU. Matrix multiplication

is executed with a matrix size of 2048x2048 single precision values, and a block size of 64x64 elements.

- Intel Stratix 10 FPGA: dual chip 6 core Intel Xeon(R) Bronze 3204 @ 1.9GHz, without hyperthreading, Intel Stratix 10 FPGA, and Nvidia GeForce RTX 2070 SUPER GPU. Matrix multiplication is executed with a matrix size of 4096x4096 single precision values, and a block size of 128x128 elements.

Figure 4.10 shows the performance obtained on the Matrix Multiplication benchmark using the combined OmpSs@OpenCL, to execute work on the Arria 10 FPGA, and OmpSs@CUDA, to execute work on the Nvidia GeForce Titan X GPU. The benchmark uses the implements approach to indicate that the block by block matrix multiplication kernel on block sizes of 64x64 elements is available in three versions: for the Intel cores, using the MKL library, for the Arria 10 FPGA, compiled from OpenCL, and for the Nvidia GPU provide in CUDA or using the CUBLAS library.

Results show that OmpSs running on the FPGA only (labeled OmpSs@FPGA) introduces some overhead compared to the native OpenCL execution (1st column, labeled opencl-fpga).

In this particular case, the FPGA performance is comparable to the performance of a single Core i7 (labeled smp 1). The SMP cores scale up to 4, reaching around 350 Gflop/s. Hyperthreads (labels smp 5 to smp 8) do not add additional performance due to the sharing of the floating point units across them. This is a well-known behaviour for hyperthreading and HPC applications.

Nevertheless, hyperthreading is actually providing additional performance when used in combination with the FPGA. In this case, every core included in the execution, adds 50 Gflop/s (labels smp 1+fpga to smp 4+fpga) and every additional hyperthread adds around 20 Gflop/s (smp 5+fpga to smp 7+fpga). The particular case of smp 8+fpga gets lower performance as in this case we need to have one of the hyperthreads running matrix multiplication kernels in the SMP, and also taking care of the events of the FPGA through the OpenCL runtime. This sharing is introducing delays in the synchronization and this lowers the performance obtained.

The performance is compared also with the CUDA versions (labeled cuda-gpu and cublas-gpu). They obtain around 150 Gflops, and when adding additional cores (labels smp 1+gpu to smp 3+gpu) we achieve an increase in performance. When using 4 additional cores, the performance is getting a lot of variability due to the use of one additional hyperthread to manage the GPU. This seems to indicate that the management of the GPU takes more resources than the management of the OpenCL FPGA, if we compare columns smp4+gpu with smp 4+fpga.

Hyperthreading is not helping in any of the cases, labels smp 5+gpu to smp 8+gpu. Finally, label OmpSs@CUDA/FPGA shows the performance obtained when using the CUDA GPU and the OpenCL FPGA at the same time. In this case, we do not use additional SMP cores, as they only reduce the performance obtained. As there is no device that outperforms the rest, in this case the data transfers between the GPU/FPGA and main memory hinder the possibility to obtain performance when using both devices.

Figure 4.11 shows the performance obtained when running the same benchmark on the Stratix 10 FPGA and GeForce RTX 2070 SUPER GPU.

Comparing the performance of OmpSs with the FPGA (labeled OmpSs@FPGA), with the performance of the native OpenCL execution, we also observe a light drop in performance due to the overhead of OmpSs task management.

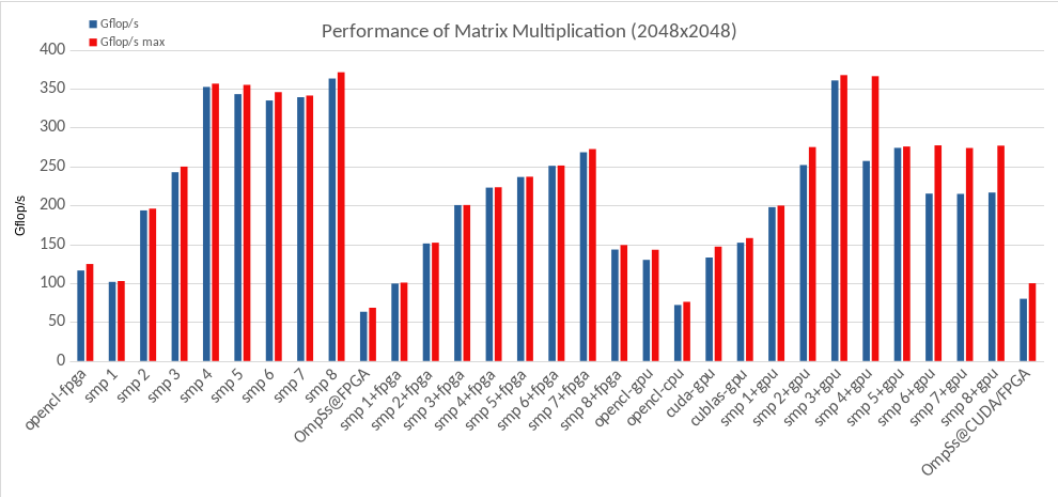


Figure 4.10. Performance of OmpSs@OpenCL/CUDA Matrix Multiplication on the Intel Arria 10 FPGA and the Nvidia GeForce GTX Titan X GPU

In this case, the SMP cores (with no hypertreading), labels smp 1 to smp 12, scale from around 40 Gflop/s to 420 Gflop/s when using 11 cores. The single core performance is much lower than the Intel Core i7, but the Xeon architecture provides better memory bandwidth, and scalability. This also allows that 11 cores add performance on top of the FPGA (labels smp 1+fpga to smp 12+fpga). The later smp 12+fgpa is not scaling properly due to the sharing of one core between the SMP tasks execution and the FPGA management.

On this environment, the Nvidia GPU is much more powerful that the rest of the devices (label cuda-gpu, reaching 800 Gflop/s). This prevents getting additional performance when adding cores to the experiments (label smp 1+gpu to smp 12+gpu). In these cases, the GPU is so fast that allowing a core getting a task for execution already slows down the execution of the benchmark.

When using the combined execution on GPU and FPGA, the performance is also reduced compared to GPU only, as the effect is similar. The task executing on the FPGA delays the rest of the application. On this environment, we also show how the cores contribute when using the GPU and the FPGA. Initially the performance drops a lot (smp 1+gpu+fpga), increasing slightly with additional cores (smp 2+gpu+fpga to smp 5+gpu+fpga), and getting a little lower and stable up to smp 10+gpu+fpga. The executions on smp 11+gpu+fpga and smp 12+gpu+fpga get lower performance due to the overhead introduced by sharing two cores with SMP execution and GPU and FPGA management.

In conclusion, we think that the implements approach can be interesting when the performance of the several resources used is not so different, like what happens with the FPGA and the SMP cores, and in those situations adding several additional cores to the execution can improve the performance (like in the experiments labeled smp 1+fpga to smp 11+fpga).

4.2.1.2. OmpSs@FPGA energy efficiency

We have executed a set of matrix multiplication experiments in the Xilinx ZCU102 development kit, with 4 ARM A53 cores and an integrated XCZU9EG FPGA. Each matrix multiplication experiment consists of 20 executions of matrix multiply, on a matrix of 2816x2816 single precision floating point elements. The matrix is blocked in tiles of 256x256 values.

The program uses OmpSs to offload tasks, either to the FPGA or the ARM cores. The FPGA is programmed through OmpSs to have 3 instances of the matrix multiplication IP core, that can

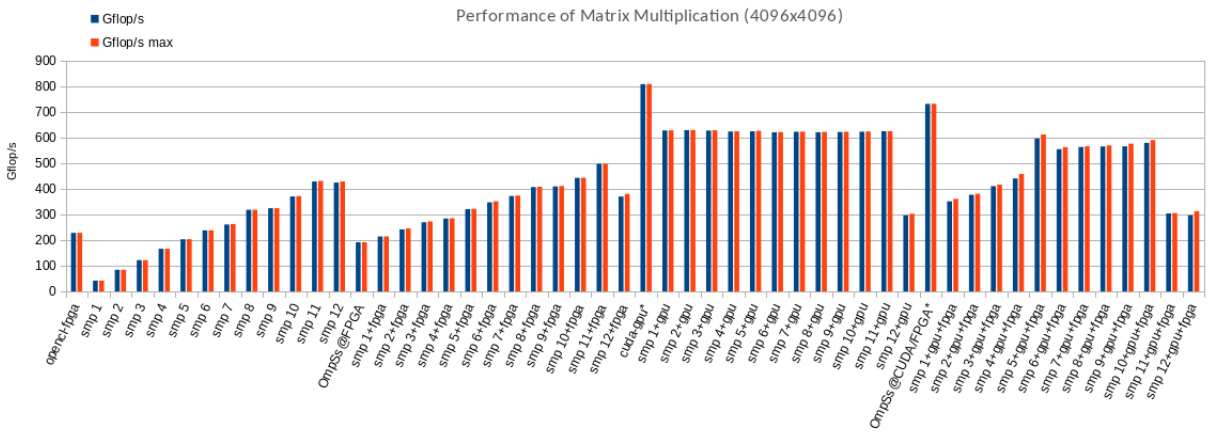


Figure 4.11. Performance of OmpSs@OpenCL/CUDA Matrix Multiplication on the Intel Stratix 10 FPGA and the Nvidia GeForce RTX 2070 SUPER GPU

be used in parallel, and independently of each other.

Figure 4.12 shows the power consumption of the execution of 7 experiments, in this order:

1. the two first peaks reaching up to 18W (between seconds 17-33 and 49-65, are two executions of the 20 matrix multiplications, using the 3 instances of the matrix multiplication IP core.
2. the next two peaks reaching 14W (between seconds 75-97 and 110-129) are two executions of the 20 matrix multiplications, using 2 of the IP core instances.
3. the next two sets of bars reaching 10.5 - 11W (seconds 141-200 approximately) are the same two executions of the 20 matrix multiplications, using a single of the IP core instances.
4. the final flat line stable at 3W is the same execution of the 20 matrix multiplications using the 4 ARM A53 cores only. In this last execution, we have substracted the power consumption of the PL logic (around 3.8W while idle), as it is not used in the execution.

Figure 4.12 shows the total power consumption (light blue line), the Processing System (PS) power (dark blue), and the Processing Logic (PL) power (yellow), as the main contributors. DDR, BRAM, and Management power consumptions are constantly below 0.5W.

If we compute the GFlops obtained by the different sets of executions, we get:

- 3x IP core instances achieve 94.5 Gflops, with a mean consumption of 15.1W and a maximum of 6.26 GFlops/W.
- 2x IP core instances achieve 77.6 Gflops, with a mean consumption of 10.6W and 7.32 GFlops/W. This is the best energy-efficient experiment.
- 1 IP core instance achieves 47.1 Gflops, with a mean consumption of 9.2W and 5.11 GFlops/W.
- 4 ARM A53 cores achieve 12.1 Gflops, with a mean consumption of 2.95W, and 4.1 GFlops/W.

Performance (GFlops) and energy efficiency (GFlops/W) for the same experiments is shown in Figure 4.13. Results show that the best performance is obtained with 3 matrix multiply IP accelerators, while the best energy efficiency is obtained with 2 matrix multiply IP accelerators.

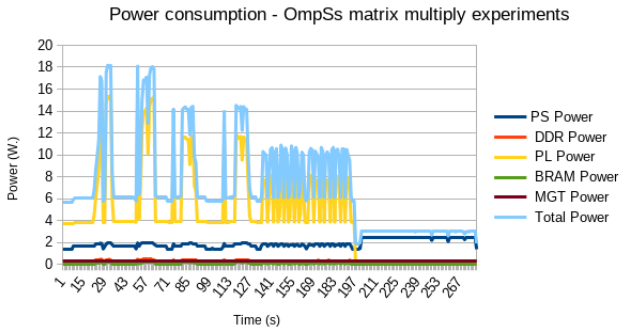


Figure 4.12. Online power consumption on the matmul experiments with OmpSs@FPGA

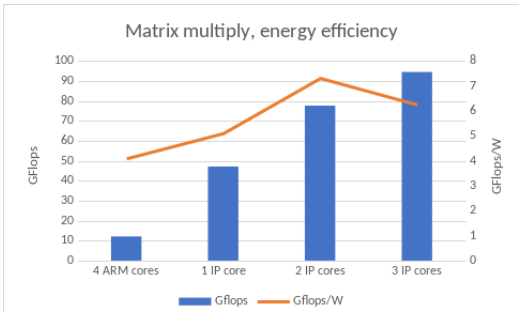


Figure 4.13. Performance and energy-efficiency with OmpSs@FPGA matrix multiplication

Concluding, the code transformations done with OmpSs to exploit matrix multiplication on the FPGA result in an increase of the GFlops/W obtained from the Zynq U+ chip, increasing power efficiency when using more resources in the FPGA fabric.

4.2.1.3. OmpSs-2 support for FPGA/CUDA separate memories

While developing the new OmpSs-2 runtime system, we have had the need to incorporate support for GPUs and FPGAs with separate global memory. This environment covers discrete GPUs with or without unified memory support and discrete FPGAs. We have designed the directory and software caching mechanism for OmpSs-2 to support such devices. This work has been done in the context of a master thesis [28]. From the compiler, at task creation point the nanos6 runtime receives the information on the task dependences, that can be directly used to implement the data caching mechanism on the target device memory, and issue the memory transfers. Those memory transfers are implemented by means of the CUDA runtime, for Nvidia GPUs, and using our xtasks/xdma [9] libraries for Xilinx FPGAs. We have leveraged the same code generation that we use for OmpSs-1 in OmpSs-2, so the infrastructure that we have in the FPGA is the same, and it changes the software side that we have made integrated with the OmpSs-2 runtime system.

Figure 4.14 shows the comparison of the performance obtained on the matrix multiplication benchmark on the ZCU102 FPGA environment, when using different matrix sizes and the OmpSs-1 and OmpSs-2 environments. As it can be seen the performance is very similar, with a small performance drop on the new OmpSs-2 environment, that we will examine and try to further reduce in the future. Figure 4.15 shows the comparison of the performance obtained in the OmpSs-2 environment, with the new OmpSs-2 directory/cache support, OmpSs-2 with Prefetching and OmpSs-2 with Unified Memory. Results demonstrate that the new implementation of OmpSs-2 with the directory/cache outperforms both the prefetching and the unified memory support. This fact makes us think that the implementation done with the directory and cache

support will be useful for future developments with OmpSs-2.

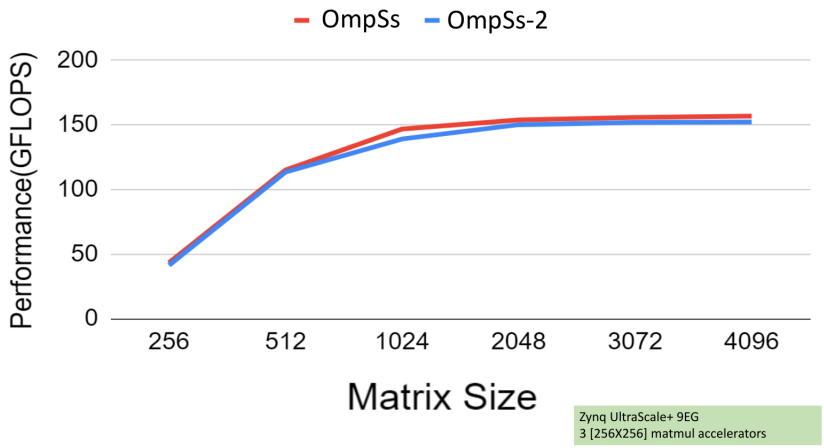


Figure 4.14. Comparison of the performance of OmpSs-1 and OmpSs-2 matrix multiplication on the ZCU102 FPGA development kit, with the new OmpSs-2 directory/cache support

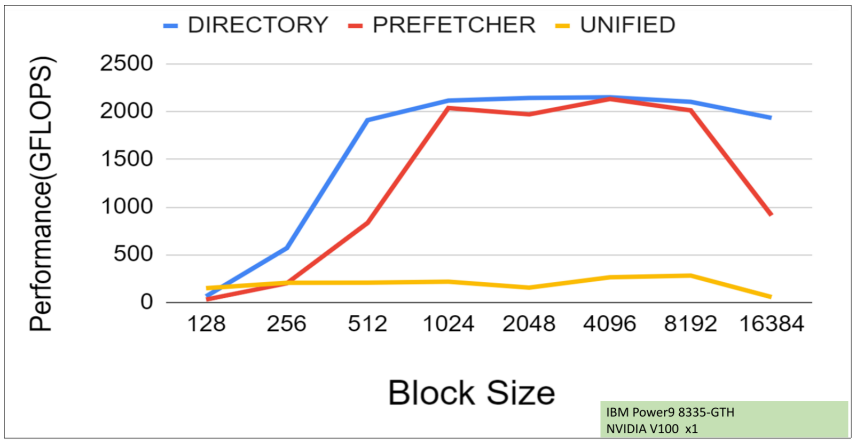


Figure 4.15. Comparison of the performance of OmpSs-2 matrix multiplication, with the new OmpSs-2 directory/cache support, OmpSs-2 with Prefetching and OmpSs-2 with Unified Memory

4.2.2. OmpSs@Cluster

OmpSs-2@cluster is the distributed memory variant of the OmpSs-2 programming model. It is based on the latest iteration of the OmpSs programming model, *i.e.*, OmpSs-2 [5], which supports efficient task nesting through weak dependencies and early release of dependencies [23]. OmpSs-2@cluster was originally developed in ExaNoDe [27], and it is compatible with the SMP version of OmpSs-2, so that the same program and compiled binary can be executed on either an SMP or a cluster of SMPs. OmpSs@cluster has been part of the OmpSs-2 public release since June 2019.

4.2.3. Smart-Mirror/OmpSs@Cluster

The Smart Mirror use case is an interactive human interface to a smart environment, such as a home, where sensory information is collected from the surroundings to give the user helpful information, such as the weather forecast or temperature, in addition to helping them by monitoring, remotely controlling or automating their daily actions (*e.g.*, opening a window or setting a schedule) [30].

The Smart-Mirror software comprises an object recognition module that employs a neural network to identify objects and faces seen by the mirror, in addition to voice and gesture con-

trol modules. The hardware prototype and setup employs two NVIDIA Jetson Xavier developer boards.

The object detection module includes a camera part by which an image is captured (in raw format) from a video stream and processed (*e.g.*, scaled) to match the requirements of the neural network that performs the actual object detection and extraction. Finally, a Kalman and Hungarian filter step is introduced which increases the certainties of current detection by incorporating previously detected objects along with the new ones.

OmpSs-2@cluster allows the CPU and GPU compute to be divided among two nodes, increasing the frame rate by a factor of three from 6 fps to 19 fps at a 17% increase in power consumption, from 47 W to 55 W. This is possible using the annotations described in this section at minimal development cost.

4.2.3.1. Single Node Implementation

The original implementation run on a single Xavier board, leaving the second node non-utilized. The single node implementation also executed the camera, neural network, and Kalman filter parts sequentially, hence not fully utilizing all the cores on the single Xavier node.

Figure 4.16 highlights a snippet code of the original implementation. The neural network is initialized (line 2) according to some desired configurations and weights. The application goes into the main loop (line 11), which runs until the program quits. At each iteration, it captures an image (line 14) from the camera, feeds this image to the network (lines 18–19), and finally applies the Kalman filter (line 23).

4.2.3.2. OmpSs-2@Cluster Implementation

The main objective was to divide the work between the two Xavier nodes such that the first node will be used for the camera capturing part alongside other modules, while the second node accelerates the object detection and the filter parts. It was also necessary to do so in a portable and productive way, including the minimum of details about the platform and no explicit data transfers. To fully utilize both Xavier nodes, the Smart-Mirror application (specifically the object detection, and the camera parts) was ported to OmpSs-2@Cluster task-based programming model, and executed by the Nanos6 runtime system that implements the OmpSs-2@Cluster model.

Porting Smart-Mirror to OmpSs-2@Cluster required a straightforward and clear transformation by dividing the code into tasks. The *taskification* was done by surrounding the single node code with directives (`#pragmas`). To express data dependencies between different tasks, extra clauses `in`, `out`, and `inout` were appended to the `#pragmas` to specify all memory accesses of these data. Tasks are then executed in parallel by the Nanos6 runtime, and easily scheduled by the runtime to run on any desired Xavier node. Scheduling tasks to run on a specific node however can be forced by using the node clause.

Figure 4.17 and Figure 4.18 shows the OmpSs-2@Cluster taskified code for the same code in Figure 4.16. Task `init_node_one` represents the neural network initialization. As mentioned above, since the prediction step will be entirely computed on the second Xavier node, we use the node clause here with value of 1 indicating that we want to schedule this task on the second node (counting from 0). The node clause was introduced in D3.3 [25].

As mentioned earlier, at each iteration we capture an image frame, perform predictions, then

```
// Initialize the  neural network with the given weights
network = load_network_custom(CFG_FILE, WEIGHT_FILE,1,1);
set_batch_network(network, 1);
fuse_conv_batchnorm(*network);
calculate_binary_weights(*network);
...
// Initialize Kalman and Hungarian filter
init_trackers(classes)
...
// Main loop
While(1)
{
  // Capture raw image from the camera and scale it.
  inS = get_image_from_stream_resize(cap, IMAGE_WIDTH, IMAGE_HEIGHT,
                                     CHANNELS, &pInImg, 1);
  ...
  // Feed the image into the netwrok and perform prediction.
  network_predict_image(net, det_s);
  pDetections = get_network_boxes(network, IMAGE_WIDTH,IMAGE_HEIGHT,
                                  THRESH, HIER_THRESH, 0, 1, &nBoxes,0);
  ...
  // Update the tracked detections using the Kalaman filter.
  updateTrackers(*pDetections, nBoxesTaskThreshed, THRESH, &pTrackedDets,
                 &trackedNBoxes, IMAGE_WIDTH, IMAGE_HEIGHT);
}
```

Figure 4.16. Single node implementation of the object detection part of the smart mirror.

```
#pragma oss task out(nBoxesLastLoop, pTrackerIDLastLoop, pObjectTypLastLoop,    \
        pBBoxLastLoop, pDets0, pDets1, nBoxesTask0, nBoxesTask1,                \
        pNetwork, pIn1S1,pIn0S1) in(classes) node(1) label("init_node_one")
{
...
    pNetwork = load_network_custom(CFG_FILE, WEIGHT_FILE, 1, 1);
    set_batch_network(pNetwork, 1);
    fuse_conv_batchnorm(*pNetwork);
    calculate_binary_weights(*pNetwork);
    init_trackers(classes);
...
}
```

Figure 4.17. Initialization step of the OmpSs-2@Cluster implementation of of the object detection part of the smart mirror.

update the tracked features (trackers) resulted from the predictions using the Kalman filter. Since the image frame capturing code is executed entirely on the first node and also it will be the only code that will be executed on the first node, so it is not taskified and is executed as a normal function call, `get_image_from_stream_resize`. Note that the captured image on node 1 needs to be copied to node 2 for the predictions, however OmpSs-2@Cluster simplifies this by only adding a pointer to the image data as an input dependency to the predict task,

```
While(1)
{
    if(itr % 2 == 0) //even iteration
    {
        inS0 = get_image_from_stream_resize(cap, IMAGE_WIDTH, IMAGE_HEIGHT, CHANNELS,
                                            &pInImg, 1);
        ...
        #pragma oss task in(pData1[0; inSDataLength]) node(1) label("even_buffer")
        {
            for (size_t i = 0; i < inSDataLength; ++i)
                pIn1S1->data[i] = pData1[i];
        }
        ...
        #pragma oss task in(pIn0S1) node(1) label("even_predict")
        {
            ...
            network_predict_image(pNetwok, *pIn0S1);
            *pDets0 = get_network_boxes(pNetwok, IMAGE_WIDTH, IMAGE_HEIGHT, THRESH,
                                        HIER_THRESH, 0, 1, nBoxesTask0, 0);
            ...
        }
        ...
        #pragma oss task out(nBoxes) out(pBBox[0;4*defaultDetectionsCount]) \
                        out(pTrackerID[0;defaultDetectionsCount])          \
                        out(pObjectTyp[0;defaultDetectionsCount])          \
                        out(changed) node(1) label("even_update")
        {
            ....
            updateTrackers(*pDets1, *nBoxesTask1, THRESH, &pTrackedDets, &trackedNBoxes,
                           IMAGE_WIDTH, IMAGE_HEIGHT);
            ...
        }
    } else // odd iteration
    {
        // same as the even iteration, howver using pData0 buffer
    }
```

Figure 4.18. Main loop of the OmpSs-2@Cluster implementation of the object detection part of the smart mirror.

hence data copy will be made implicitly by the Nanos6 runtime.

To add more parallelism, we needed to execute the prediction step concurrently with the update step. However the update cannot occur until after the prediction has been made. Hence we use two predictions buffers `pDets0`, and `pDets1`, iteration `(i+1)` will be predicting the current iteration `(i)` image frame, and updating the the predictions of the previous iteration `(i-1)`. To avoid copying the data between the two buffers at each iteration, we divide the while loop into even and odd iterations, and swap the pointer passed to each of the prediction, and update tasks.

Note that the `node(1)` clause is also used with the `even_buffer`, `even_predict`, and `even_update` tasks to schedule them on the second Xavier node as desired. In addition, transferring the image is implied via the dependency representation clause.

	Single Node		OmpSs-2@Cluster	
	Xavier node 1	*Xavier node 2*	*Xavier node 1*	*Xavier node 2*
CPU utilization	30%	0%	116%	122%
GPU utilization	80%	0%	0%	80%

Table 4.3. CPU and GPU utilization of the single node and the OmpSs-2@Cluster implementations running alone.

	Frame rate	**Power consumption**
Single node	6 fps	47 W
OmpSs@cluster two node	19 fps	55 W

Table 4.4. Frame rate and power consumption of whole smart mirror application.

4.2.3.3. Results

To fully utilizing both Xavier nodes and shifting the GPU usage from the first Xavier node entirely to the second Xavier node, Table 4.3 shows the CPU and GPU utilization of the single node implementation versus the OmpSs-2@Cluster.

Table 4.3 shows the performance of the object recognition module running alone. Both the single and OmpSs-2@Cluster implementations are limited by the camera's frame rate to 30 FPS. The energy consumption is 25 W for the single node implementation, whereas for OmpSs-2@Cluster this dropped to 13 W for the first node due to 10 W consumption for the per-processing of the raw captured image (*i.e.*, scaling), and total of 17 W for the second Xavier node.

Table 4.4 shows the frame rate and power consumption for the whole full smart mirror application. In both versions, single node and OmpSs-2@cluster, the bottleneck is the GPU on the first node. Offloading the object and gesture recognition from the first to second node allows the GPU compute to be divided among two nodes, increasing the frame rate by a factor of three from 6 fps to 19 fps at a 17% increase in power consumption, from 47 W to 55 W. This was possible using the annotations described in this section at minimal development cost.

5. Runtime support for Fault Tolerance and Security

5.1. FPGA Checkpointing

Current supercomputers are in need of efficient fault tolerance systems due to their sheer amount of computing units; FPGAs also need to co-exist with them and be able to take advantage of those resilience techniques as well. Thus, evaluating the performance of fault tolerance schemes in HPC systems with FPGA devices is crucial for the future of large scale supercomputers.

Checkpointing is the preferred software resilience mechanism for Scientific Computing which typically feature long running applications. In Legato, we propose to implement Checkpoint-Restart (CR) for FPGA applications running on heterogeneous systems (i.e., CPU-FPGA). For that, we adapt our multilevel checkpoint library called Fault Tolerance Interface (FTI) [6], in order to be able to checkpoint FPGA applications. This is the same library we used in the past to checkpoint GPUs (See Deliverable 3.3). With this, we achieve a common resilience interface, to provide fault tolerance to all type of heterogeneous systems with power efficient accelerators (*i.e.*, GPUs and

FPGAs).

In this section we explain the methodology which we followed to implement FPGA checkpointing for applications that run with OmpSs@FPGA. The work is separated into two different main approaches when it comes to implementing FTI functionality: host-based checkpointing and checkpointing of partial data from the FPGA task. We use the term *FPGA task* to describe the portion of the application which is implemented for FPGA.

5.1.1. Host-based Checkpointing

As the name implies, this implementation approach is completely transparent to the FPGA, and is only done from the host application, once the FPGA task has finished. It is the most typical implementation, where the FTI annotations are usually put in the main loop and keeps track of the iteration number and the data needed to recover the execution upon a failure. This is the ideal method when the FPGA tasks do not take a considerable amount of time to execute. This implementation is also most useful when the FPGA tasks are repeated a considerable amount of times, in which case it gives the user enough flexibility for setting the checkpoint frequencies.

5.1.1.1. Adapt Applications to MPI/FPGA

The first step is to adapt the applications that will be used for the evaluation to the proper environment. That is, to adapt the message passing programming model of the host application and transform the work of the inner kernels of the application into FPGA tasks. FPGA's configurable logic.

We work mainly with three different applications: *(1)* a Jacobi solver, *(2)* a YUV filter and *(3)* a K-means application (See Section 5.1.4). In the case of the Jacobi solver, MPI communications were already implemented in the original code (`heatdis.c`) [1] and thus we only needed to transform the computation part to FPGA tasking. The YUV filter was already implemented as an FPGA task but was not implemented as a distributed application, which we did. This algorithm was also slightly modified to perform various filters for the same image repeatedly, instead of doing it for different instances of the images. The reason of this change was to avoid the hosts having to deal with thousands of instances of the same image, since that would saturate their limited memory capacity. The K-means application was already implemented as a MPI/FPGA applications, and so it did not need any significant change.

5.1.1.2. Add FTI support

The change mainly consisted in protecting the variables necessary for the recovery of the execution in case of failure, and adding the FTI annotations inside the main loop to decide when it is time to perform a checkpoint or recovery. Fig. 5.1 shows an example of this checkpointing method. The important data is protected by using `FTI_Protect`, which will be recovered in the case of a failed execution. Note that in addition to the main datasets, we also need to protect the iteration number since it is crucial to properly resume the execution.

In the case of the Jacobi solver and YUV filter, `FTI_Snapshot` was used to signal the application to checkpoint at a given interval using the configuration file. For the K-means application, however, a manual checkpoint was performed instead for more than one minute.

```
   // Read input image
   image_read(img_rgb, myid, numprocs);
#ifdef __FTI__
   FTI_Protect(0, &i, 1, FTI_INTG);
   FTI_Protect(1, img_rgb, sizeof(image_t), FTI_CHAR);
#endif
   wtime = MPI_Wtime();
   for (i=0;i<n_images; i++){
#ifdef __FTI__
      int checkpointed = FTI_Snapshot();
#endif
      yuv_filter(img_rgb, img_restore, 128, 128, 128);
      #pragma omp taskwait
```

Figure 5.1. Example of host-based checkpointing

5.1.1.3. Host-based Checkpointing Task Implementation

Figure 5.2 shows the implementation of the FPGA task from the host-based version of the Jacobi Solver application. Since the checkpoints are performed inside the main loop of the application, the FPGA task does not contain any checkpoint specific logic inside.

```
#pragma omp target device(fpga) copy_in([1]sizex, [1]sizey) \
        copy_out([1]error)
#pragma omp task in([MAX_BSIZE*MAX_BSIZE]h_mem, [1]sizex, [1]sizey) \
        inout([MAX_BSIZE*MAX_BSIZE]g_mem, [1]error)
void jacobi_fpga(float g_mem[MAX_BSIZE*MAX_BSIZE], float h_mem[MAX_BSIZE*MAX_BSIZE],
                                        float error[1], int sizex[1], int sizey[1]){
        int nbLines = *sizex;
        int M = *sizey;
        float localerror = 0.0f, diff;
        float h[(FPGA_BLOCK_ROWS+2)*362];
        float g[(FPGA_BLOCK_ROWS+2)*362];
        int nb = nbLines/FPGA_BLOCK_ROWS + 1;
        for(int b = 0; b < nb; b++){
                int lines = ((b+1)*FPGA_BLOCK_ROWS < nbLines) ?
                                FPGA_BLOCK_ROWS : nbLines-(b*FPGA_BLOCK_ROWS);
                if(b > 0) lines += 2;
                int start = (b == 0) ? 0 : b*M*FPGA_BLOCK_ROWS - 2*M;
                int size = lines*M*sizeof(float);
                memcpy(h, &h_mem[start], size);
                memcpy(g, &g_mem[start], size);
                for(int i = 1; i < (lines-1); i++){
                        for(int j = 1; j < (M-1); j++){
                                g[(i*M)+j] = 0.25*(h[((i-1)*M)+j]+h[((i+1)*M)+j]+
                                                        h[(i*M)+j-1]+h[(i*M)+j+1]);
                                diff = g[(i*M)+j] - h[(i*M)+j];
                                localerror += diff * diff;
                        }
                }
                memcpy(&g_mem[start], g, size);
        }
        *error = localerror;
}
```

Figure 5.2. Implementation of Host-based checkpointing for the Jacobi Solver application

In the implementation, the host copies all the matrix data to FPGA memory at once, and then calls the FPGA task. The FPGA task gradually copies the data from the memory to local variables (which are stored in the BRAM), and performs the computation. The task also copies the important data back to memory so that the host can gather the results after the task has finished.

5.1.2. Checkpointing partial work of the FPGA task

The intuition of this alternate implementation is to be able to checkpoint the progress of the FPGA task at certain points of its execution, without the need for the FPGA task to send its intermediate results to the host. Note that checkpointing inside the FPGA task is unnecessary for these specific applications, since the FPGA tasks only last a few seconds. However, this technique could prove useful when used on FPGA tasks which run for a long time before returning the final result.

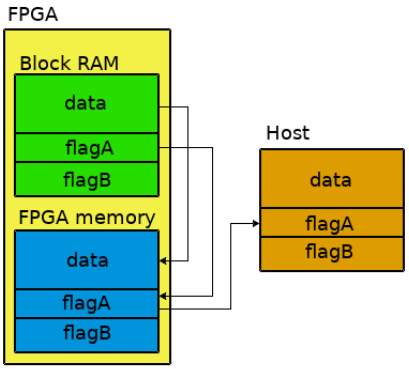


Figure 5.3. FPGA copies data to memory.

5.1.2.1. Adapting application to handle partial checkpointing

Because the host is unable to access the Block RAM of the FPGA, every data transmission or communication has to be done through the FPGA memory or stream through the PCI interface.

The first approach we have taken is to make the FPGA task and the host synchronize through the FPGA memory and communicate when a checkpoint is needed. That is, after a certain amount of work the FPGA would copy the work back to FPGA memory and set a flag (we will call it *flagA*) to tell the host that data is ready to be checkpointed. We can observe a representation of this in Figure 5.3.

The host would then copy the data that is relevant to host memory and perform a `FTI_Checkpoint`. While the host is copying the data and performing the checkpoint, the FPGA waits to avoid overwriting data and making the checkpoint in an inconsistent state. When the host finishes, it tells the FPGA task to resume its execution by setting another flag (we will call this one *flagB*) in the FPGA memory. We can see a representation of this in Figure 5.4.

For performance objectives, it is better to avoid synchronizations whenever possible. Thus, the next step was to implement a modified version of the previous approach which does not need synchronization between the host and the FPGA task. In this implementation, the host checks the progress of the FPGA in global memory while the FPGA task is constantly running.

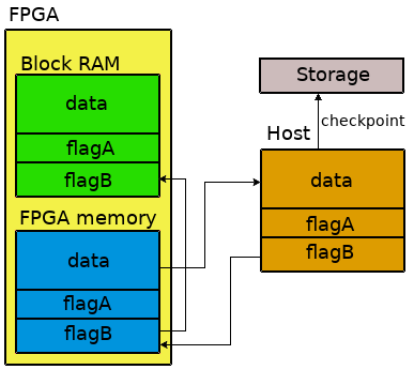


Figure 5.4. Host performs a checkpoint of partial FPGA data.

The idea behind this new approach is the following: if the FPGA task has started computing the data for checkpoint *i*, that means that the data for checkpoint *i-1* is ready to be copied (assuming HLS optimizations do not make the operations out of order). In order to have a secure way to

copy data without risking overrides, some structures will need to be replicated. That way, if we need data *a* for checkpoint *i*, we need to access it through *a[i]*.

In the case of the Jacobi solver, the FPGA task divides the computation into a fixed amount of blocks. Because of this, we decided that a checkpoint will be performed for every block computation. The local error needs to be replicated, since it is overwritten on every block iteration. The *g* matrix does not need to be replicated, since each block number modifies a different region of the matrix.

As for the YUV filter, we limited it to a total number of 3 checkpoints. In this case, the checkpoint is not related to the partial computation of the data, but rather to the stage in which the FPGA task is currently at. That is, checkpoint 1 refers to the image being converted from RGB to YUV, checkpoint 2 refers to the filter being applied and checkpoint 3 refers when the image has been converted back to RGB. This partitioning might look unintuitive for this specific application but it could prove useful depending on the use-case.

5.1.2.2. Partial work Checkpointing Task Implementation

As can be seen in Figure 5.5, the implementation of the checkpointing partial work of the FPGA task adds the required extra logic to handle the checkpointing and the recovery of the FPGA task progress.

```
#pragma omp target device(fpga) copy_in([1]sizex, [1]sizey)
#pragma omp task in([MAX_BSIZE*MAX_BSIZE]h_mem, [1]sizex, [1]sizey) \
        inout([MAX_BSIZE*MAX_BSIZE]g_mem, [MAX_BSIZE]error, [2]params)
void jacobi_fpga(float g_mem[MAX_BSIZE*MAX_BSIZE], float h_mem[MAX_BSIZE*MAX_BSIZE],
                 float error[MAX_BSIZE], int sizex[1], int sizey[1], int params[2]){
        int nbLines = *sizex;
        int M = *sizey;
        float localerror = 0.0f, diff;
        float h[(FPGA_BLOCK_ROWS+2)*362];
        float g[(FPGA_BLOCK_ROWS+2)*362];
        int nb = nbLines/FPGA_BLOCK_ROWS + 1;
        //if recovery is available, set localerror to the last protected block number
        if(params[0] > 0){
                localerror = error[params[0]-1];
        }
        for(int b = params[0]; b < nb; b++){
                params[0] = b; //update block number to global memory
                int lines = ((b+1)*FPGA_BLOCK_ROWS < nbLines) ?
                                FPGA_BLOCK_ROWS : nbLines-(b*FPGA_BLOCK_ROWS);
                if(b > 0) lines += 2;
                int start = (b == 0) ? 0 : b*M*FPGA_BLOCK_ROWS - 2*M;
                int size = lines*M*sizeof(float);
                memcpy(h, &h_mem[start], size);
                memcpy(g, &g_mem[start], size);
                for(int i = 1; i < (lines-1); i++){
                        for(int j = 1; j < (M-1); j++){
                                g[(i*M)+j] = 0.25*(h[((i-1)*M)+j]+h[((i+1)*M)+j]+
                                                   h[(i*M)+j-1]+h[(i*M)+j+1]);
                                diff = g[(i*M)+j] - h[(i*M)+j];
                                localerror += diff * diff;
                        }
                }
                memcpy(&g_mem[start], g, size);
                error[b] = localerror;
        }
        params[1] = 1; //notify finish
}
```

Figure 5.5. Implementation of Partial work Checkpointing for the Jacobi Solver application

If *param[o]* is found to be bigger than zero, it means that the FPGA task is resuming from an execution failure. In that case, it recovers the local error and the block iteration and continues the computation from there. Otherwise, it is assumed to be a normal execution and both the block iteration and local error variables are initialized to zero. The recovered matrices are already given to the FPGA task by the host, and so does not require additional logic. Unlike in the previous version, the local error needs to be copied to memory periodically to tell the host that there is a new checkpoint procedure available. Since the host has to be actively waiting for changes in memory, a *taskwait* pragma does not suffice as a way to know that the task has

finished, and so *param[1]* is used instead.

5.1.3. Environment

We performed the experiments on a cluster of 4 Xilinx Zynq-7000 SoC nodes. Each node is composed of a 32bit 2-core ARM processor with 1GB of DDR3 memory with an integrated Xilinx FPGA device. The FPGA device has the following specifications[2]:

- Logic Cells: 85
- Block RAM: 4.9Mb
- DSP Slices: 220
- Maximum I/O Pins: 200

As for the framework, we used version 2.2.0 of OmpSs@FPGA environment, along with Vivado Tools 2017.3 for the synthesis and generation of the FPGA bitstream.

The FTI version used was 1.4.1. The FTI configuration parameters were the following:

- No head (FTI dedicated process)
- 1 process per node
- 2 group size

The reason why we used 1 process per node is because the OmpSs@FPGA runtime implementation only allows one process using an FPGA device. Even though a dedicated FTI process would not make use of the device, the xTasks library (i.e., the library which is responsible for communicating with the FPGA device) still tries to access it at startup time and fails to execute if there's already another process accessing it. Because of this, the head option was not possible to test at this time, but it could be part of future work.

5.1.4. Applications

In this section we explain the applications that were used for the evaluation of this work.

5.1.4.1. K-means clustering

K-means clustering[20] is an iterative algorithm that assigns n observations into k clusters, k being a fixed parameter. The procedure is the following:

- k initial means are generated
- k clusters are generated from the observations based on the nearest mean
- The centroid of each cluster becomes the new mean
- Process is repeated until it converges

5.1.4.2. Jacobi Solver for heat equation

This algorithm consists in an iterative propagation of the heat represented as a 2D matrix. For every iteration, the average of the 4 neighbor positions (up, down, left, right) is assigned to every position (excluding the matrix borders). The algorithms stops when reaching a small enough error or when reaching a maximum amount of iterations.

5.1.4.3. YUV Filter

As the name implies, this algorithms consists on applying a YUV based filter to an image given as input. The steps are the following:

- Convert RGB image to YUV format
- Apply YUV filter to image
- Convert image back to RGB format
- Perform these steps a fixed amount of times

5.1.5. Host-Based FTI

5.1.5.1. K-Means

Since the K-Means application did not a long execution time, we decided to only evaluate the cost of a manual checkpoint using `FTI_Checkpoint`.

Two executions of the application were performed, the first using a ramdisk as the location for the checkpoint and the second using a drive mounted by Network File System (NFS). These experiments are situated on the opposite sides of the performance versus fault coverage trade-off state space. Taking the checkpoint to ramdisk is fast and thus high performance but since ramdisk is volatile on-device memory, this option does not provide fault coverage for FPGA node faults, but rather just for detected unrecoverable faults such as a double bit flip on the SECDED ECC implementations that are found in modern FPGAs. On the other hand, taking the checkpoint to NFS is slower but provides coverage for FPGA node faults.

For both executions, the application performs a single checkpoint in approximately the middle of the execution. This way, we can evaluate the cost of the checkpoint compared to the computation of the application in both cases.

The input used for the execution was 43000 points of 1024 dimensions each, stored in a file (173MB). The variables to be protected by FTI were the centroids, the labels, the current iteration, the current error and the error of the previous iteration. Taking this in consideration, the size of a L1 checkpoint is 0.20MB per process.

When using the ramdisk as L1 checkpoint location, the execution of the algorithm took 14.18 seconds. The checkpoint taken had a cost of 0.21 seconds. Knowing the checkpoint had to be taken in the same node as the computation (FTI dedicated process was disabled), we can observe an overhead of 1.5% due to checkpointing.

When using the NFS drive, the execution of the algorithm took 14.40 seconds. The checkpoint taken had a cost of 0.59 seconds. In this case, we can observe an overhead of 4.2% due to checkpointing. As expected, the overhead of writing the checkpoint in a ramdisk is much lower than when using a NFS drive. However, ramdisks have the downside of being volatile, making them not ideal for checkpointing purposes. The FTI library having 4 levels of checkpointing can help mitigate this weakness.

5.1.5.2. Jacobi Solver

For the Jacobi solver, we analysed the performance of the FTI library using checkpoint levels 2, 3 and 4 in separate experiments. For each experiment, we executed the application with FTI

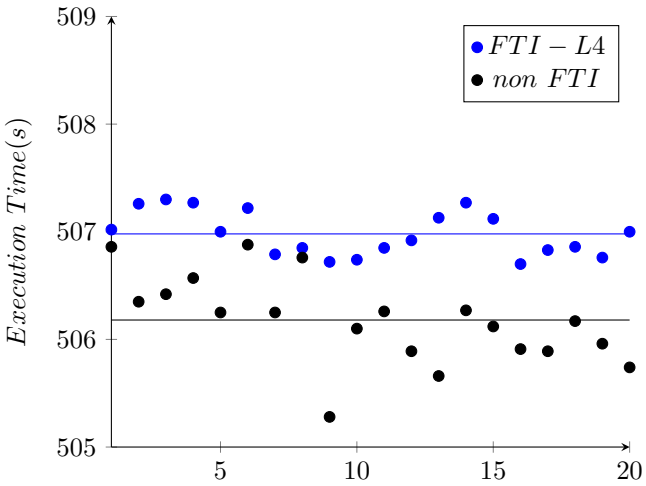


Figure 5.6. Performance comparison of Jacobi Solver implementation with FTI using L4 checkpoint and without FTI support

functionality for a total of 20 times. After the executions, we took the averages of each one and compared it to the execution of the application without FTI support. We interleaved the order of the executions to account for possible system slowdown over time.

For this application, we set a precision limit of 0.005 and a maximum number of 35000 iterations, which made the executions run for almost 10 minutes each.

As mentioned previously, this application makes use of the `FTI_Snapshot` functionality instead of using a manual checkpoint. We set an interval of 5 minutes of the desired checkpoint level in each experiment in the FTI configuration file, while disabling the rest. The reason why we decided on L4 checkpoint instead of L1 is because it made the most sense since our storage drive is NFS. However, we expect it to behave very similarly to a L1 checkpoint in this case due to there only being 4 nodes writing at the same time, meaning no bandwidth bottleneck should occur. going to start the evaluation with the L4 experiment.

In this case, the FTI library had to protect the iteration counter and both g and h matrices. The total size of a checkpoint is 0.26MB per process. The results for L4 experiment are shown in Figure 5.6. The lines represent the average execution time of each version. We observe that, on average, the overhead of `FTI_Snapshot` with the L4 checkpoint in this application is 0.16%.

Figure 5.7 shows the experiment with the L2 checkpoint, also known as partner copy. In this case, we also see a very low overhead of 0.25%. L2 checkpoint seems to have slightly higher overhead than L4. This is expected because of the extra communication and I/O required for saving an extra checkpoint copy of another node.

Experiment with L3 checkpoint is shown in Figure 5.8. As with the previous experiments, we can observe a slightly higher overhead of 0.32%. This level of checkpointing consists in the generation and communication of Reed-Solomon erasure codes in order to tolerate failures in multiple nodes, so the slight overhead increase is due to the extra computational workload from the nodes and the increased communication needed for the computation.

All 3 experiments show very low overheads in comparison to the original application. As such, we consider the performance of the FTI library to be a viable option for host-based checkpointing

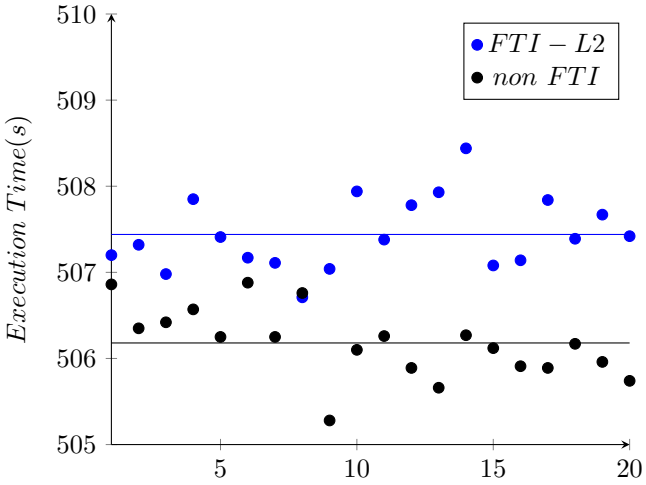


Figure 5.7. Performance comparison of Jacobi Solver implementation with FTI using L2 checkpoint and without FTI support

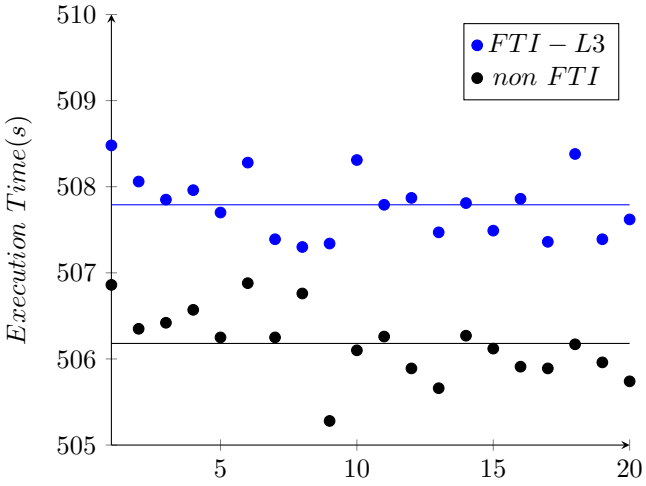


Figure 5.8. Performance comparison of Jacobi Solver implementation with FTI using L3 checkpoint and without FTI support

in FPGA heterogeneous systems.

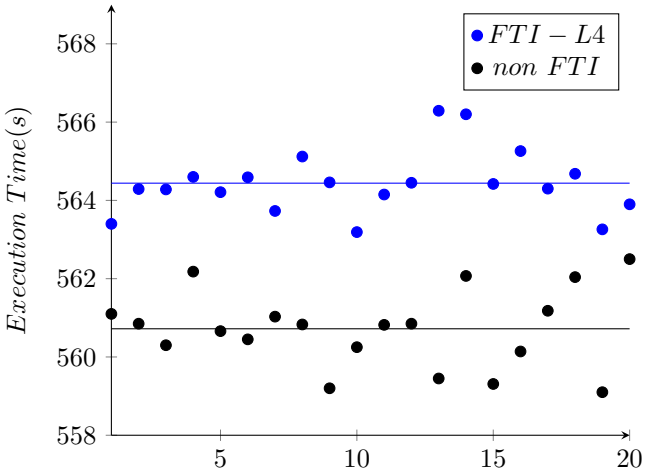


Figure 5.9. Performance comparison of YUV Filter implementations with and without FTI support

5.1.5.3. YUV Filter

Just like with the previous application, the YUV filter was executed 20 times with FTI and another 5 times without FTI. The `FTI_Snapshot` configuration was also the same as the previous application. For this application, we set it to a total of 80000 iterations in order to have a long enough execution time. We set the L4 checkpoint interval to 5 minutes and disabled the rest of the checkpoint levels. The FTI library protected the iteration counter and the image data (in which the filters are applied). The total size of a checkpoint is 0.16MB per process.

The results are shown in Figure 5.9. We can observe an overhead of 0.66% by using `FTI_Snapshot` and performing the L4 checkpoint, which is larger than the respective Jacobi solver experiment. One possible explanation could be that the amount of times that `FTI_Snapshot` is called is more than double (as many times as iterations), and thus the total overhead is increased. That said, this is still a very low overhead.

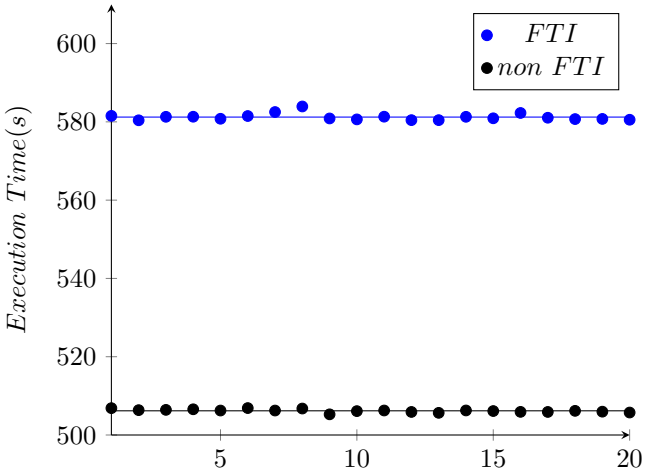


Figure 5.10. Performance comparison of Jacobi Solver implementation with partial FPGA task checkpointing and the non-FTI version

5.1.6. Partial FPGA Task Checkpointing

5.1.6.1. Jacobi Solver

We ran the FTI version of the Jacobi Solver that checkpoints partial work of the FPGA task a total of 20 times, the same as the previous versions. In Figure 5.10 we compare the original application with no FTI with this new version where the checkpoint is taken during the FPGA task execution. In this case, we can see an overhead of 14.82% by using this approach of FPGA checkpointing in comparison to the original non-FTI application.

While the overhead in this case is certainly larger than the host-based approach, it is worth pointing out that this application does not benefit from this method of checkpointing, and has considerably fast FPGA task execution times so the overhead of every FPGA task execution adds up. A better application for evaluating this method would be one with an FPGA task that takes significantly longer to give a final result.

Unfortunately, it was not possible to evaluate the YUV Filter version because the modified application did not fit in the BRAM of the boards.

5.2. Unvervolting High-Bandwidth Memories

Previously we had empirically evaluated an undervolting technique to improve the power-efficiency of Convolutional Neural Network (CNN) accelerators mapped to Field Programmable Gate Arrays (FPGAs). In this report, we focus on High-Bandwidth Memory (HBM). Modern high-performance devices employ HBM in order to meet high memory bandwidth requirements. HBM uses multiple DRAM chips piled together with the compute device (e.g., CPU, GPU, FPGA) as 3D stacks in the same package to have a wide interface of at least 1024 bits for data transfer. Although such an integrated HBM provides high bandwidth at a small form factor, the stacked memory modules consume a substantial portion of the chip's power budget. Therefore, power-saving techniques that preserve the performance of HBM are desirable. Undervolting is one such technique that reduces the supply voltage to decrease power consumption without decreasing the operating frequency of the device to avoid performance loss. Undervolting takes advantage of voltage guardbands put in place by manufacturers to ensure correct behavior under all environmental conditions. However, reducing voltage without changing frequency leads to reliability issues resulting in unwanted bit flips. We provide an experimental study of real HBM hardware under reduced-voltage operating conditions. Our measurements show that the guardband region in our HBM modules is 19% of the nominal voltage; pushing the supply voltage down to that region provides a 1.5X power saving gain for all bandwidth utilizations. Pushing voltage down further by 11% provides 2.3X power saving at the cost of unwanted bit flips. We explore and characterize the rate and type of these bit flips and show that the majority of them are clustered together in less than 16% of the memory's entire address space.

5.2.1. Structure of HBM

The general organization of an HBM-enabled device is shown in Fig. 5.11. In order to build an HBM-enabled device, several DRAM chips (and an optional IO/controller chip) are stacked on top of one another and connected by Through Silicon Vias (TSVs). An efficient way to utilize a stack like this is by placing it on a silicon interposer next to a computational unit (like FPGA, GPU, or CPU) inside the same package. Signals between memory stacks and computational units go through the silicon interposer. This means there can be far more data lanes (1024 per HBM stack compared to a regular 64-bit DRAM) and each one can be more efficient (smaller RC) compared

to DRAM modules which are placed outside the package on a PCB. As a result, HBM provides at least an order of magnitude higher bandwidth at lower power consumption (nearly 7pJ/bit as opposed to ∼25pJ/bit for a regular DRAM) within a much smaller form factor.

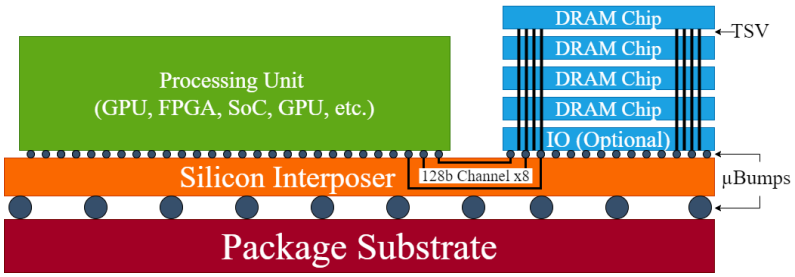


Figure 5.11. General structure of an HBM-enabled Device.

5.2.2. Hardware Under Test

The platform we use for our experiments consists of a Xilinx VCU128 board [31] mounted with an XCVU37P FPGA. This FPGA part includes two HBM stacks of 4GB totalling 8GB. Each stack has four DRAM chips with 1GB capacity each (similar to Fig. 5.11). Fig. 5.12 shows a general overview of the underlying HBM memory. The traditional FPGA fabric is divided into three Super Logic Regions (SLR). Each SLR is a separately fabricated chip with configurable circuitry. They are connected with the same technology that enables them to connect to HBM stacks.

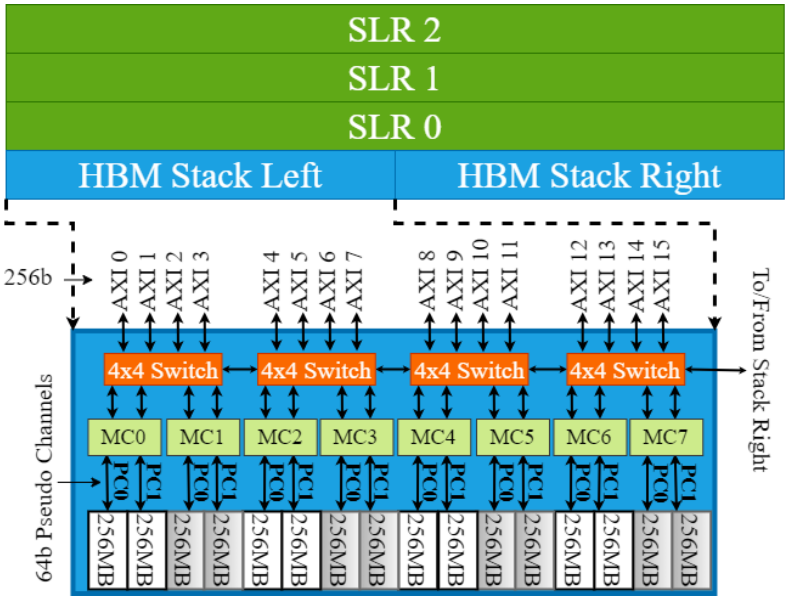


Figure 5.12. HBM Interface and internal organization of XCVU37P.

Address space of each stack is divided between 8 independent memory channels (MC) that are 128b wide and work on 512MB of memory that is assigned to them. Address space of each channel is subsequently divided between two 64b pseudo-channels (PC). These two PCs share clock and command signals but have separate data buses. They interpret commands separately and work with their own non-overlapping 256MB portion of the memory array. Therefore, on the memory side, there are a total of 32 PCs, 64b wide each. On the user's side, Xilinx's HBM IP core provides 32 AXI ports. Each AXI port corresponds to one PC. However, any packet from one AXI port can be routed to any PC if the switching network is enabled at the cost of extra delay and less bandwidth. These AXI ports are 256 bits wide which provides a 4:1 data width ratio over PCs. This implies that a customized hardware on the SLRs can work with a clock frequency that

is a quarter of memory data transfer rate and yet manage to take advantage of maximum HBM bandwidth. The maximum clock frequency allowed for memory arrays in our device is 900MHz, and being a DDR memory, it translates to a maximum data transfer rate of 1800 Mega-Transfers per second (MT/s).

We tune the supply voltage of our HBM stacks by accessing onboard voltage regulators through system controller interface of VCU128 board. We have implemented custom control commands in VHDL that communicate with HBM over UART. We also implemented controllers for the two HBM stacks. Each controller includes 16 AXI traffic generators (TG), one for each 16 AXI ports in their corresponding stack. The controller is in charge of configuring each TG, sending commands to it, receiving its response, checking its status and reporting statistics back to the host. Each TG implements customized macro commands that we later use to implement our test scenarios.

5.2.3. Power Measurement

Active power consumption in DRAMs is related to the square of the supply voltage (V_{dd}) as shown in equation (5.1) where C_L is load capacitance and f is operating frequency [3]. Therefore we expect a similar reduction in power consumption with undervolting.

$$P = C_L \times f \times V_{dd}^2 \tag{5.1}$$

We confirm this by experimental results shown in Fig. 5.13. This figure shows the power consumption of the entire HBM at different bandwidths. The maximum bandwidth we reach is 310GB/sec aggregated for two HBM stacks.

Working within the voltage guardband (1.2-0.97 V), provides a 1.5X power saving without introducing any faults. This is regardless of the bandwidth utilization of the memory. Pushing the supply voltage further down to 0.85V will give us a total 2.3X power saving compared to the default 1.2V.

As for *idle power*, we measure the power consumption of HBM when bandwidth utilization is zero. Even when HBM is idle, power consumption is nearly 30% of when HBM is at full load, regardless of the supply voltage. This means disabling all AXI ports except one will result in a 3X power saving instead of 32X.

We can also use equation (5.2), where I is the operating current, to interpret power consumption. Based on our experiments we already know that power consumption on the left side of equation (5.2) depends on the square of the supply voltage. Therefore we expect that the current consumed by HBM must drop linearly when the supply voltage scales down linearly. This is because the operating current depends on the supply voltage too [3].

$$P = V_{dd} \times I \tag{5.2}$$

Our experimental results in Fig. 5.14 confirm this expectation. As shown in this figure, the drop in operating currents is proportional to the drop in supply voltage at all bandwidths.

On the other hand, based on equation (5.1), if we divide our power measurement results by V_{dd}^2 we are left with raw values for $C_L \times f$. The dimension for these values is *farad per second* which shows how much active capacitance is being charged/discharged every second. In this equation, f is constant since the clock frequency of our design, clock frequency of HBM memory, and the

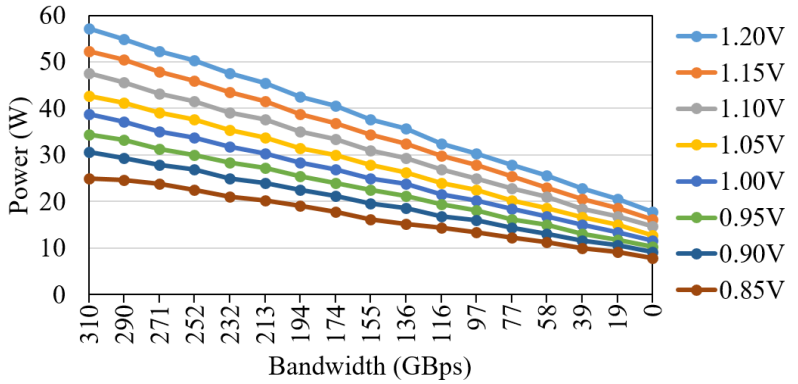


Figure 5.13. HBM stack power saving by undervolting (nominal voltage is 1.2V)

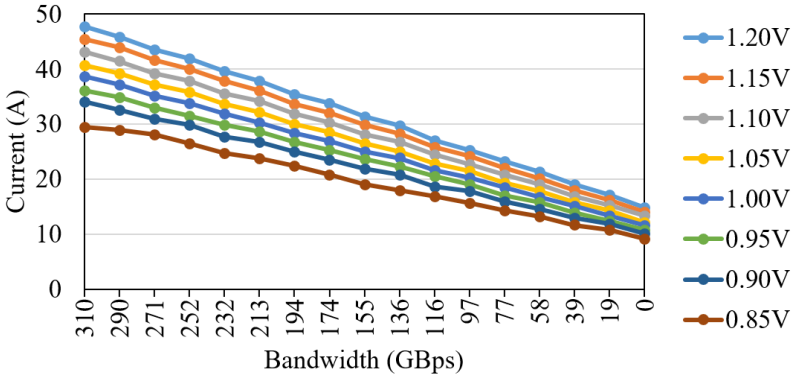


Figure 5.14. Operating current of HBM by undervolting (nominal voltage is 1.2V)

sequence that we run these tests are always the same.

Fig. 5.15 shows $C_L \times f$ of our entire HBM for all bandwidth utilizations and voltages that we tested. The active capacitance is linearly decreasing with bandwidth utilization. This is because to decrease bandwidth we progressively disable more memory channels and the capacitance that belongs to these channels is not charged/discharged. However, these regions of memory still consume idle power since they are left *pre-charged*. As voltages are pushed down and bit flips start manifesting, $C_L \times f$ for those voltages also drop since, with more bits that are stuck at 1 or 0, their capacitance is not active anymore. Active capacitance at 0.85V is far lower than the rest of the voltages since it introduces orders of magnitude more faults. We discuss the detailed reliability results in the next section.

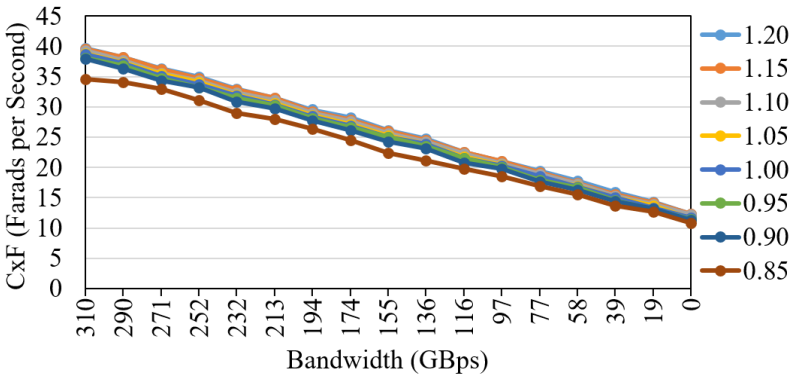


Figure 5.15. Active portion of the memory under load. Vertical axis shows how much capacitance is being charged/discharged every second.

5.2.4. Reliability Analysis Through Accessing Data Sequentially

Fig. 5.16 shows the number of stuck-at faults we encountered for each HBM stack. Here are our main observations:

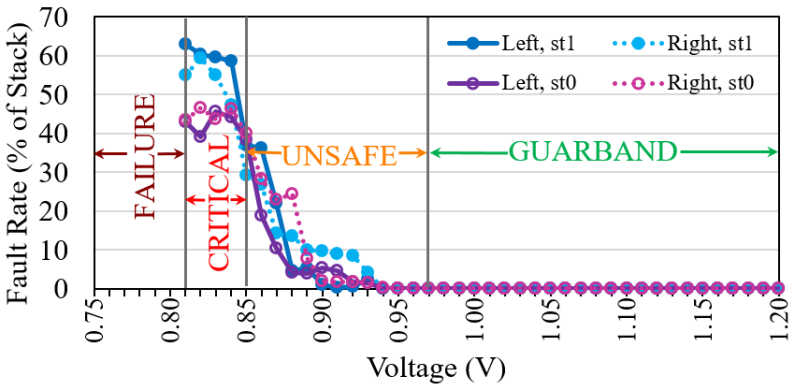


Figure 5.16. Percent of data bits in each stack that are st1 or st0 with undervolting.

- From nominal 1.2V down to 0.97V there are no faults in the memory. Taking advantage of this *guardband* region is safe since all operations can be carried out without faults.
- St0 and st1 faults start manifesting around 0.97V and 0.96V respectively. This is the beginning of the *unsafe* region where the design needs to take these faults into account to ensure correct operation. Reducing the voltage introduces additional faults in both scenarios in an *exponentially* growing manner until around 0.85V. This exponential growth has already been reported on DRAMs [10], although for different supply voltage range and fault count. In this region, the number of st1 faults remains on average 13% higher than that of st0 faults. Only two voltages behave differently: 0.85V and 0.88V. We believe these are a result of within-die process variation of the stacks.
- For voltages 0.84V down to 0.81V, early signs of the system failure show up as most bits are either st1 or st0. Working in this *critical* region is not suitable for any application since large portions of memory have very high fault rate.
- In our tests, 0.80V and voltages lower than that cause a complete *failure* of the memory system. Even restoring the supply voltage does not help and a power down is required.

5.2.5. Reliability Analysis Per Pseudo Channel

Fig. 5.17 shows the results of our tests on the fault rate per pseudo channel. We focused on voltages from 0.97V (edge of voltage guardband) down to 0.81V (just above the failure point of our HBM stacks). The number of faults is broken down for each AXI port (and their corresponding PC). The Left and right halves of the figure match the left and right HBM stacks in our system. The top and bottom halves show the number of st1 and st0 faults respectively.

5.2.5.1. st1 faults

These faults start at 0.96V for both stacks. Ten PCs have these faults although the fault rate is less than 0.0001%. With lower voltages, the fault rate for these PCs increase and other PCs start having faulty bits too until 0.93V when all PCs have at least one faulty bit. At 0.93V, most PCs have less than 2% st1 fault rate, except for PC5, PC18, and PC19. When we reach 0.85V, only three

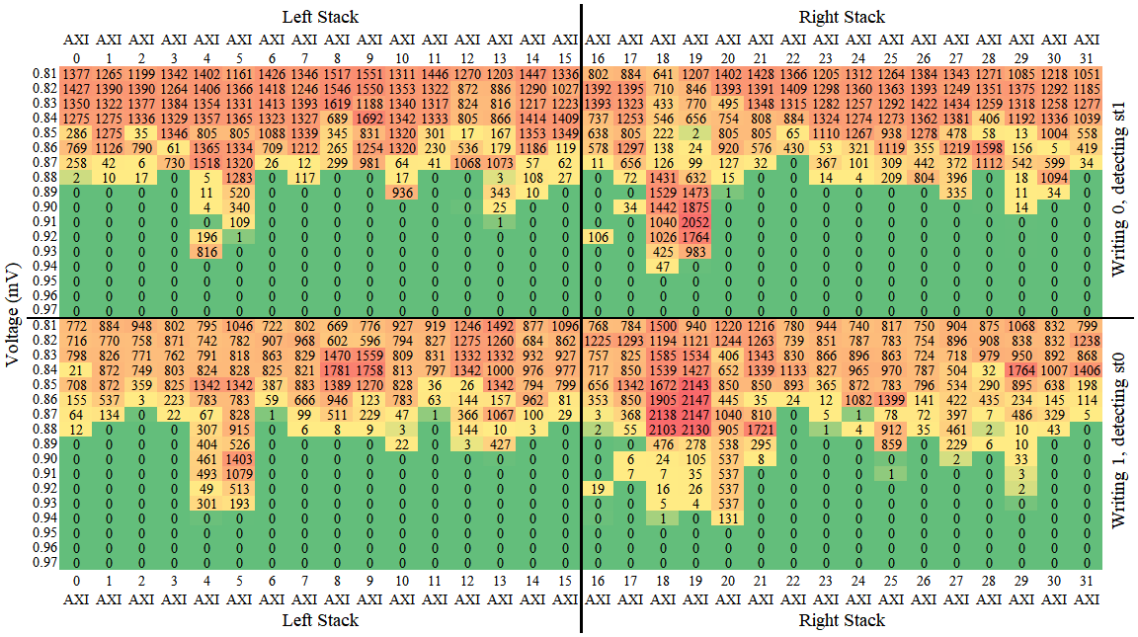


Figure 5.17. Fault count (in Millions) for AXI ports (and their corresponding PCs). Top half is for st1 and bottom half is for st0. Lower numbers mean a portion of memory is healthier and more reliable.

PCs have less than 1% and only 7 remain below 10% st1 fault rate. Below this point, all PCs have at least 18% fault rate.

5.2.5.2. st0 faults

The first faults appear as st0 faults at 0.97V. Only four PCs experience these faults: 5, 19, 20, and 22. As we reduce the voltage, the number of faults for these channels increase. Meanwhile, other PCs start showing faulty behaviour too. At 0.94V, all pseudo channels have at least one faulty bit that is stuck at 0. Fault rate of every PCs is less than 0.1% except only pseudo channel 20 that has nearly 6%. From 0.93V down to 0.90V, fault rate for PC20 goes up to 25% while five other PCs reach fault rates above 1%: 4, 5, 18, 19 and 29. By the time we reach 0.87V, only nine PCs have less than 1% fault rate. After this point, all PCs reach above 1%.

5.2.5.3. Pseudo Channel Comparison

Looking at how our two stacks behave when tested for st0 and st1 faults, we can see that some PCs are much more sensitive to faults than others. In our left stack, PC4 and PC5 are more sensitive than others: their faulty behaviour starts sooner than most others and they reach significant fault count before others. They reach 10% fault rate at 0.93V while the rest of the PCs in left stack start at 0.89V and all of them reach 10% or more around 0.86V.

As for right stack, PC18, PC19 and PC20 are more sensitive than others. Similar to PC4 and PC5, these PCs reach 10% fault rate at 0.93V while other PCs in right stack start having similar fault rates at 0.89V. This behaviour is due to process variation, some channels are weaker than others where more cells cannot keep their state. When the supply voltage is at 0.86V, all PCs from stack right have 10% fault rate similar to left stack.

5.2.5.4. Stack Comparison

Our experiments show that both stacks have a few PCs that are sensitive than others. Even the voltages that these PCs have significant changes in behaviour, such as reaching 10% threshold, are similar.

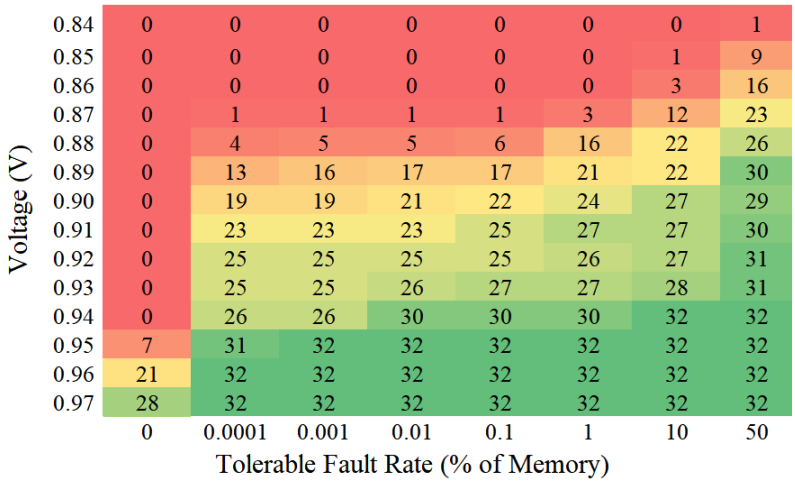


Figure 5.18. Number of PCs that can be used under different thresholds (showing the sum of st0 and st1 faults). Bigger numbers mean larger portions of memory are accessible.

5.2.5.5. Individual pseudo-channel observations

We understand that not applications have the same level of tolerance for faults when it comes to memory. Also, faults are not distributed uniformly across our HBM. Therefore we need an assessment of how much of our HBM can be used for a certain level of fault rate. Fig. 5.18 shows the number of PCs that can be used at each voltage based on an acceptable fault rate. Since pushing supply voltage down creates faults, the amount of power saving that a particular application can get with undervolting depends on how much memory that application needs and what fault rate it can tolerate in those regions of memory.

Those applications that cannot tolerate any faults and need the entire memory's address space available, have to work only in the guardband region of supply voltage which starts at nominal 1.2V and ends above 0.97V. Applications that work with a smaller amount of memory as long as that memory is fault-free, are able to push voltage further down to 0.95V at cost of disabling 25 PCs.

Those applications that are able to tolerate some faults as long as the rate of those faults is less than a certain threshold, will have more memory to work with. For example, an application that can tolerate a maximum of 1% fault rate and requires nearly half the memory (4GB in our case) can push voltage down to 0.88V and get nearly 2X power saving. A 2.3X power saving is possible if the application can tolerate up to 50% fault rate and is willing to work with only 9 PCs out of 32.

5.3. Intel SGX Framework Comparison

In LEGaTO project, we develop security mechanisms based on TEEs, e.g., Intel SGX or ARM TrustZone to ensure the integrity and confidentiality of legacy applications. However, the security guarantees come with performance overhead. To understand this overhead, we have developed the performance monitoring framework TEEMon [16] which has been reported in Deliverable D4.2. We use TEEMon to measure the overhead of Intel SGX frameworks such as SGX-LKL [26], and Graphene-SGX [29] and our toolchain SCONE [4]. We conduct this experimental evaluation to show that TEEMon is designed in a generic way so that it can be used for different Intel SGX frameworks without changing their source code; as well as to show the advantages of our toolchain SCONE compared to other SGX frameworks. In addition, we also demonstrate

that based on the performance metrics and statistics provided by TEEMon, we can identify the cause of bottlenecks of these SGX frameworks. We focus on a head-to-head comparison using the Redis in-memory key-value store as application, deployed and run inside Intel SGX enclaves using these SGX frameworks.

We benchmarked Redis (v5.0.5) running inside SGX enclaves using SGX-LKL,[1] SCONE,[2] and Graphene-SGX.[3] These SGX frameworks can run legacy applications Intel SGX without changing their code, simply by recompiling or relinking Redis using their provided compilation toolchains.

While Redis was executed directly on the host, we adapted the configuration of Redis to allow for stable execution with all frameworks. Foremost, we disabled the periodic creation of persistent snapshots, it requires the availability of the `fork()` system call within the SGX-enclave, which is not available in SGX-LKL and Graphene-SGX. Furthermore, we configured Redis to use at most 1 GB of memory, *i.e.*, the heap size of the enclave configured for all SGX frameworks.

We make use of the *memtier_benchmark* suite[4] to measure the performance of Redis and configure it to use 8 concurrent threads for optimal performance. Hence, the indicated number of connections is always a factor of 8.

First, we pre-populate the database with $720\,000$ keys. During the measurements, the benchmark issues GET requests. The *memtier_benchmark* is configured to use a pipeline of 8 requests and 8 connections per client-thread as these settings provided the best results in preliminary tests. We run experiments with different Redis database sizes (78 MB, 105 MB, and 127 MB) by setting the size of values (in the key-value messages) of 32 , 64 , and 96 bytes, respectively. The reason we conducted the experiments with different database sizes is that the most current SGX hardware supports only ~ 94 MB EPC size for applications running inside enclaves. When more memory is required, the applications inside enclaves need to perform the paging mechanism, usually very expensive performance-wise.

Next, we first present the performance comparison of Redis running with different SGX frameworks. Then, we describe how to use performance metrics captured by TEEMon to identify the bottlenecks of these SGX frameworks.

5.3.1. Performance Comparison

In the following, we discuss the performance measurements for Redis running with Intel SGX using SGX-LKL, SCONE, and Graphene-SGX. Note that the native version in this experiment is the vanilla Redis running without Intel SGX.

Throughput. Figure 5.19 shows the throughput of Redis using the different SGX frameworks. The native Redis achieves the throughput of 1.01 M - 1.2 M input/output operations per second (IOP/s) with different Redis database sizes at 320 client connections (Figure 5.19 (a)). The throughput of native Redis decreases when the number of connections is higher than 320. This is because, above 320 client connections, the host's network is squeezed at its capacity of 1 GBps.

Meanwhile, Figure 5.19 b depicts a similar behavior of throughput of Redis running with SCONE. The maximum throughput of SCONE is 278 KIOP/s at 560 connections ($\sim 23\%$ throughput of native Redis). The throughput of Redis with SCONE drops when the database size increases due to the EPC limitation of the SGX hardware. Increasing the database size from 87 MB to 105 MB reduces

[1] Commit ff8a1a3d, master branch.
[2] Commit fab5a2b7c, master branch.
[3] Commit e98be31, master branch.
[4] `https://github.com/RedisLabs/memtier_benchmark`

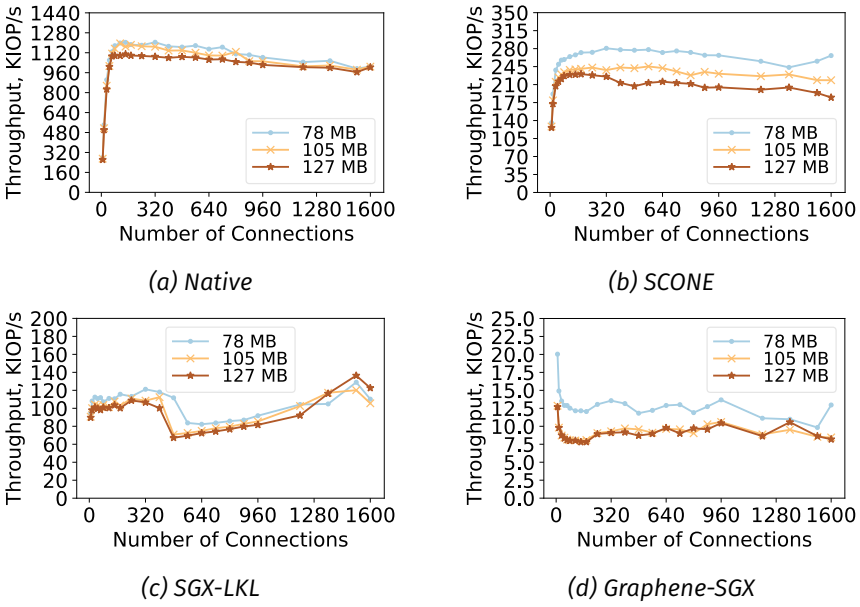


Figure 5.19. The throughput comparison between native Redis and Redis with different SGX frameworks. The total memory usage of Redis is set to different sizes of 78 MB, 105 MB, and 127 MB.

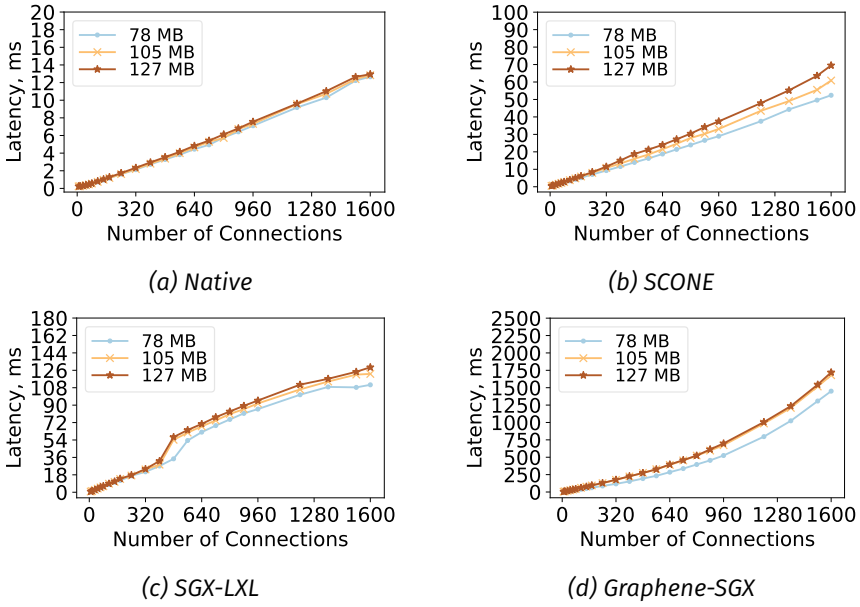


Figure 5.20. The latency comparison between native Redis and Redis with different SGX frameworks. The total memory usage of Redis is set to different sizes of 78 MB, 105 MB, and 127 MB.

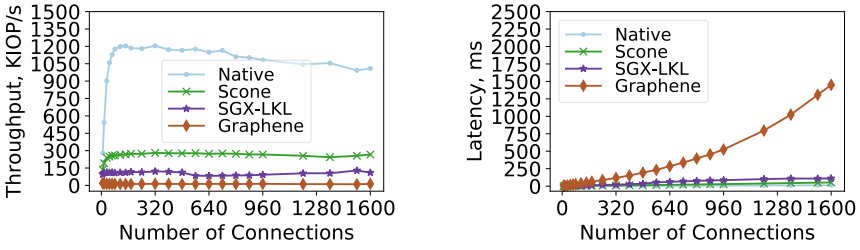
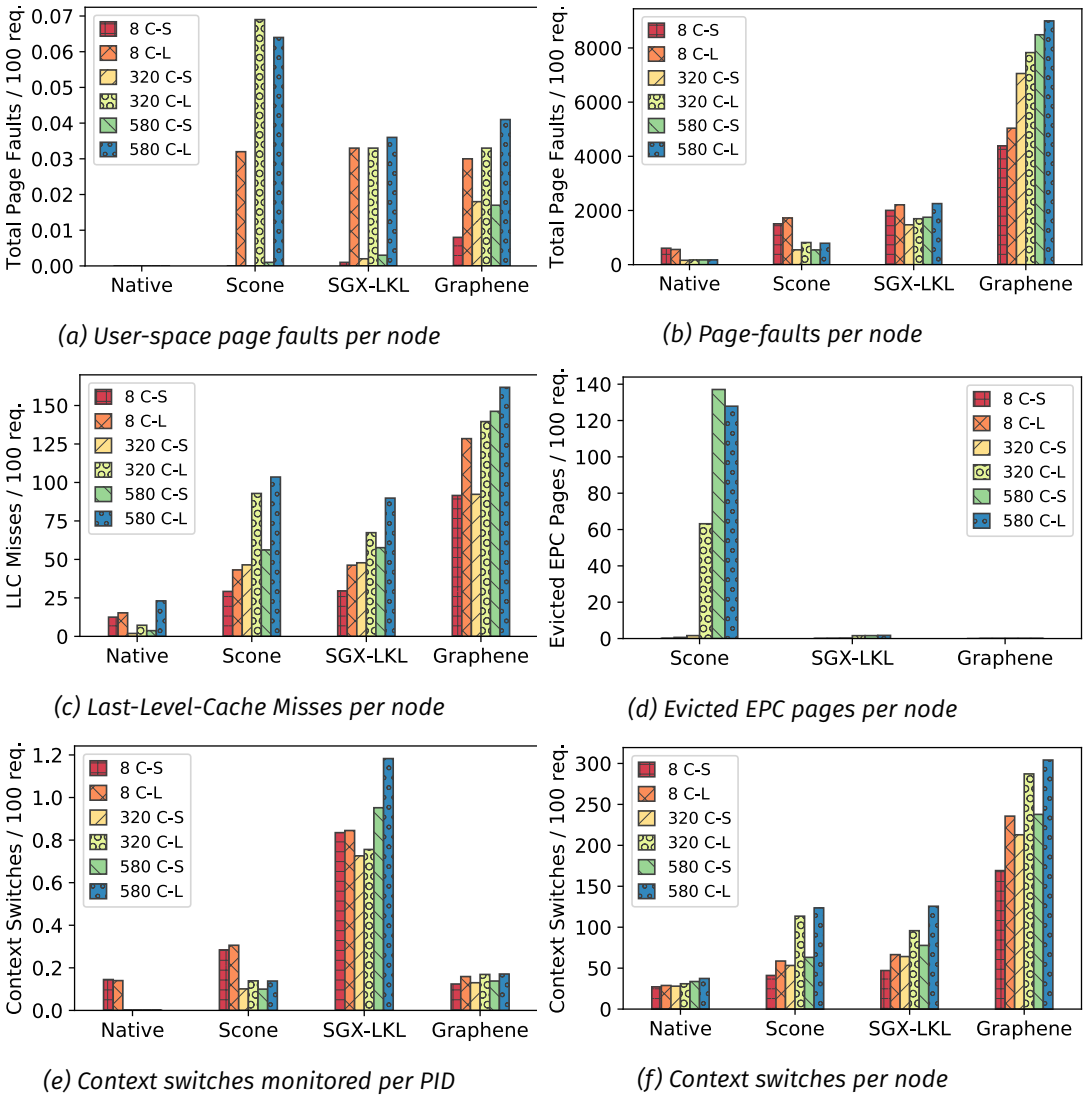


Figure 5.21. The throughput and latency comparison between native Redis and Redis with different SGX-frameworks. The total memory usage of Redis is set to 78 MB.



(a) User-space page faults per node

(b) Page-faults per node

(c) Last-Level-Cache Misses per node

(d) Evicted EPC pages per node

(e) Context switches monitored per PID

(f) Context switches per node

Figure 5.22. The detailed statistics of monitored performance metrics of native Redis and Redis running inside SGX enclaves using different SGX frameworks. The experiments are conducted with different configurations: 8 connections and 78 MB database size (8 C-S); 8 connections and 105 MB database size (8 C-L); 320 connections and 78 MB database size (320 C-S); 320 connections and 105 MB database size (320 C-L); 580 connections and 78 MB database size (580 C-S); and for 580 connections and 105 MB database size (580 C-L).

the peak performance of Redis with SCONE by 32 KIOP/s (decrease of 12%). Further increasing the database size to 127 MB decreases the peak performance at 29 KIOP/s.

Figure 5.19 c shows the results for the throughput of Redis with the SGX-LKL framework. While Redis with SGX-LKL peaks at 320 connections with 121 KIOP/s (∼10% of native Redis throughput), our results also show a steep drop in performance of Redis with SGX-LKL at 560 connections with a steady increase afterward.

Figure 5.19 d shows that, differently from the other SGX frameworks, Graphene-SGX performs best for one client (8 connections) and exhibits a reduced performance for more connections. The peak performance of Graphene-SGX was measured at 20 KIOP/s for 8 connections, (∼1.6% of native Redis throughput). Similar to SCONE, Figure 5.19 d shows a drop in throughput of Graphene-SGX if the database size increases from 78 MB to 105 MB. For a single client, the throughput decreases from 20 KIOP/s to 12 KIOP/s.

Latency. Figure 5.20 presents the Redis latency comparison between the different SGX frameworks. As expected, the latency of all evaluated systems increases when the number of connections increases. At 320 connections, the latency of the native Redis is ~ 2 milliseconds (ms), whereas the latency of Redis with SCONE, SGX-LKL, and Graphene-SGX are ~ 9 ms, ~ 20 ms, and ~ 249 ms, respectively. All latency measurements show an overall similar correlation between the number of connections and the latency. However, Redis with Graphene-SGX imposes a significantly higher latency compared to other frameworks. Figure 5.21 a and b show a performance comparison of native Redis and Redis with different SGX frameworks, with a database size of 78 MB and an increasing number of clients connections. In general, these results only show the overall performance trends of the SGX frameworks. To understand the insights of these SGX frameworks, we analyze the detailed performance metrics data during runtime and identify the bottlenecks using TEEMon.

5.3.2. Performance Metrics Analytics

Figure 5.22 presents the performance metrics statistics data of native Redis and Redis with different SGX frameworks, collected during benchmarks as described in §5.3.1. All presented statistics and data are similarly presented by the TEEMon front-end during a monitoring session.

Page Faults. Figure 5.22 a and Figure 5.22 b show the page faults in user space for Redis and the total page faults per host during the benchmark, respectively. The user space page faults include: `no_page_found`, `write_prot_fault`, `write_fault` and `instr_fetch_fault`.

While Figure 5.22 a indicates an overall low rate of user space page faults, it also shows that native Redis causes no page faults in user space. For the SGX frameworks, the rate of user space page faults increases with database sizes exceeding the EPC size (~ 94 MB). This happens when the SGX-enabled Redis reads data that was previously swapped out of the EPC and is unavailable for the current request. For 320 and 580 connections, with the database size of 105 MB, Redis with SCONE reaches the peaks of 0.069 and 0.064 user space page faults. Graphene-SGX and SGX-LKL show a similar pattern of page faults with ~ 0.03 page faults per 100 requests for larger database sizes. While SCONE and SGX-LKL introduce negligible page faults (*e.g.*, almost no page fault) with the database size of 78 MB which fits into EPC, the measurements show that Graphene-SGX still has the number of page faults of 0.02 per 100 requests.

In contrast to the low rate of user level page faults, Figure 5.22 b shows that on host-wide scope more page faults are registered. Native Redis has 607 total page faults per 100 GET requests for 8 connections, however, this number decreases (<170 page faults) for larger numbers of connections. This closely follows the finding that few connections lead to context switches in native Redis. While SCONE and SGX-LKL have the page fault rates increasing from 500 to 2200 total page faults per 100 GET requests, Graphene-SGX has a significant number of total page faults. For 580 connections and database size of 105 MB, Graphene-SGX has 8996 total page faults per 100 requests on average.

Last Level Cache Misses. Figure 5.22 c illustrates the last level cache (LLC) misses during the benchmark. Compared to native Redis, all SGX frameworks induce an elevated rate of LLC misses. With native Redis we observe $1.8 - 23$ LLC misses per 100 GET requests. Instead, SCONE and SGX-LKL achieve similar (yet higher) rates, *i.e.*, 29 to 103 LLC misses per 100 GET requests. Graphene-SGX has the highest LLC misses: 91 for 8 connections and 78 MB database size, and up to 161 LLC misses for 580 connections with 105 MB database size (per 100 GET requests).

Evicted EPC Pages. Figure 5.22 d shows the measured evicted pages from the enclave page cache

(EPC).

Graphene-SGX has at most 0.02 evicted pages per 100 GET requests for the database size of 78 MB which fits in the EPC. For the database size of 105 MB, Graphene-SGX exhibits at most 0.03 evicted pages per 100 GET requests. SGX-LKL shows a very similar behavior with up to 1.6 evicted pages (per 100 GET requests) for the database size of 78 MB and up to 1.7 evicted pages with the database size of 105 MB. Meanwhile, SCONE has a stark increase of evicted pages compared to other SGX frameworks.

With the number of connections of 580 and for the database size of 105 MB, SCONE has 137 evicted pages per 100 GET requests. We attribute the differences to the individual implementation and potential shortcomings of the framework's enclave memory management.

Context Switches. A common cause of SGX performance overheads is costly enclave transitions. The context switches were filtered by PID, to make it easier to monitor specific applications in the system. Figure 5.22 e shows these results and indicates that per 100 GET requests, Redis with SGX-LKL hits the most context switches. Instead, native Redis exhibits 0.14 context switches per 100 requests for the evaluation with just 8 connections. Since Redis uses an event queue and in combination with the findings shown in Figure 5.19 a, we conclude that, for 8 connections, Redis often waits (sleeps) for new messages and thereby causes context switches. With the exception of Graphene-SGX, SCONE and SGX-LKL show a similar pattern for 8 connections.

Figure 5.22 f shows the number of total context switches on the host while the GET requests are issued. The Figure suggests the total (host-wide) context switches of Redis with Graphene-SGX increases dramatically (up to $12\times$) compared to Redis with other SGX frameworks and native Redis. For 580 connections with a database size of 105 MB, Redis with Graphene-SGX has 304 context switches per 100 GET requests, while native Redis has only 37. SCONE and SGX-LKL expose a similar pattern as native Redis, with at most 125 context switches per 100 GET requests. We believe that Graphene-SGX has the lower performance as shown in Figure 5.21 because it has significantly more context switches than the other frameworks as reported by TEEMon.

Note that Figure 5.22 e shows only the context switches by Redis process itself, including its threads while Figure 5.22 f shows the total (host-wide) amount of context switches which includes the context switches between kernel processes as well as context switches to the `ksgxswapd` (Intel® SGX swapping daemon) process.

In summary, in this experiment, we show that TEEMon provides detailed performance data during runtime (e.g., cache misses, context switches, page faults, evicted EPC pages, etc) of applications (e.g., Redis) running inside Intel SGX which helps us to understand the performance behavior of the applications. The presented performance metrics by TEEMon are helpful for developers using SGX frameworks to identify performance issues and to provide guidance for improving the performance of these frameworks, especially with regard to scarce resources such as EPC memory and the expensive enclave exit and enter operations (due to system calls). This is achieved by presenting valuable graphs that show, e.g., high occurrences of the *clock_gettime* system call dominating the desired read-write system calls for network IO. While different metrics could in principal be gathered individually with different tools, TEEMon provides a single frontend for continuous and effortless monitoring of application to analyse their *behavior* in a production ready environment. In addition, we also demonstrate that our toolchain SCONE incurs less overhead compared to other SGX Frameworks. In LEGaTO project, SCONE has been used to develop a secure machine learning framework [18], a secure distributed data analytics

framework [19], a secure software update mechanism [22], and more.

6. Conclusion

In conclusion, this deliverable has evaluated the contributions described by D3.3 ("Final release of the task-based runtime"). This work package considered the middleware (interface to LEGaTO hardware and resource description), the back-end (energy-efficient mapping and utilization of hetergeneous hardware, and locality-aware execution), increased energy-efficient usage of FPGAs (via undervolting), and support for fault-tolerance and security (GPU and FPGA checkpointing and trusted execution). The RECS Master management software has shown resilience against data loss after potential shutdowns, and the Ethernet management architecture has been reworked by building a fabric over all Ethernet switches. XiTAO's Energy-Aware Scheduler (EAS) relies on a combination of per-task energy prediction and resource selection, and energy saving techniques during events of core under-utilization. Overall, by combining these techniques, it obtains energy savings up to 81% when compared to random work stealing on an asymmetric NVIDIA TX2 platform. The locality-aware scheduler that adopts the XiTAOs topologies shows that leveraging a highly dynamic scheme outperforms locality maximizing schemes by up to 30% for both classes of applications. Also, based on the performance trace of compute-bound and memory-bound task types, the scheduler shows the ability to tune the resources oblivious of predefined user-level annotations. OmpSs@FPGA has shown an efficiency of up to 7.32 GFlops/W using 2xIP core instances. Using the distributed variant of the OmpSs, i.e. OmpSs@Cluster, the Smart-Mirror use-case has an energy consumption up to 25W for the single node implementation, whereas in the OmpSs-2@Cluster, this dropped to 13W for the first node with 10W consumption for the per-processing of the raw captured image (i.e. scaling), and total of 17W for the second Xavier node. Undervolting has been shown to be an effective energy saving technique. Experimental results showed a 2.3X energy saving using HBM undervolting at all bandwidth utilizations. The checkpointing library has transparently worked with CPU, GPU and FPGAs under the same API. It has demonstrated the low-overhead, making heterogeneous computing more resilient and fault-tolerant. Finally, we have shown a performance comparison between state-of-the-art Intel SGX frameworks using the monitoring tool TEEMon, the first continuous performance monitoring and analysis tool for TEE-based applications. Using TEEMon, we have demonstrated that our toolchain SCONE incurs less overhead compared to other SGX Frameworks.

7. References

[1] Heat distribution. `https://github.com/leobago/fti`.

[2] Xilinx zynq-7000 soc zc702 evaluation kit. `https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html`.

[3] Calculating Memory Power for DDR4 SDRAM. Technical report, Micron Technology, Inc., 2017.

[4] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *2016 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, OSDI '16, pages 689–703. USENIX Association, 2016.

[5] Barcelona Supercomputing Center. Ompss-2 specification. https://pm.bsc.es/ftp/ompss-2/doc/spec. (Online; Last access: 11.05,2020).

[6] Leonardo Bautista-Gomez, Seiji Tsuboi, Dimitri Komatitsch, Franck Cappello, Naoya Maruyama, and Satoshi Matsuoka. Fti: high performance fault tolerance interface for hybrid systems. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*, pages 1–32, 2011.

[7] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of PPoPP '95*. ACM, July 1995.

[8] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[9] Jaume Bosch, Xubin Tan, Antonio Filgueras, Miquel Vidal Piñol, Marc Mateu, Daniel Jiménez-González, Carlos Álvarez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Application acceleration on fpgas with ompss@fpga. 12 2018.

[10] Kevin K. Chang and Others. Understanding Reduced-Voltage Operation in Modern DRAM Devices. *Measurement and Analysis of Computing Systems*, 1(1), 6 2017.

[11] Quan Chen, Minyi Guo, and Haibing Guan. Laws: Locality-aware work-stealing for multi-socket multi-core architectures. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 3–12, New York, NY, USA, 2014. ACM.

[12] christmann informationstechnik + medien GmbH & Co.KG. Extended Redfish API documentation. https://christmann.github.io/recs-redfish-api/index.html, 2020. (Online; Last access: 28.10.2020).

[13] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 329–338, New York, NY, USA, 2015. ACM.

[14] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, 2009.

[15] D.A. Knoll and D.E. Keyes. Jacobian-free newton–krylov methods: a survey of approaches and applications. *Journal of Computational Physics*, 193(2):357 – 397, 2004.

[16] Robert Krahn, Donald Dragoti, Franz Gregor, Do Le Quoc, Valerio Schiavoni, Pascal Felber, Clenimar Souza, Andrey Brito, and Christof Fetzer. TEEMon: A continuous performance monitoring framework for TEEs. In *Proceedings of the 21th International Middleware Conference (Middleware)*, 2020.

[17] A Kukanov and M Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11:309–322, 2007.

[18] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkeland, Pramod Bhatotia, and Christof Fetzer. secureTF: A Secure TensorFlow Framework. In *Proceedings of the 21th International Middleware Conference (Middleware)*, 2020.

[19] Do Le Quoc, Franz Gregor, Jatinder Singh, and Christof Fetzer. Sgx-pyspark: Secure distributed data analytics. In *Proceedings of the World Wide Web Conference (WWW)*, 2019.

[20] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[21] Jun Nakashima and Kenjiro Taura. Massivethreads: A thread library for high productivity languages. 8665:222–238, 2014.

[22] Wojciech Ozga, Do Le Quoc, and Christof Fetzer. A practical approach for updating an integrity-enforced operating system. In *Proceedings of the 21th International Middleware Conference (Middleware)*, 2020.

[23] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. Improving the integration of task nesting and dependencies in openmp. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 809–818, 2017.

[24] Miquel Pericàs. Elastic places: An adaptive resource manager for scalable and portable performance. *ACM Trans. Archit. Code Optim.*, 15(2):19:1–19:26, May 2018.

[25] Miquel Pericàs. Final release of the task-based runtime. Technical Report D3.3, May 2020.

[26] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the host OS interface for trusted execution. In *arXiv:1908.11143*, 2019.

[27] A. Rigo, C. Pinto, K. Pouget, D. Raho, D. Dutoit, P. Martinez, C. Doran, L. Benini, I. Mavroidis, M. Marazakis, V. Bartsch, G. Lonsdale, A. Pop, J. Goodacre, A. Colliot, P. Carpenter, P. Radojković, D. Pleiter, D. Drouin, and B. Dupont de Dinechin. Paving the way towards a highly energy-efficient and highly integrated compute node for the exascale revolution: The exanode approach. In *2017 Euromicro Conference on Digital System Design (DSD)*, pages 486–493, 2017.

[28] Rubén Cano Díaz. Communication in task-based runtimes for heterogeneous systems. Master Thesis, 2020.

[29] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of USENIX ATC*, 2017.

[30] Micha vor dem Berge. Final report on development and optimization of use-cases and integration. Technical Report D5.3, November 2020.

[31] Xilinx, Inc. *Virtex UltraScale+ HBM VCU128-ES1 FPGA Evaluation Kit*.

[32] A.B. Yoo, M.A. Jette, and M. Grondona. Slurm: Simple linux utility for resource management. In *Proceedings of the 9th Job Scheduling Strategies for Parallel Processing*, pages 44–60. LNCS, Springer, Berlin, Heidelberg, 2003.