SANNA: Secure Acceleration of Neural Network Applications

Akash Poptani, *IIT Dharwad*, Dharwad, India, 200020005@iitdh.ac.in
Abhishek Mittal, *IIT Dharwad*, Dharwad, India, 200030003@iitdh.ac.in
Rishit Saiya, *IIT Dharwad*, Dharwad, India, 180010027@iitdh.ac.in
Rajshekar Kalayappan, *IIT Dharwad*, Dharwad, India, rajshekar.k@iitdh.ac.in
Sandeep Chandran, *IIT Palakkad*, Palakkad, India, sandeepchandran@iitpkd.ac.in

***Abstract*—The threat of Hardware Trojans looms large on safety-critical systems. A Design-For-Trust technique to mitigate this threat without significant loss in performance is to implement these systems as a *Heterogeneous Secure System – HSS*. An HSS is built using an array of trustworthy home-grown cores and untrusted but fast third-party cores in a way that prevents unverified results from third-party cores reaching IO peripherals and devices. In this work, we propose to use the unverified results to initiate a speculative execution of subsequent layers of a Neural Network (NN) application on trustworthy cores. Our experiments on six popular NN applications show that on an average, the secure execution on an HSS is slower than the corresponding untrusted execution by up to 6.26% as compared to the slowdown of 80.89% experienced by a conventional trustworthy system.**

***Index Terms*—Hardware Trojans, Design-for-Trust, Neural Networks, Assisted Parallelization**

I. Introduction

Hardware Trojans (HTs) are deliberate modifications to a circuit made with malicious intent. These modifications can often have disastrous system-level consequences when they are deployed in-field and activated. The US took cognizance of the threat posed by HTs to its critical installations such as military equipment and other sensitive strategic infrastructure quite early and launched the Trusted Foundry Program (TFP) (in 2004). The goal of the TFP was to secure the supply chain of the electronic components used in its critical infrastructure by sourcing them only from trusted sources only. However, there is a recent push within the US to move away from the TFP into Zero-Trust Environments due to the economic infeasibility of maintaining state-of-the-art manufacturing facilities exclusively for military systems [1].

We account for these developments and assume the existence of two kinds of electronic components: (i) *trustworthy but slow* Home Grown Cores (HGCs), and (ii) *untrusted but fast* Third-party Cores (3PCs). Further, this work advocates constructing safety-critical systems using a combination of HGCs and 3PCs by adhering to the principle of *Sphere of Containment* [2]. This design principle prevents the malicious behavior triggered by an HT from reaching the IO peripherals by requiring the HGCs to verify every access to the peripherals. We call such systems *Heterogeneous Secure Systems (HSS)*. The implementation of an HSS is a Design for Trust (DFT) technique to counter HTs and can mitigate attacks from the Model D HTs [3]. It can handle attacks such as (i) changed functionality, (ii) leakage of sensitive information (through visible IO peripherals), and (iii) degraded or unreliable performance (in non-real-time systems). In this work, we assume that the 3PCs are fabricated separately from the HGCs, and hence each is a complete standalone system by itself that interact over Ethernet. However, this work is directly applicable to untrusted third-party IPs (3PIP) too that are integrated into a system at design time and fabricated in a trusted foundry together with HGCs.

The primary challenge in designing an HSS is to bridge the performance gap between the 3PC and the HGC. We achieve this by using *Assisted Parallelism*, which is in turn inspired by the domain of Assisted Execution. Unlike an earlier proposal that employed Assisted Parallelism to secure the execution of task-graph based applications [2], this work focuses on Neural-Network (NN) applications and do not require any changes to its source code.

Figure 1(a) shows the execution of a 3-layer NN application on an HSS when an HT is not triggered. HGC_1 acts as the master and schedules the execution of the first layer on HGC_2 and $3PC_1$. Since the computation on the 3PC is faster, the results of layer 1 from $3PC_1$ are available at the master before the same from HGC_2. The results from $3PC_1$ are untrusted because they have not been verified by the master at this point. These unverified results are forwarded to HGC_3 to start the computation of layer 2 speculatively. When layer 1 results from HGC_2 are available, the master compares them with the results received from $3PC_1$ earlier. If the results match, the ongoing computation (speculative until now) of layer 2 on HGC_3 is allowed to continue. Such overlapped execution of layers 1 and 2 on HGC_2 and HGC_3 using the unverified results from $3PC_1$ makes the execution of the NN application faster on an HSS as compared to a system constructed using HGCs alone. This increased performance comes at the cost of increased hardware overhead (additional HGCs and 3PCs).

Figure 1(b) illustrates a scenario where the results of layer 2 computed by $3PC_1$ and HGC_3 do not match because of an HT activation in the 3PC. In this case, the computation of layer 3 on HGC_2 that was speculatively initiated is aborted. The computation of layer 3 is restarted again on HGC_2 using the results from HGC_3. In a scenario where the computation of several subsequent layers has started using unverified results, all the speculative executions are aborted and intermediate re-

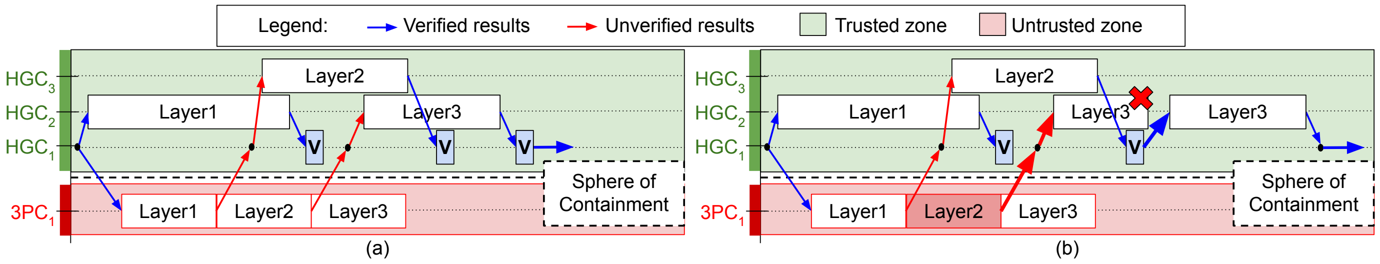

Fig. 1: Secure execution of a NN application on HSS: (a) when HT is not triggered, and (b) when HT is triggered in layer 2

sults discarded. The computations are restarted again on HGCs using the results from other HGCs. This ensures that unverified results do not reach the IO peripherals while enabling the overlapped execution of layers.

The performance of a NN application on such a system is maximized when the duration of computations on the different HGCs is similar and the volume of results transferred between the cores is minimal. The non-homogeneity in the layers in terms of computations performed as well as in the size of the results of each layer makes it challenging to achieve high performance.

In this work, we propose an Integer Linear Program (ILP) based technique to schedule the layers of an NN application on the HSS such that the overall runtime of the application is minimized without compromising the security of the system. Our experiments indicate that doing so helps the HSS outperform an HGC-only system by upto $5.4\times$ across six popular NN applications. The ILP technique, however, exhibits high latency, with the optimal solution remaining unfound in many cases even after ten hours. To alleviate this problem, we propose 4 alternate techniques to identify a good (near-optimal) schedule of execution on the available HGCs. These techniques run in a few milliseconds, and the best among them provides solutions that are $< 0.23\%$ slower on average than the optimal ILP solution.

II. Related Work

Hardware trojan countermeasures: The design of HTs and their countermeasures has been under active investigation for over a decade [4]. A popular class of countermeasures that comes under the ambit of DFT is the building of trusted systems using untrusted components. A technique under this approach is to execute different software implementations of the same functionality on untrusted hardware [5]. Another technique is to procure hardware implementations of desired functionality from different vendors [6]. These approaches are based on the expectation that two untrusted implementations (hardware or software) will always misbehave differently. Such an assumption may be risky in some contexts. To the best of our knowledge, design techniques that rely on the sphere of containment are the only ones that do not make this assumption.

Assisted Execution: Prior works have used assisted execution in three ways. The first way is to execute a trimmed version of the application and pass on beneficial hints to the original application to improve its performance [7]. The second way is used in Fault-tolerant systems extensively. Here an execution on a high-performance core is checked by a redundant execution on a simpler core aided by hints from the former [8]. The third way, SecCheck [2], was the first work to employ the idea of assisted execution in the realm of security to enable low-performance trustworthy cores to verify the functioning of high-performance but untrusted cores. However, SecCheck requires the application to be specified as a task graph so that the task-level parallelism can be easily exploited. In contrast, this proposal automatically infers opportunities for assisted parallelism by exploiting the nature of NN applications. In addition to this, we have proposed several heuristics that compute a near-optimal schedule of an NN application on an HSS. Further, we have evaluated our proposal on actual systems unlike the analytical methods adopted by SecCheck.

III. Assisted Parallelization of Neural Networks

The straightforward approach to parallelizing a NN's execution is to get each layer of the NN to execute on a different HGC. This is not always feasible because of the prohibitively high hardware overhead. In a HSS where the number of HGCs is less than the number of layer, scheduling the layers on to HGCs is non-trivial because of the asymmetry in the duration of execution of layers as well as the variations in volume of communication between the layers.

Table I lists the different symbols used in the description of the different proposed scheduling techniques.

A. Optimal HSS scheduling technique

We obtain the optimal schedule by formulating an ILP that takes into consideration the execution times of each layer as well as the volume of data transfers required.

The objective function is:

$$minimize \;\; T_{total} \tag{1}$$

The constraints of the ILP are:

✠ Every layer must be executed exactly once.

$$\forall_{l \in L} : \sum_{h \in HGCs} E_{l,h} = 1 \tag{2}$$

TABLE I: Symbols used in the description of techniques to schedule the application on the HSS

Symbol	Description
L	set of layers in the network
H	set of HGCs
T_{HGC_i}	time taken to execute layer i on HGC
T_{batch_i}	time taken to execute batch i on HGC
T_{3PC_i}	time taken to execute layer i on 3PC
T_{tr_i}	time to transfer layer i result from 3PC to HGC
T_{ready_i}	time taken by 3PC to execute layers 0 to $(i-1)$
$E_{i,j}$	layer i executed on HGC j
T_{start_i}	time when layer i starts HGC execution
T_{total}	total time required to securely execute the NN

✠ An HGC must execute a layer only if the layer inputs are available. If the previous layer has been executed on the same HGC, then the layer inputs need not be sought from the 3PC.

$$\forall_{l \in L, h \in H} : T_{start_l} \geq T_{ready_l} + T_{tr_{l-1}} \times (2 - E_{l-1,h} - E_{l,h}) \tag{3}$$

✠ An HGC must execute only one layer at a time.

$$\begin{aligned}\forall_{i \in L, j \in L, h \in H, i>j} : T_{start_i} \geq T_{start_j} &+ T_{HGC_j} \\ &- K \times (2 - E_{i,h} - E_{j,h})\end{aligned} \tag{4}$$

where K is a large constant.

✠ All layers have to complete execution on HGCs.

$$\forall_{l \in L} : T_{total} \geq T_{start_l} + T_{HGC_l} \tag{5}$$

Different instances of the problem were solved using the GNU Linear Programming Solver (glpsol).

B. Near-optimal HSS scheduling techniques

The near-optimal HSS scheduling schemes discussed below are aimed at reducing the time taken to obtain a schedule of the application on the HGCs without compromising too much on the overall performance of the system.

1) ApproxBatch: In this scheme, we logically coalesce contiguous layer into *batches* and execute each batch on a single core with the intention of balancing the duration of computations as well reducing the amount of data transferred between cores. It is important to note that the batching of layers has no implication on the security guarantees provided because the system continues to adhere to the principle of Sphere of Containment.

ApproxBatch ascertains the batch boundaries by making two approximations: (i) it considers only the computation time (ii) it assumes that the throughput of a 3PC is always x times that of an HGC irrespective of the nature of the layer. Therefore, x is as follows:

$$x = \frac{\sum_{i=0}^{|L|} T_{3PC_i}}{\sum_{i=0}^{|L|} T_{HGC_i}} \tag{6}$$

The ideal number of batches is $|H|$, and each batch will be executed on a different HGC. The ideal composition of the batches must be such that they complete their respective executions at the same time. For example, the requirement that $batch_0$ and $batch_1$ must complete at about the same time implies that the time taken to execute $batch_0$ (T_{batch_0}) should be equal to the sum of the time taken for executing $batch_1$ (T_{batch_1}) and the time taken to get its inputs (from 3PC). The latter term is the time taken to run $batch_0$ on a 3PC, which according to equation 6, is approximately $x \times T_{batch_0}$. Constructing the other equations in a similar fashion gives us:

$$\begin{cases} T_{batch_0} & = T_{batch_1} + x \times T_{batch_0} = \frac{T_{batch_1}}{1-x} \\ T_{batch_0} & = T_{batch_2} + x \times T_{batch_1} + x \times T_{batch_0} = \frac{T_{batch_2}}{(1-x)^2} \\ & \cdots \\ T_{batch_0} & = \frac{T_{batch_{|H|-1}}}{(1-x)^{|H|-1}} \end{cases} \tag{7}$$

In addition to this, we also know that the time taken to execute all the batches on the HGCs is equal to the time taken to execute all layers separately.

$$\sum_{i=0}^{|H|-1} T_{batch_i} = \sum_{i=0}^{|L|-1} T_{HGC_i} \tag{8}$$

Solving the above system of equations gives us the estimates of the ideal values of the different T_{batch_i}'s. We then find the layer boundaries closest to these values, leading us to the composition of the batches.

2) GreedyHGC and GreedyECT: The *GreedyHGC* and the *GreedyECT* techniques use a greedy approach to compute the schedules. The former prioritizes the layers with longer execution times on HGC (T_{HGC_i}) and balances their execution across HGCs. The latter prioritizes those layers that have a higher *earliest completion time* (T_{EC}) (see equation 9). Here, T_{EC} is defined as the earliest absolute time a layer can complete its execution, including the time it takes to verify the results by re-executing it on an HGC.

$$T_{EC_i} = \sum_{j=0}^{i-1} T_{3PC_j} + T_{tr_{i-1}} + T_{HGC_i} \tag{9}$$

Algorithm 1 details the working of the greedy approach. The $metric$ is set to T_{HGC} or T_{EC} in Line 1 based on the heuristic used. The algorithm goes over the layers in the order of their priority considering one layer at a time. It then finds the time when the inputs of the layer under consideration will be available at an HGC (T_{avail}). Lines 6 to 10 finds an HGC (j) that is free for the duration of the execution of the layer and schedules it onto j.

3) TaskStealing: Under this scheme, we use the task-stealing paradigm to schedule layers on HGCs and is detailed in Algorithm 2. For each of the HGCs, the time when it will be free next is maintained. Then, all the layers are considered one at a time. For each layer (i), the time of when its inputs will be available at an HGC (T_{avail}) is computed next. Line 4 identifies the HGC (j) that is either free now, or will become free first (in future). Lines 5 to 8 schedules the layer i on to the HGC j.

Algorithm 1 Greedy approach to scheduling tasks on the HSS

```
1: $metric = T_{HGC}$ or $T_{EC}$
2: $s\_layers = layers$ in descending order of $metric$
3: for $k \leftarrow 0$ to $|L| - 1$ do
4:     $i = s\_layers[k]$
5:     $T_{avail} = T_{ready_i} + T_{tr_{i-1}}$
6:     find earliest time $t$ such that $t \geq T_{avail}$ and an HGC is free in $[t, t + T_{HGC_i}]$
7:     $j =$ an HGC that is free in $[t, t + T_{HGC_i}]$
8:     $E_{i,j} = 1$
9:     $T_{start_i} = t$
10:    set $j$ as occupied in $[t, t + T_{HGC_i}]$
11: end for
```

Algorithm 2 TaskStealing approach to scheduling tasks on the HSS

```
1: $HGC\_nextFreeAt = [0...0]$
2: for $i \leftarrow 0$ to $|L| - 1$ do
3:     $T_{avail} = T_{ready_i} + T_{tr_{i-1}}$
4:     $j = \min(HGC\_nextFreeAt)$
5:     $startTime = \max(T_{avail}, HGC\_nextFreeAt_j)$
6:     $E_{i,j} = 1$
7:     $T_{start_i} = startTime$
8:     $HGC\_nextFreeAt_j = startTime + T_{HGC_i}$
9: end for
```

IV. System Design

We begin by first profiling the NN application's execution on the 3PC and the HGC separately to get the timing characteristics of the given NN. We then find the schedule of the different layers on the HGCs by adopting one of the techniques listed in Section III.

At runtime, our proposed framework first launches a `master` process on one of the HGCs. The task of the `master` process is to orchestrate the entire execution of the application on the HSS under consideration. The master initiates the execution of the application on the `3PC_slave` – a process on the 3PC – with the given inputs. The `master` also initiates the execution of the application on an array of HGCs. A multitude of `HGC_slaves`, each on an HGC, work together to execute the application. The `3PC_slave` sends the results of each layer to the `master` and the `master` forwards the same to the appropriate HGC based on the computed schedule. The `HGC_slave` receives the inputs of layer i from the `master` and executes the same, and returns the results to the `master`. For each layer, the master compares the results returned by the `3PC_slave` and the corresponding `HGC_slave`. An HT activation is inferred if the results do not match, and a suitable reaction strategy such as re-executing all subsequent layers only on HGCs using the result of the previous layer available from an `HGC_slave` is adopted.

TABLE II: Specifications of the Evaluation Platform

Big Core	
# cores: 4; ISA: ARMv7; Micro-arch: Cortex-A15 (3-issue, out-of-order) Studied Frequencies: 800 MHz, 1200 MHz, 1600 MHz, 2 GHz	
Little Core	
# cores: 4; ISA: ARMv7; Micro-arch: Cortex-A7 (2-issue, in-order) Studied Frequencies: 200 MHz, 600 MHz, 1000 MHz, 1.4 GHz	
Memory: 2GB	OS: Ubuntu 16.04

V. Evaluation

A. Methodology

We evaluated our proposal using multiple Odroid MC1 Solo boards. Each board consists of 4 "big" high-performance cores and 4 "little" low-performance cores. Table II shows the detailed specifications of the cores (and boards). A big core on one of the boards was designated as the 3PC, while the little cores on the other boards function as HGCs. Also, the big core operates at a frequency higher than that of the little cores. Standard Linux TCP sockets over a wired Gigabit Ethernet network are used to transfer the results from the 3PC to the HGCs. The communication time ranged from a few ms when a few KB were transferred, to a couple of seconds when the layer results were a couple of MB.

This experimental setup serves as an accurate testbed for evaluating the performance of an HSS. The big core plays the role of the Commercial Off-the-shelf (COTS) general-purpose processor (an untrusted component of Model D [3]). The little cores play the role of the low-performance trustworthy HGCs. The 3PC is on a separate board, with its own memory and disk, and can only communicate with the HGCs over the Ethernet, hence is isolated from the IO peripherals.

We experimented with different sizes of the HGC array and scaled the frequencies of the cores to study the trade-offs between the hardware overhead and the performance improvements obtained with respect to the performance disparity between the 3PC and the HGCs. Table II lists the frequencies studied in this work. The following shorthand notation is used to refer to the configurations studied in this work: `B`$\langle F_B \rangle$`_L`$\langle F_L \rangle$, where F_B refers to the frequency of the Big (3PC), and F_L refers to the frequency of the Little (HGC) cores.

We have chosen 6 popular NNs for our evaluation – resnet18 (68), alexnet (21), vgg16 (39), squeezenet1_0 (65), googlenet (196), mobilenet_v2 (158), where the number of layers is indicated in brackets. The NNs were executed using the Pytorch v1.1.0 and the TorchVision v0.3.0 frameworks. The TorchProfile library was used for timing characterizations. Layer execution times ranged from a few μs to a few seconds

B. Results

Figure 2 shows the execution times of the HSS (OptimalILP schedule) relative to a 3PC-only system. The missing bars indicate those configurations for which the ILP solver (glpsol) was unable to come up with a solution in a reasonable amount of time (10-hour timeout). Table III lists the maximum and mean slowdowns observed for different system configurations

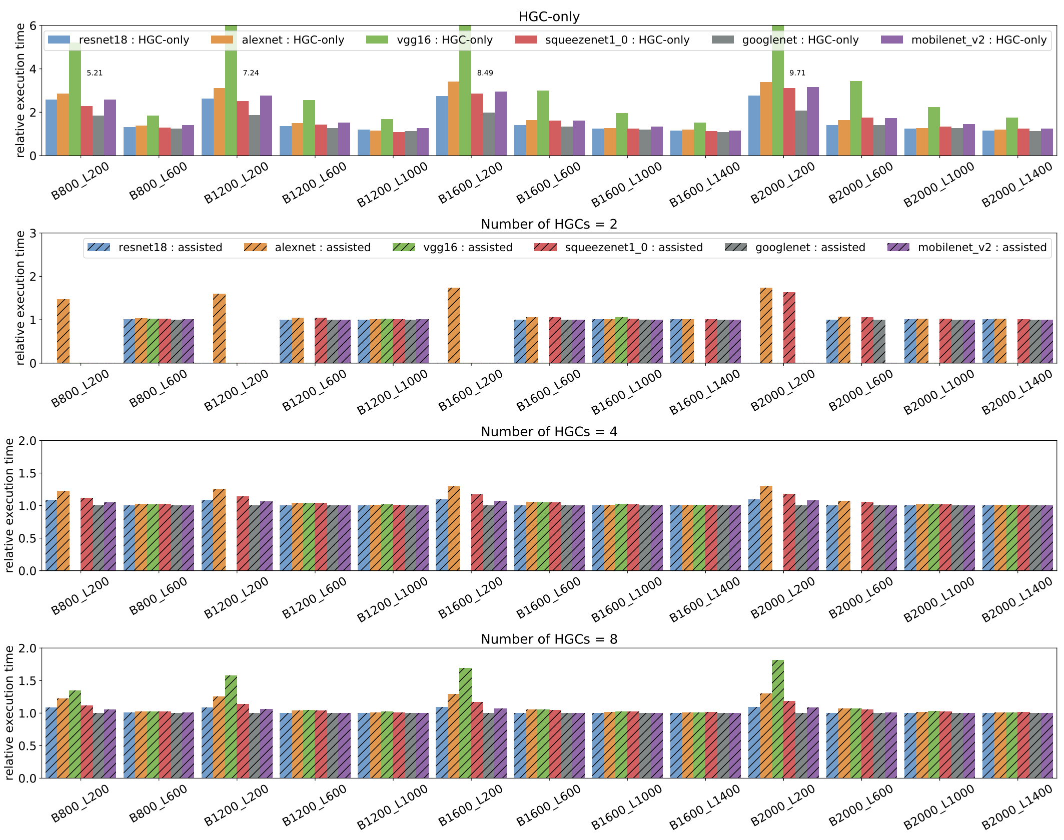

Fig. 2: Performance of the proposed HSS (optimal ILP-based schedule) and a HGC-only system relative to a 3PC-only system

relative to the 3PC-only system and the OptimalILP system. We can see that the proposed HSS far outperforms the HGC-only system at every design point – across (i) varying number of HGCs, (ii) a wide range of 3PC and HGC frequencies, and (iii) all the studied applications.

We observe that on average, the performance loss on an HSS that uses OptimalILP is under 6.26% as compared to the 3PC-only system. However, the corresponding HGC-only system is slower than the 3PC-only system by 80.89%. Therefore, the proposed scheme of assisted parallelization has improved the system performance without compromising the security of the system (due to adherence to the principle of the sphere of containment). However, this has come at a cost of increased hardware resources (upto 7 additional HGCs).

The maximum benefits of the proposed scheme is seen in the case of `vgg16` where the time taken by the `HGC-only` execution was 9.71× that of the 3PC-only execution (`B2000_L200`), while the time taken by an HSS with 8 HGCs was only 1.81× which is an improvement of 5.4×.

Figure 3 shows the performance of the HSS when using the different scheduling techniques proposed. The Near-optimal techniques were programmed in C++, and produced schedules in a few milliseconds for each of the design points. In comparison, the ILP solver took anywhere from a few seconds to a few hours. In many cases, no solution was produced even after the timeout period of ten hours. Each bar represents the geometric mean of the relative execution times (relative to the 3PC-only system) of all six applications across those configurations for which the ILP solver produced a schedule. The configurations where the ILP solver did not return a solution are not considered. The mean and maximum slowdowns are also summarized in Table III.

TABLE III: Slowdowns of different trusted systems relative to a 3PC-only and a OptimalILP system

# HGCs	**System**	**Slowdown (in %) relative to**			
		3PC-only		**OptimalILP**	
		Max	**Mean**	**Max**	**Mean**
2	HSS-OptimalILP	73.84	6.01	0.0	0.0
2	HSS-ApproxBatch	94.99	12.98	23.61	6.58
2	HSS-GreedyHGC	77.55	10.86	16.18	4.58
2	HSS-GreedyECT	128.01	19.34	38.77	12.57
2	HSS-TaskStealing	74.48	6.26	2.85	0.23
2	HGC-only	240.28	44.23	95.75	36.05
4	HSS-OptimalILP	30.2	3.9	0.0	0.0
4	HSS-ApproxBatch	57.47	8.27	33.81	4.2
4	HSS-GreedyHGC	30.18	3.96	0.13	0.06
4	HSS-GreedyECT	98.47	5.83	53.62	1.86
4	HSS-TaskStealing	30.22	3.97	0.56	0.07
4	HGC-only	240.28	65.91	192.38	59.67
8	HSS-OptimalILP	81.09	6.26	0.0	0.0
8	HSS-ApproxBatch	112.31	7.49	17.24	1.16
8	HSS-GreedyHGC	81.02	6.31	0.13	0.05
8	HSS-GreedyECT	117.0	6.65	28.24	0.37
8	HSS-TaskStealing	81.05	6.31	0.13	0.05
8	HGC-only	871.08	80.89	436.23	70.24

We observe that across all design points, on an average, all the near-optimal techniques produce schedules almost as

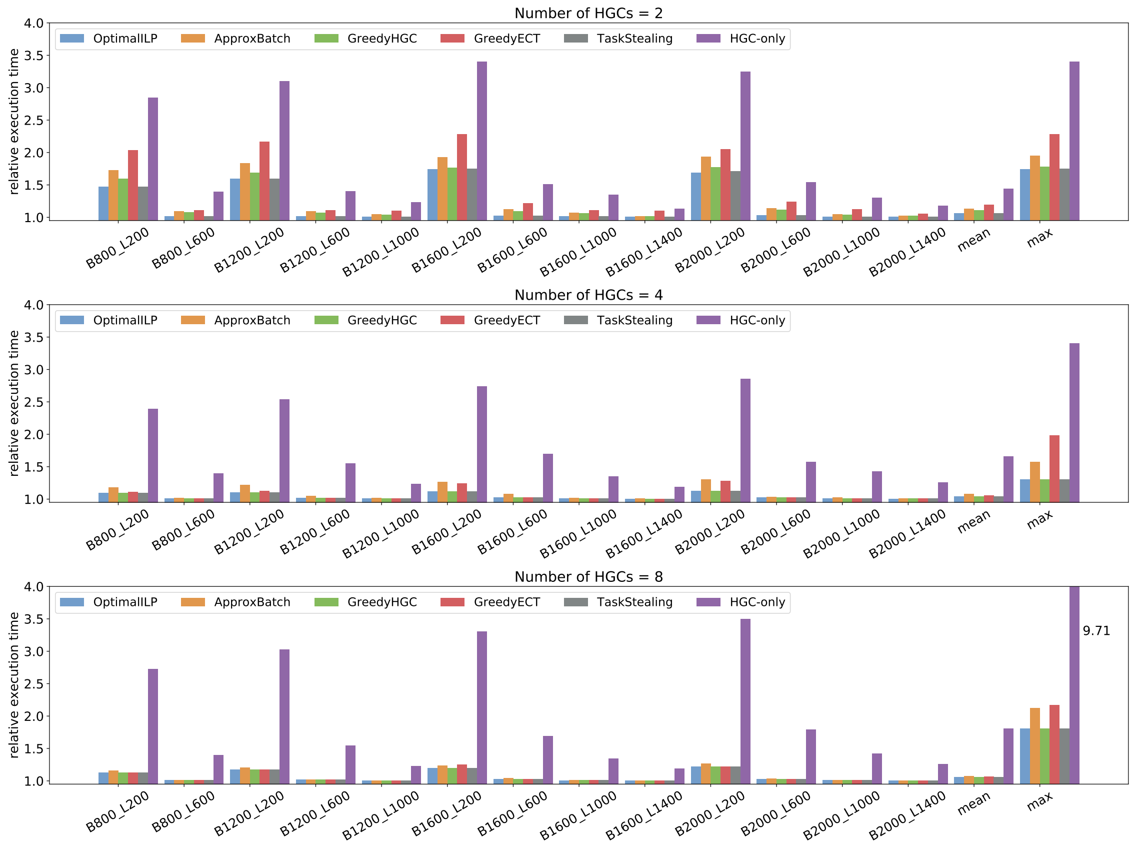

Fig. 3: Performance comparison of the proposed scheduling techniques

good as OptimalILP. The TaskStealing heuristic is the best among the four techniques with the mean loss in performance of $< 0.23\%$ of OptimalILP. The average loss in performance of GreedyHGC, ApproxBatch, and GreedyECT are $< 4.58\%$, $< 6.58\%$ and $< 12.57\%$ respectively. We also note that the maximum loss in performance of TaskStealing, GreedyHGC, ApproxBatch and GreedyECT are 2.85%, 16.18%, 33.81%, and 53.62% respectively.

1) Performance disparity in 3PC and HGC: The studies indicate that the greater the disparity in the performance (frequencies) of the 3PC and the HGC, the greater the benefit obtained through assisted parallelization. Thus, the proposed scheme allows trustworthy chips fabricated under technologically constrained programs such as the TFP to continue to be used in sensitive systems as HGCs without significant loss in performance.

VI. Conclusion

We outlined a DFT technique to construct an HSS that uses a combination of low-performance trustworthy and high-performance untrusted cores. Further, we proposed a technique of Assisted Parallelization for NN applications executing on an HSS such that the loss in performance as compared to the state-of-the-art is significantly reduced without compromising on security. Additionally, our proposed techniques requires no modifications to the source code of the NN application. Our evaluation on six NN applications showed significant performance improvements in the execution times of the applications on the HSS as compared to a conventional HGC-only system.

References

[1] V. Khemani, M. H. Azarian, and M. G. Pecht, “Prognostics and secure health management of electronic systems in a zero-trust environment,” in *Annual Conference of the PHM Society*, vol. 13, no. 1, 2021.

[2] R. Kalayappan and S. R. Sarangi, “Seccheck: A trustworthy system with untrusted components,” in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2016, pp. 379–384.

[3] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, “Hardware trojans: Lessons learned after one decade of research,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 1–23, 2016.

[4] S. Bhunia and M. Tehranipoor, *The Hardware Trojan War*. Springer, 2018.

[5] C. Liu, J. Rajendran, C. Yang, and R. Karri, “Shielding heterogeneous mpsocs from untrustworthy 3pips through security-driven task scheduling,” *IEEE Transactions on Emerging Topics in Computing*, vol. 2, no. 4, pp. 461–472, 2014.

[6] T. Reece, D. B. Limbrick, and W. H. Robinson, “Design comparison to identify malicious hardware in external intellectual property,” in *2011IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2011, pp. 639–646.

[7] Z. Purser, K. Sundaramoorthy, and E. Rotenberg, “A study of slipstream processors,” in *Proceedings of the 33rd annual ACM/IEEE International Symposium on Microarchitecture*, 2000, pp. 269–280.

[8] T. M. Austin, “Diva: A reliable substrate for deep submicron microarchitecture design,” in *MICRO-32. Proceedings of the 32nd Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1999.