Tokyo Institute of Technology 2014 second term class

High Performance Computing

Resume (Jan. 5, 2015)

Masahiro Yano
Department of Biological Information, Tokyo Institute of Technology
Kurokawa Nakashima Yamada Lab.

Today's paper

Title:
HAUBERK: Lightweight Silent Data Corruption Error Detector for GPGPU

Author(s):
Keun Soo Yim
Center for Reliable & High Performance Comput., Univ. of Illinois at Urbana-Champaign, Urbana, IL, USA
Cuong Pham ; Saleheen, M. ; Kalbarczyk, Z. ; Iyer, R.

Published in: Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International
Date of Conference: 16-20 May 2011
Page(s): 287 - 300
ISSN: 1530-2075
E-ISBN: 978-0-7695-4385-7
Print ISBN: 978-1-61284-372-8
INSPEC Accession Number: 12220887
Conference Location: Anchorage, AK
DOI: 10.1109/IPDPS.2011.36
Publisher: IEEE

(This data was quoted from
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=6012845)

Introduction (1)

- Graphics processing units (GPUs) are surfacing as a compelling platform for processing general-purpose HPC programs.
- HPC programs have strong output correctness requirements.
- GPU devices targeting graphics applications usually do not need strong fault-tolerance techniques.
- Regardless of added memory error protection, HPC programs are still vulnerable to certain types of GPU hardware (e.g., ALU, FPU, or register file) due to the irregularity and high operational speed of GPU core login, i.e., constitutes a large portion of the silicon area in the GPU chip.
- Designing a technique to tolerate faults in GPU cores is challenging espacially for HPC GPU programs because of their strong performance and cost requirements.

Introduction (2)

- In this context, software-implemented full duplication (i.e., well-known techniques) can be an effective approach to detect SDC errors in GPU platforms. ※silent data corruption

- Optimizing naïve full duplication has achieved a limited success in GPU programs.

- This paper presents *HAUBERK*, a software technique to derive lightweight error detection and recovery customized for target GPU programs.

Measurement – A. Error sensitivity

- This section evaluates the error sensitivity of HPC and graphics programs executing on GPU and performance characteristics of the used HPC GPU programs.

- Figure 1 shows the error sensitivity of HPC GPU programs, graphics GPU programs, and CPU programs.

- We inject a single-bit error into each variable in benchmark program by using the fault injection tool described in Section VII.

Measurement – A. Error sensitivity

- **Observation 1:** *An SEU (or single-bit error) in the pointer, integer, and FP data leads to an SDC error with 18%, 45%, and 39% average probability, respectively.*

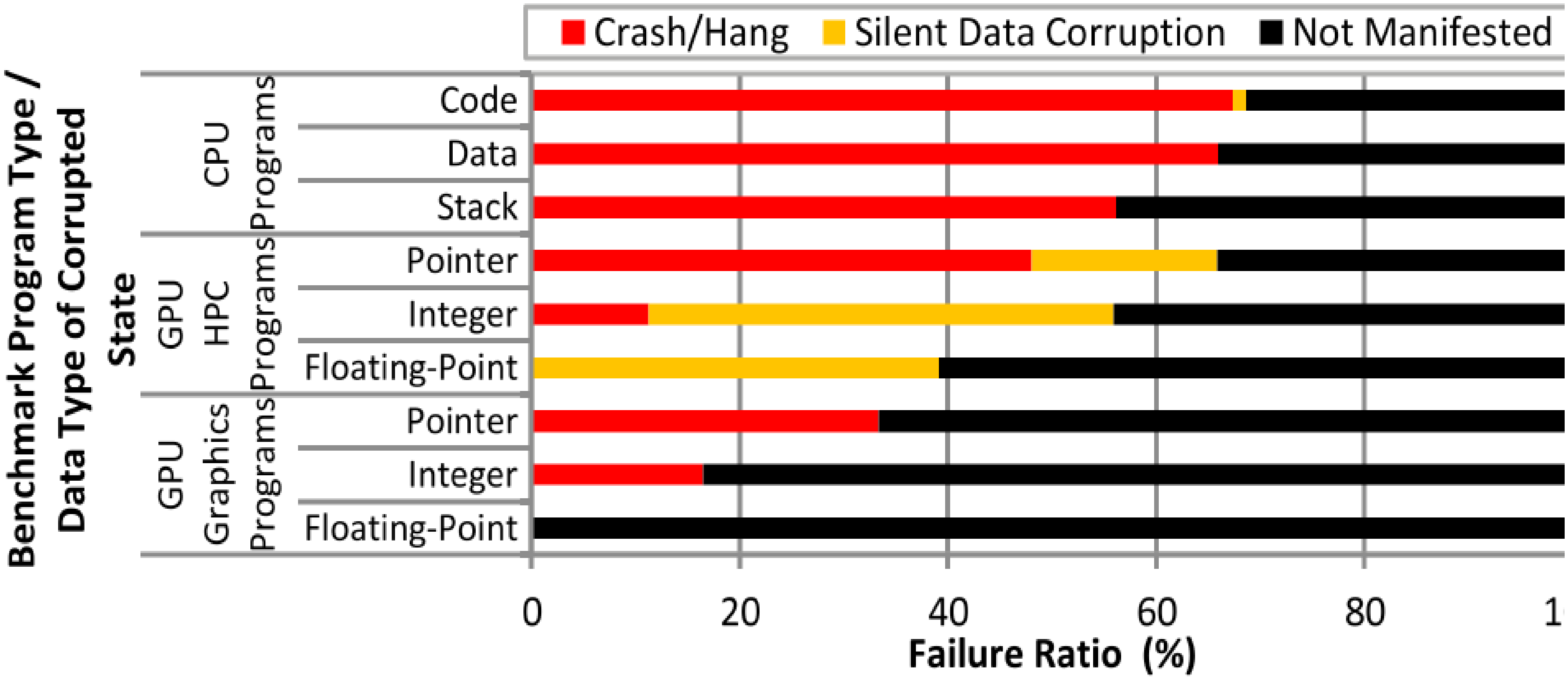


Figure 1. Comparison of average error sensitivity of HPC GPU program, graphics GPU programs, and CPU programs.

Measurement – A. Error sensitivity

- In the benchmark HPC programs, FP data occupy 3-6 orders of magnitudes larger memory space than the pointer and integer data taken together (see Figure 2).

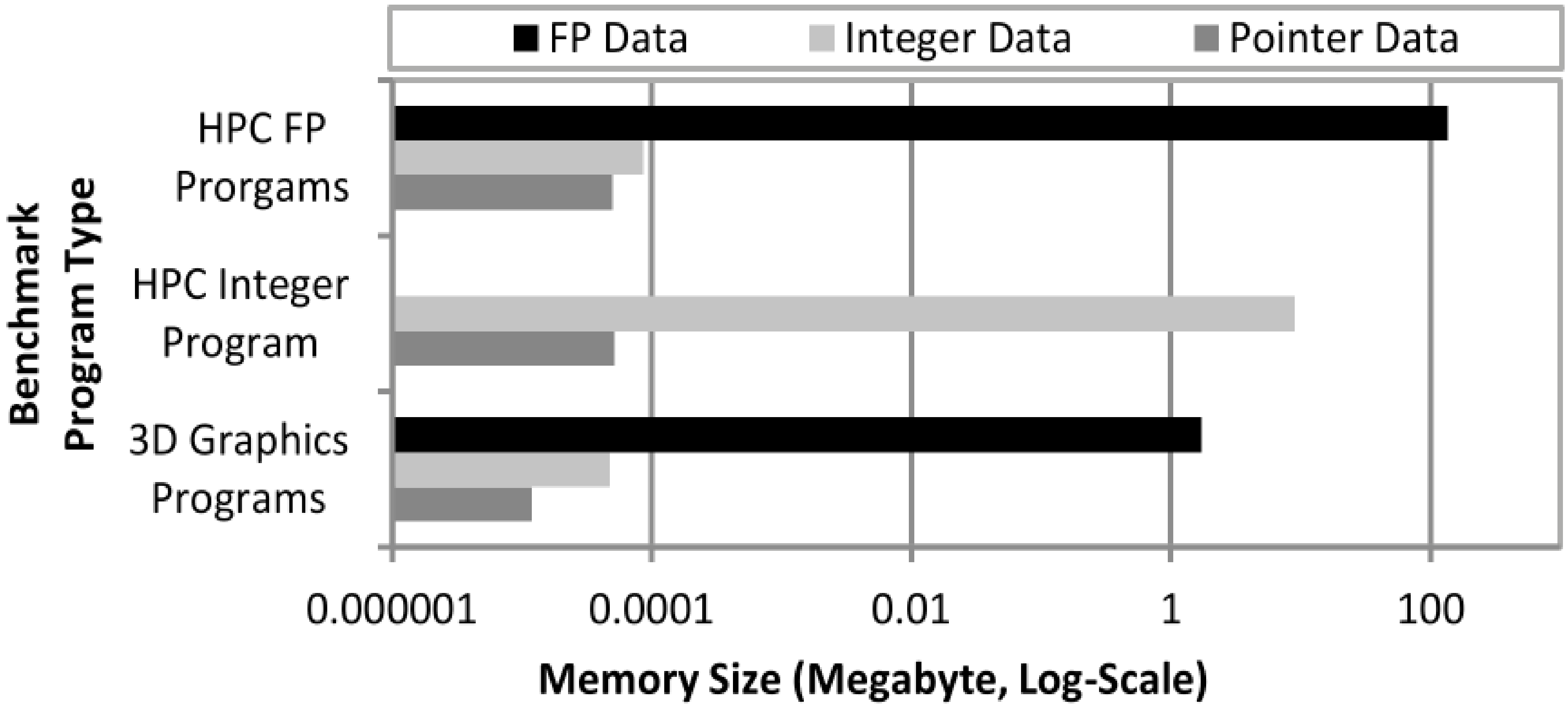
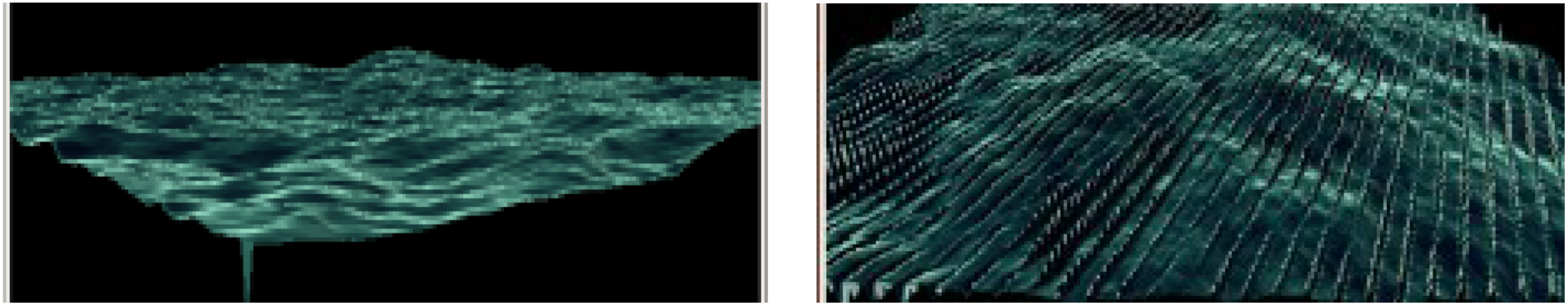


Figure 2. Data type vs. Memory size.

Measurement – A. Error sensitivity

- **Observation 2:** *A fault in an FP variable rarely leads to a GPU program failure, while faults (e.g., 16-33%) in pointer or integer variables are likely to cause program failures.*

- Figure 3(a) shows a video frame of the ocean-flow program that is corrupted by a single-bit fault in its input data stream (a spike in the image is due to the injected fault).



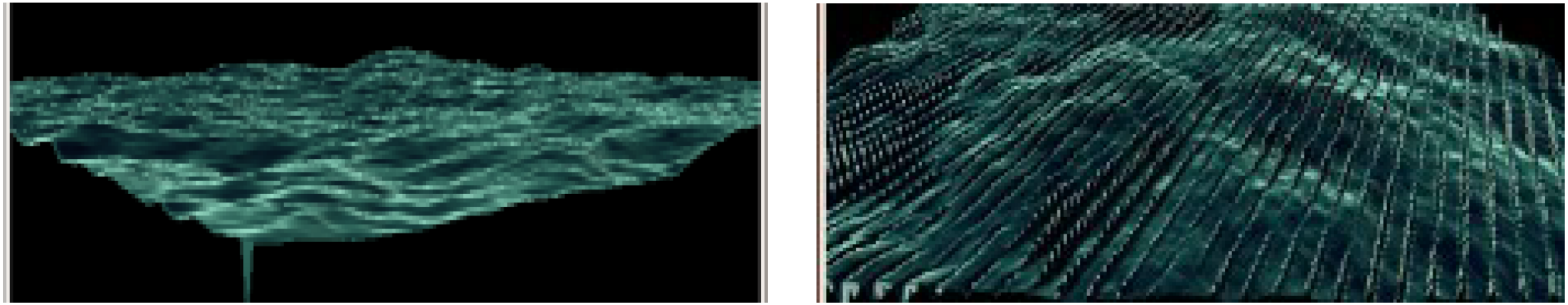
(a) Transient Fault (1 Value Error) (b) Intermittent Fault (10,000 Value Errors)
Figure 3. Impact of faults in a 3D graphics program on GPU.

Measurement – A. Error sensitivity

- **Observation 3:** *3D graphics programs can experience SDC errors when exposed to a longer duration fault in GPU.*

- The impact of an intermittent fault having a long duration time can be significant even in 3D graphics programs.

- In the ocean-flow program, corruptions of 10,000 values form a prominent stripe pattern in the rended frame image (see Figure 3(b)).



(a) Transient Fault (1 Value Error) (b) Intermittent Fault (10,000 Value Errors)

Figure 3. Impact of faults in a 3D graphics program on GPU.

Measurement – B. Performance

- This section characterizes the execution times of loop and non-loop portions of GPU kernels (see Figure 4).

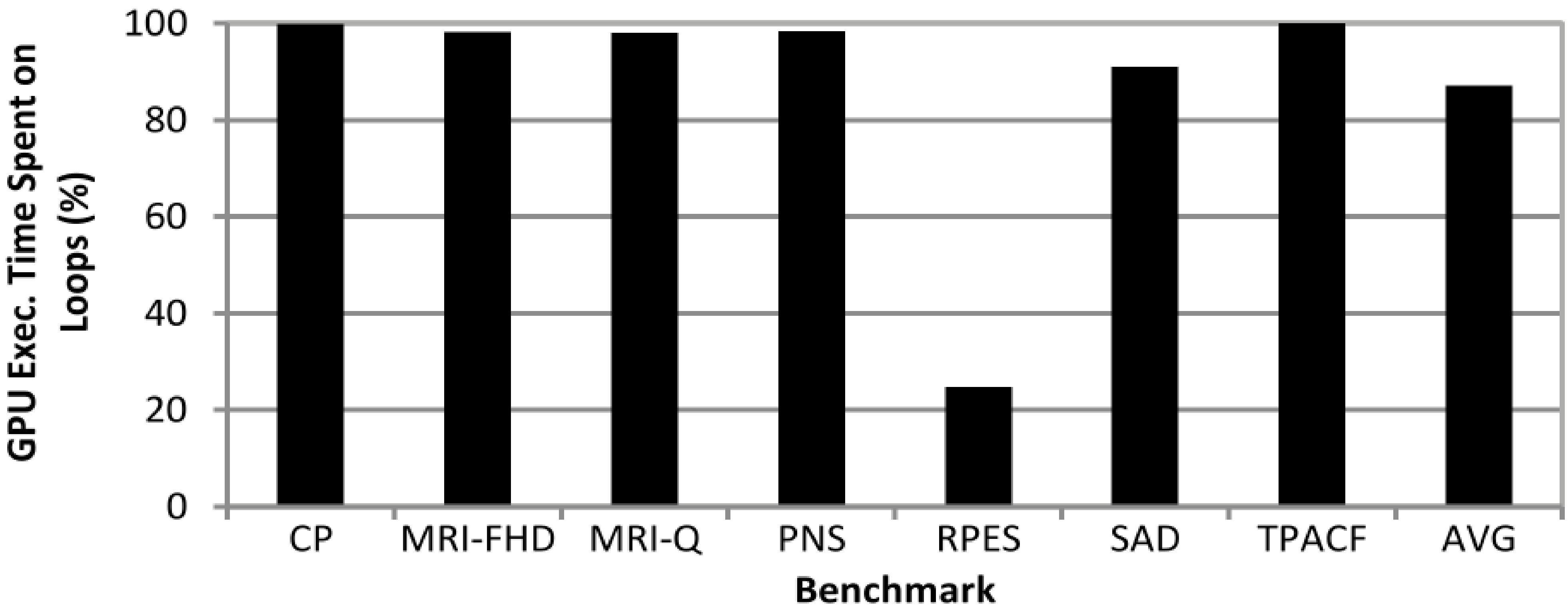


Figure 4. Percent of execution time on loops in HPC GPU programs.

Measurement – B. Performance

- **Observation 4:** *Loops (for, while, and do-while) form a large portion (> 98% in 5 out of 7 programs and 87% on average) of the total execution time spent on GPU.*

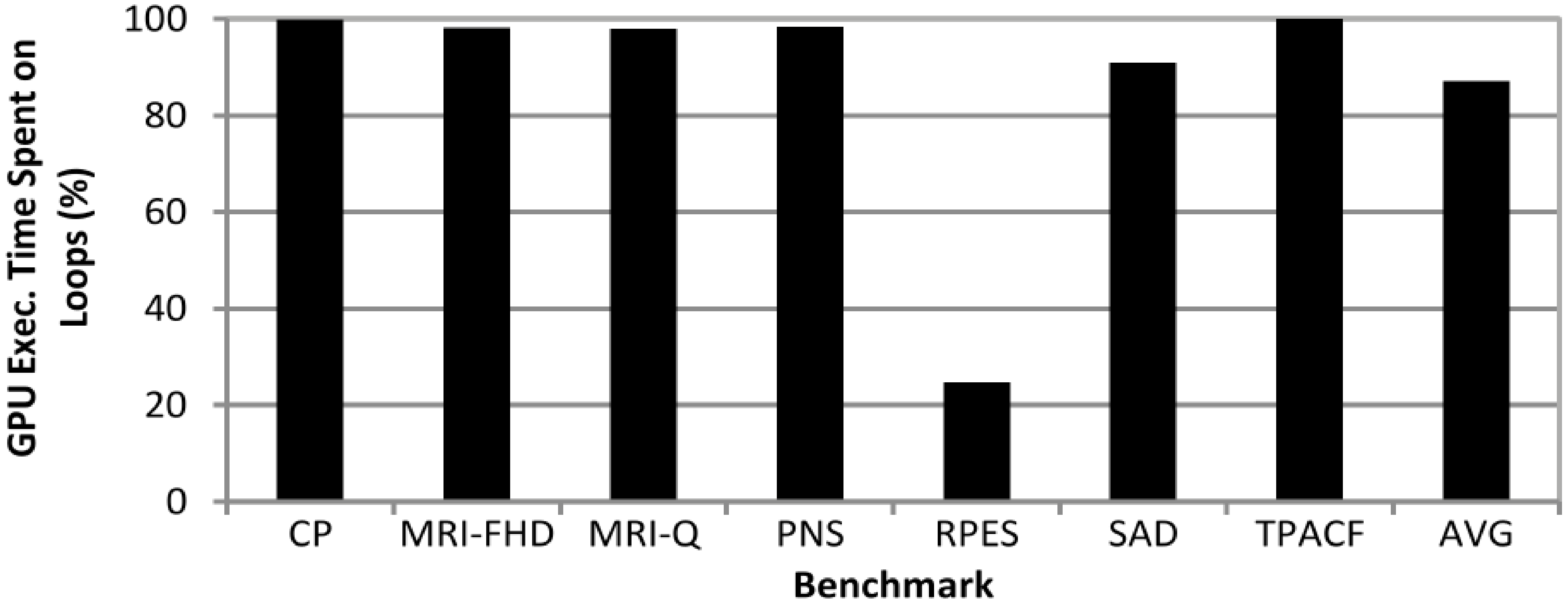


Figure 4. Percent of execution time on loops in HPC GPU programs.

Related work (image)

- This section classifies and analyzes existing error detection techniques potentially applicable in the context of this study (see Figure 5).

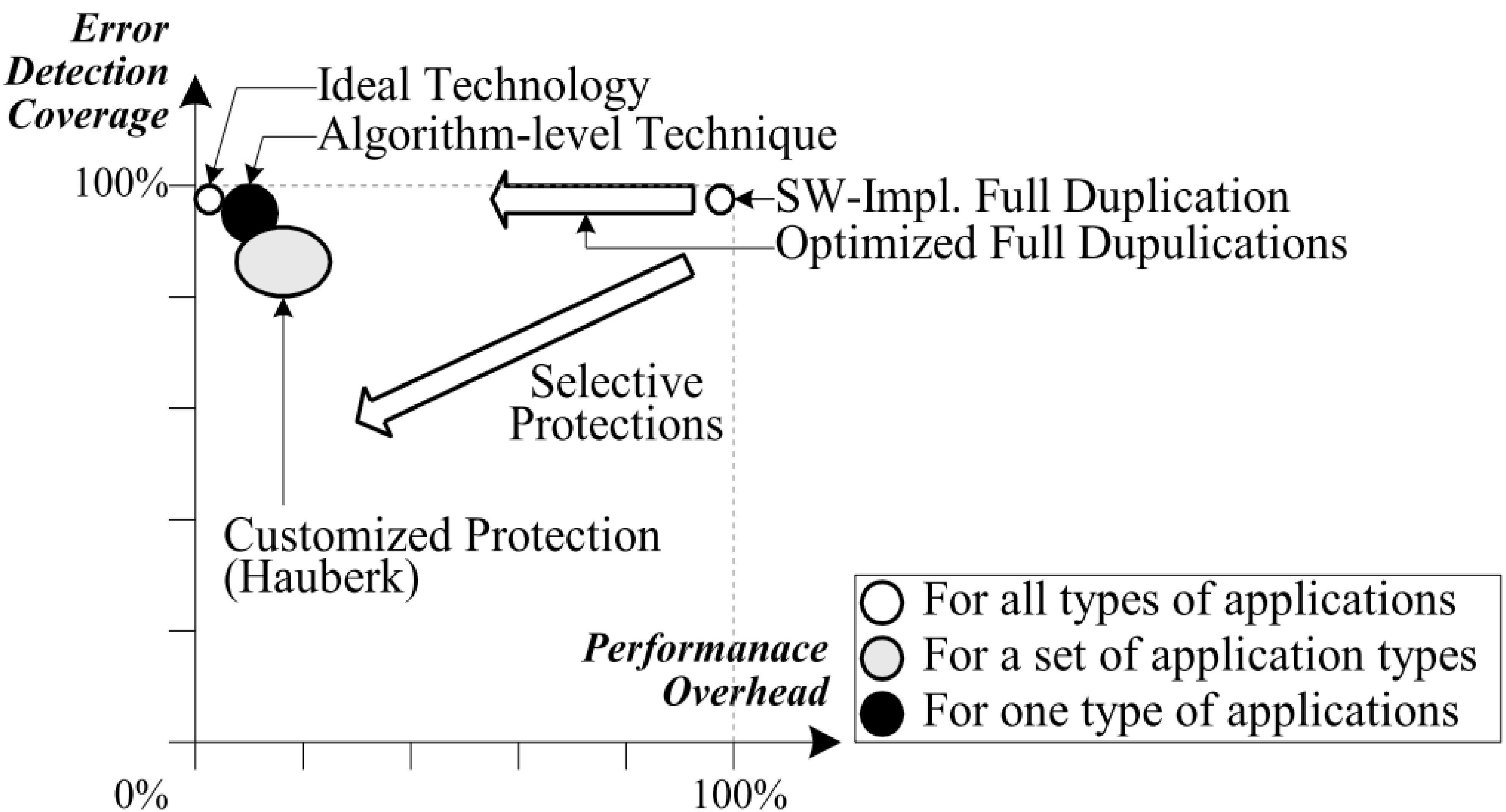


Figure 5. Spectrum of various types of error detection techniques.

Related work

- The Design goal is to find a high-coverage ditector without compromising performance.

(i) Naïve full duplication

- high SDC error detection ratio, almost doubles the execution time

(ii) Optimized full duplication

- utilize idle hardware resource, not highly effective for GPU program

(iii) Selective protection

- selectively protects parts of the program state

(a) Fault injection

- most effective if the size of program is small

(b) Static compiler analysis

- can quickly select protection target state

(C) Dynamic program analysis

- derives and selects likely program invariants by profiling and monitors selected invariants at runtime

(iv) Algorithm-level techniques

- Error detection techniques designed and optimized for a particular type of algorithm or program are usually highly efficient

GPU HAUBERK – A. Design principles

Principle 1:
HAUBERK customizes error detectors by using profiling information of common HPC GPU programs in order to minimize the impact on performance.

Principle 2:
HAUBERK selectively protects the program state where errors in other states are likely to propagate.

Principle 3:
HAUBERK places error detectors by considering the recoverability of errors.

GPU HAUBERK – A. Design principles

HAUBERK defers placements of error detectors as long as possible by taling advantage of inherent hardware-enforces error isolation between GPU and CPU.

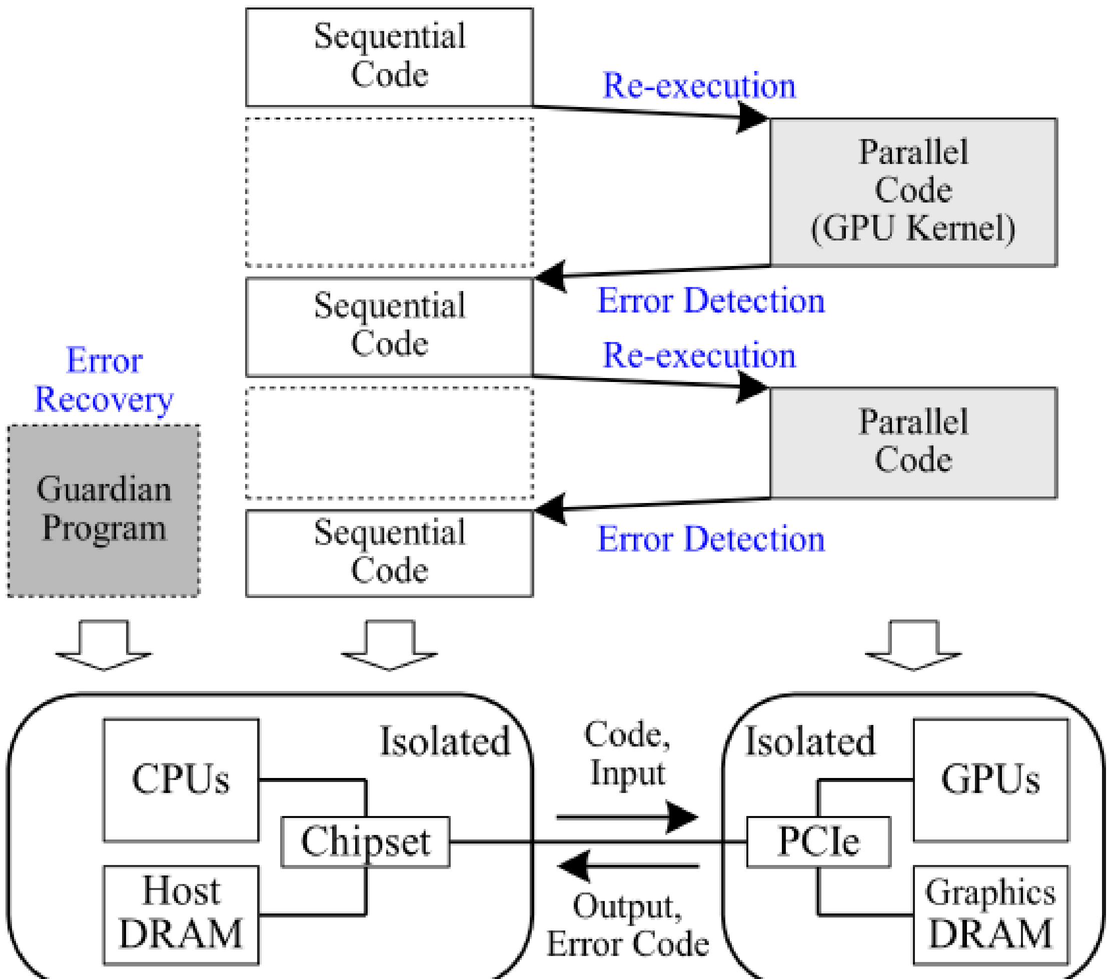


Figure 6. Isolation execution and deferred checking model of *Hauberk*.

GPU HAUBERK – B. Framework

Figure 7 depicts a compile flow of the HAUBERK framework.

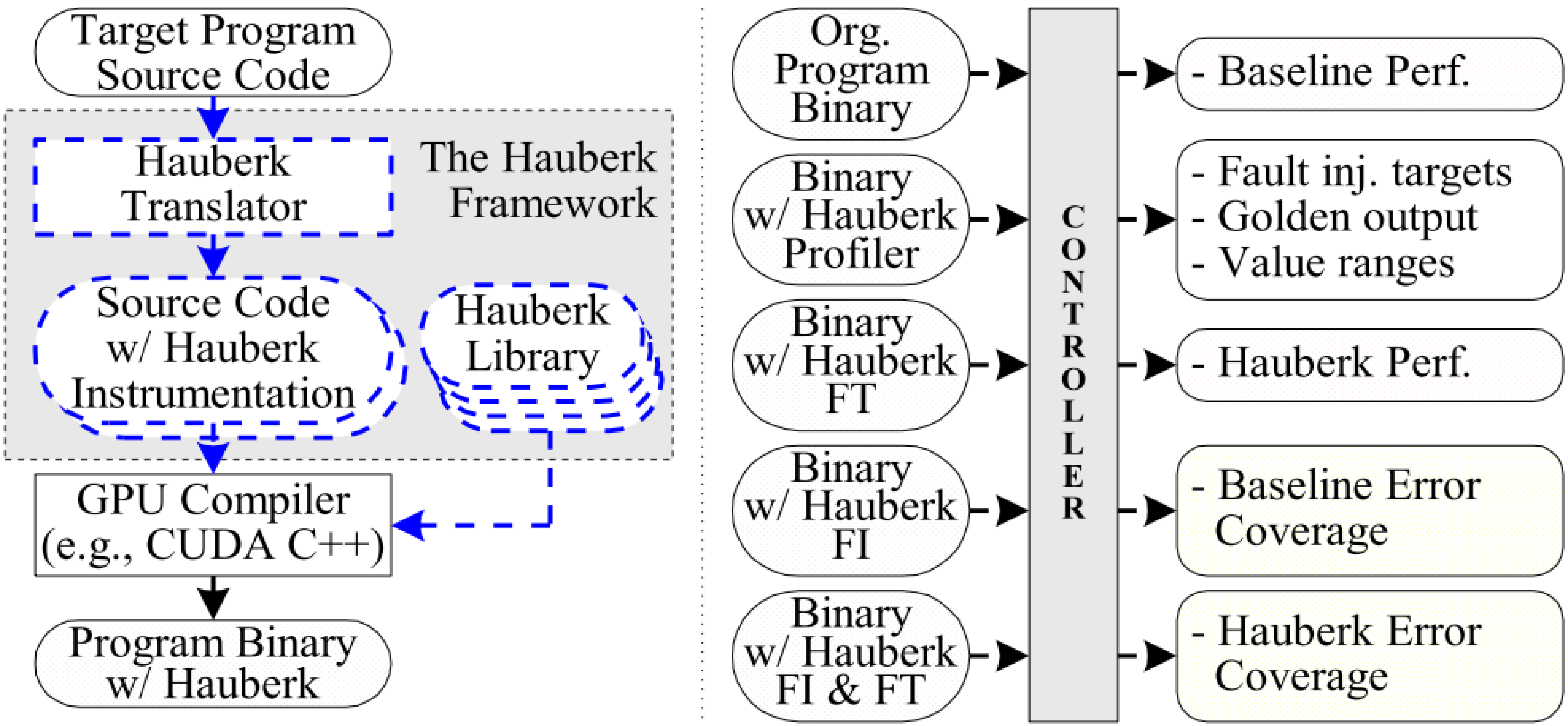


Figure 7. Compilation and evaluation flows in the *HAUBERK* framework.

GPU HAUBERK – B. Framework

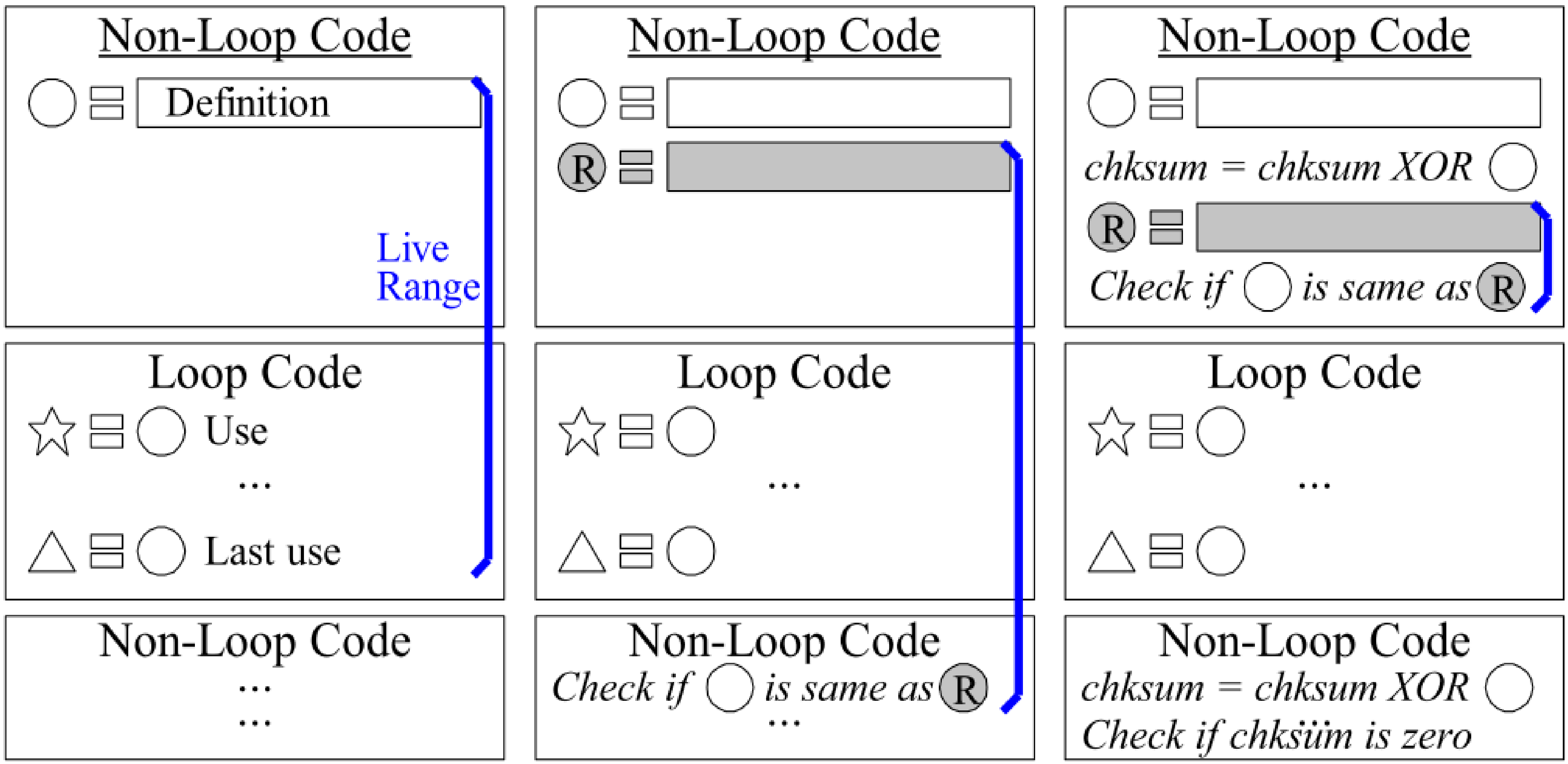
- Places where *HAUBERK* translator adds or mutates source codes are summarized in Table I.

TABLE I. DESCRIPTIONS OF INSTRUMENTATIONS USED FOR *HAUBERK*.

<table>
<tr><th>Location Lib.</th><th>FI (Section VII)</th><th>Profiler (Section V.B)</th><th>FT (Section V.A., V.B., VI)</th></tr>
<tr><td>[CPU] Top of the main file</td><td colspan="3">Includes a header file for HAUBERK libraries</td></tr>
<tr><td rowspan="2">[CPU] Entry of main()</td><td colspan="3">Initializes the control block</td></tr>
<tr><td>The control block is for the location, time, and type of fault injection target</td><td>The control block is for profiled value ranges and execution counts</td><td>The control block is for value ranges, detection results, and outliers</td></tr>
<tr><td>[CPU] Exit of main()</td><td>Stores fault activation result to a file</td><td>Stores profiling results to a file</td><td>Stores updated value ranges to a file</td></tr>
<tr><td rowspan="2">[CPU] Before launching GPU kernel</td><td colspan="3">Copies the control block from CPU to GPU</td></tr>
<tr><td colspan="2">-</td><td>Notifies this to guardian process and calls a checkpoint library (option)</td></tr>
<tr><td rowspan="2">[CPU] After GPU kernel launch</td><td colspan="3">Waits until the kernel completion and copies the control block back from GPU to CPU</td></tr>
<tr><td colspan="2">-</td><td>Calls an error recovery function</td></tr>
<tr><td>[CPU] GPU kernel function</td><td colspan="3">Adds a pointer variable for the control block as a function parameter in GPU kernel function prototype and its caller(s)</td></tr>
<tr><td rowspan="2">[GPU] After definition of virtual variable in GPU non-loop</td><td colspan="2">Calls a library function with an identifier, pointer, type, and used hardware components of variable defined in previous statement</td><td rowspan="2">Updates a checksum variable, duplicates the definition, and checks original and duplicated variables</td></tr>
<tr><td>To inject a fault into a defined variable at a designated time of execution</td><td>To count execution count per variable</td></tr>
<tr><td>[GPU] After def. of virtual variable in GPU loop</td><td>Same as “After definition of virtual variable in GPU non-loop” field</td><td colspan="2">Adds two addition statements for each protected target virtual variable (one for target variable and the other for counter) and merges the counters if possible</td></tr>
<tr><td rowspan="2">[GPU] Before loop in GPU kernel</td><td rowspan="2">-</td><td colspan="2">Defines accumulator and counter variables for each protected loop variable</td></tr>
<tr><td>-</td><td>Updates the checksum var. if needed</td></tr>
<tr><td>[GPU] After loop in GPU kernel</td><td>-</td><td>Profiles value ranges of accumulated variables divided by their counter</td><td>Checks accumulated variable value ranges and updates the checksum var.</td></tr>
<tr><td>[GPU] Exit of GPU kernel</td><td>-</td><td>-</td><td>Checks the checksum variable</td></tr>
</table>

Error detection – A. For non-loop code

- *HAUBERK* duplicates the definition of virtual variable and immediately checks the original and duplicated variables (see Figure 8(c)).



(a) Original code (b) Naïve duplication (c) *HAUBERK-NL*

Figure 8. Duplication techniques for non-loop codes where statements marked as gray symbols or italic texts are added for error detection.

Error detection – B. For loop code

- We present value-accumulation-based range checking for loop codes. Derivation of this error detector has four steps.

(i)Select target variable for protection
Among all virtual variables defined inside a target loop, we first select self-accumulating virtual variables.

(ii)Generate value accumulator code
The placed error detector accumulates the data value of each protected virtual variable in every loop iteration.

(iii)Generate accumulation counter code
An addition statement is added to count the number of accumulation operations for each accumulator variable.

(iv)Generate error checking code
An error checking routine is added right after the loop code.

Error detection – B. For loop code

- Figure 9 exemplifies a data-flow graph of a loop in a GPU kernel that is computing a coulombic potential function.

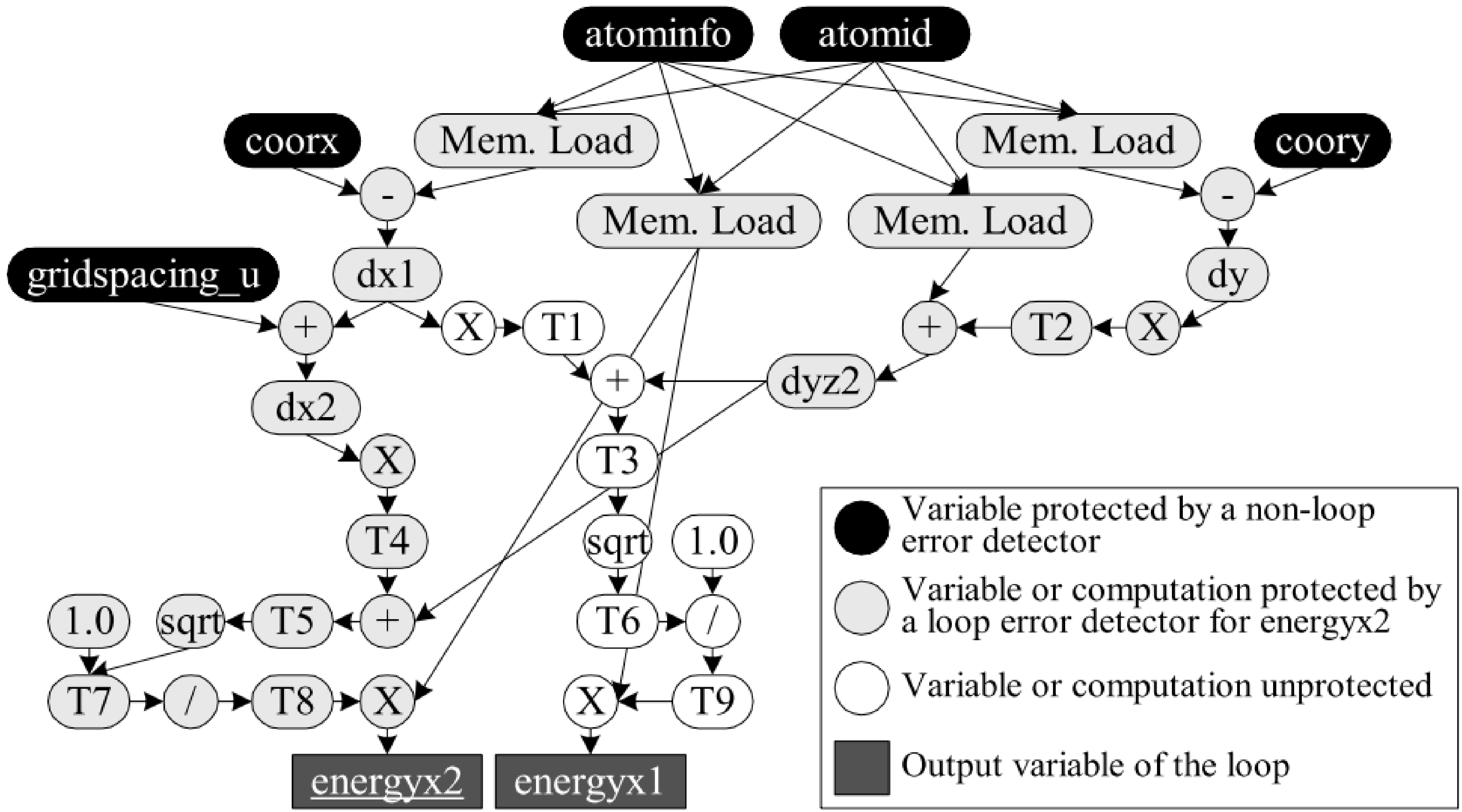


Figure 9. Dataflow graph of a loop in a coulombic potential GPU kernel.

Error detection – B. For loop code

- A strong correlation is observed in values stored in or computed for a same program variable in many HPC GPU programs.
- Figure 10 shows the value distribution of integer and FP variables in an HPC GPU program (MRI-Q).

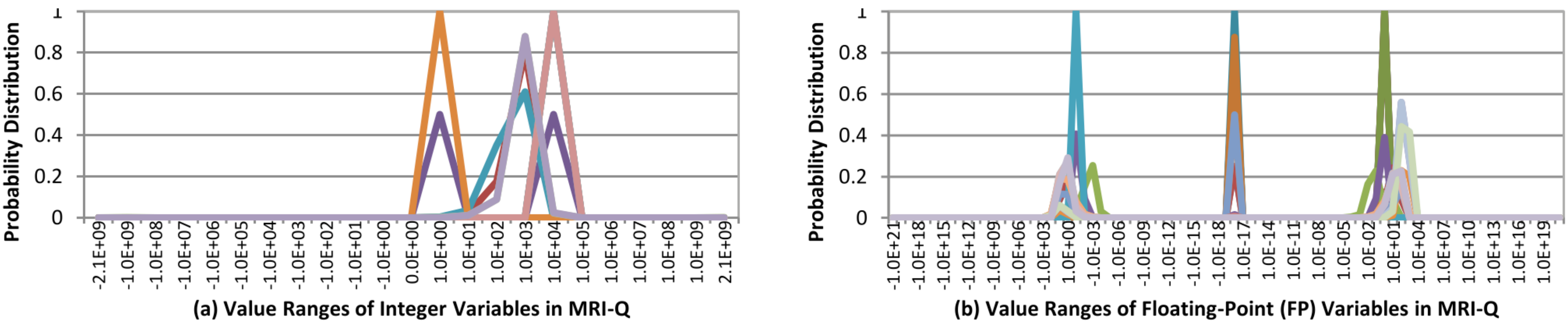


Figure 10. Value range distributions of integer (a) and FP (b) variables in the *MRI-Q* program executing on a GPU device.

Error recovery

- This section describes retry-based error recovery in *HAUBERK*, which can diagnose and tolerate errors in GPU.

(i)Guadian Program

- A gurdian program is used as a parent process of program instrumented by using the *HAUBERK* framework (see Figure 6).

(ii)Diagnosis of False Alarms

- *HAUBERK* loop error detectors may result in both false negatives.

(a)False alarm

- If the reexecution also raises an SDC alerm and its output is identical to the original output, these two are likely to be false alarms (i.e., false positive).

(b)SDC error due to transient or short intermittent fault

- If the reexecution terminates normally and does not raise an SDC alarm, we assume the alarm raised in the first execution is due to transient or intermittent faule (i.e., removed before the second execution).

(c)SDC error due to long intermittent or parmanent fault

- If the reexecution also raises an SDC alarm but its output is not identical to the original execution output, we execute a GPU program that is specifically designed to produce multiple sets of output data by examining variout parts of GPU hardware.

(iii)Configuring Loop Error Detector

- This false alarm diagnosis can calculate the false positive ratio.

Error recovery

- If the failure is repeated twice in the same GPU kernel using the same input data (see Figure 11), the guadian process runs a program to diagnose intermittent or parmanent faults in GPU device.

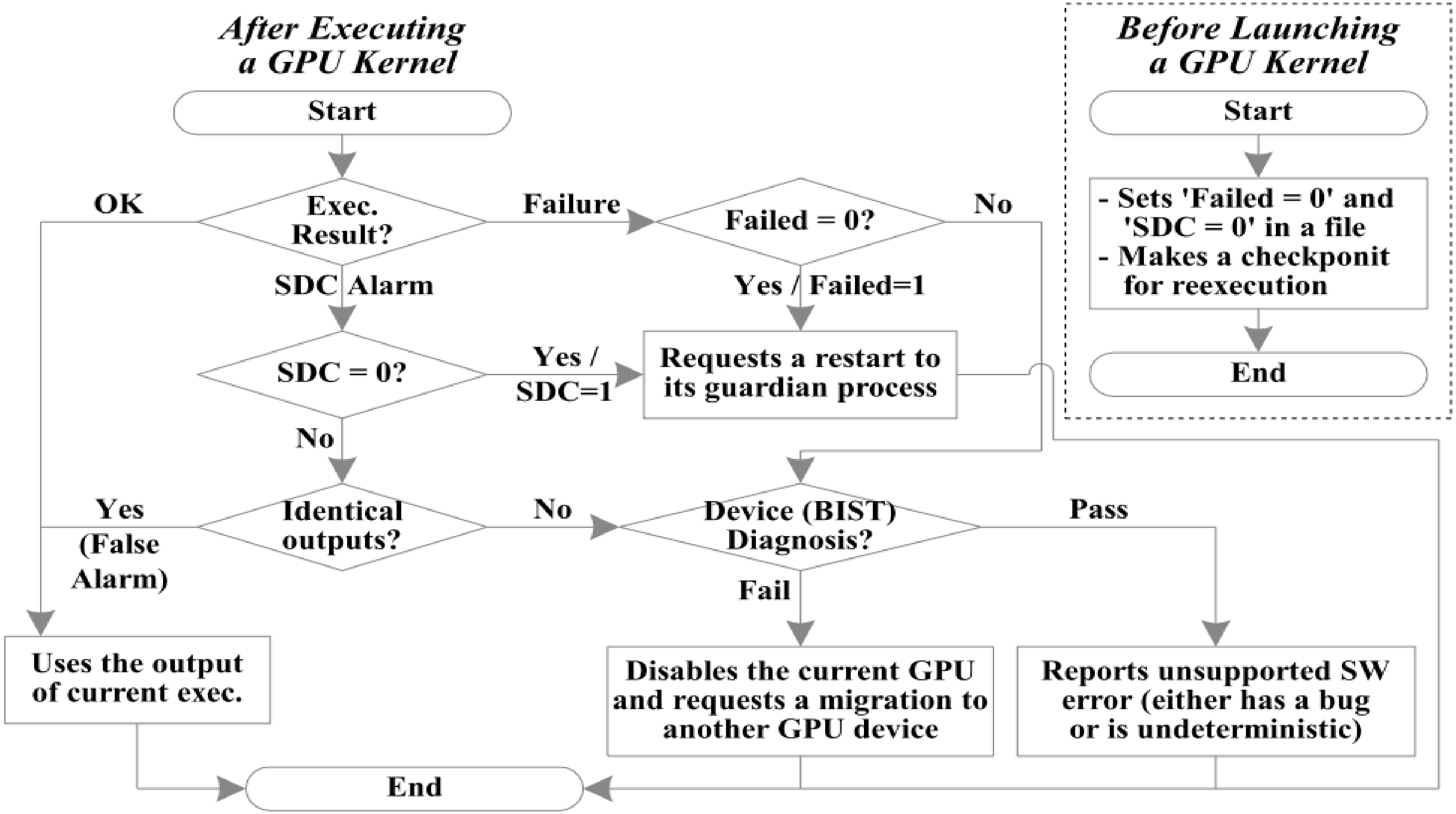


Figure 11. Error diagnosis and tolerance algorithm.

Dependability evaluation framework

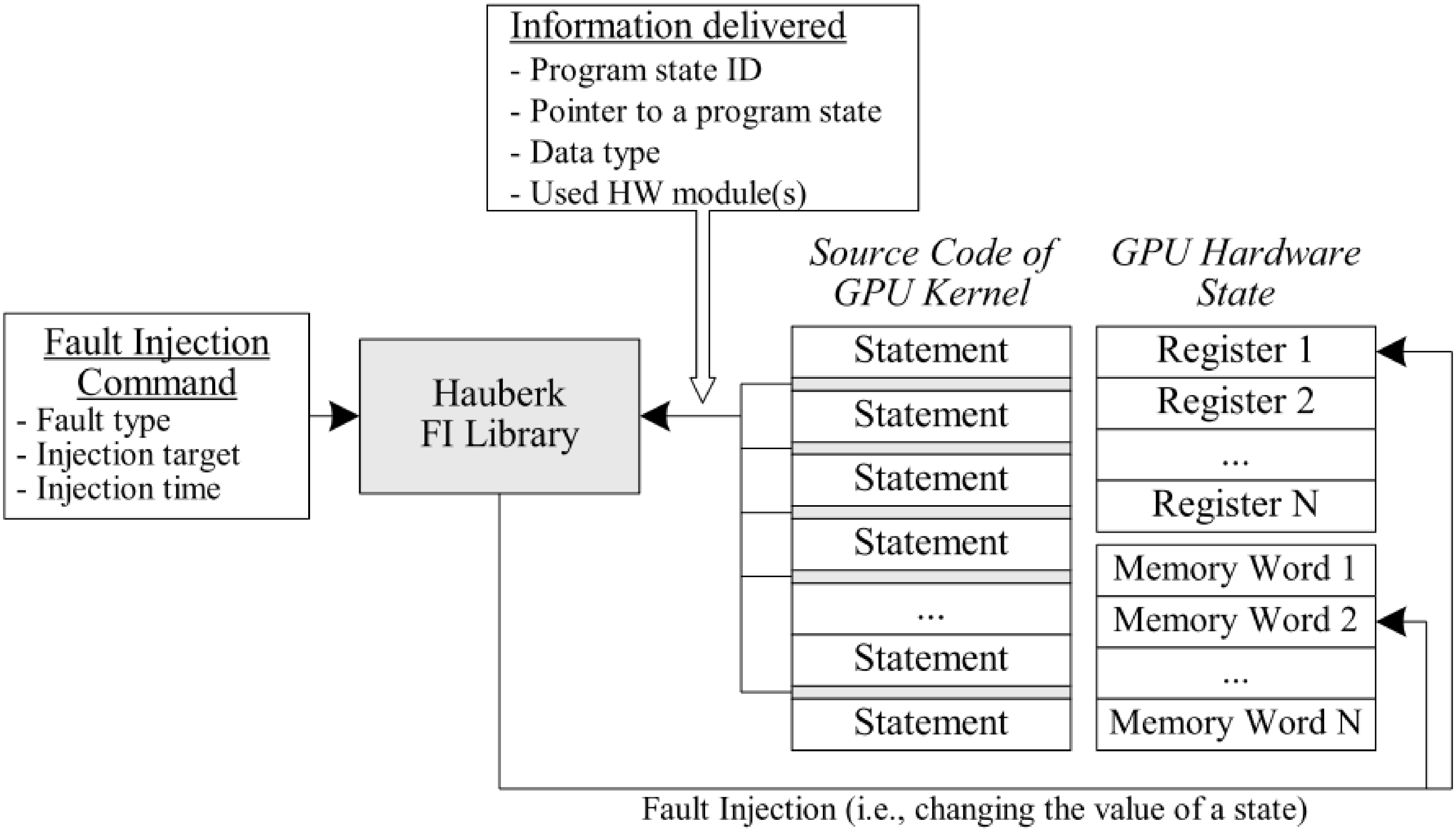


Figure 12. A GPU kernel with *HAUBERK* fault injection codes.

Experimental results

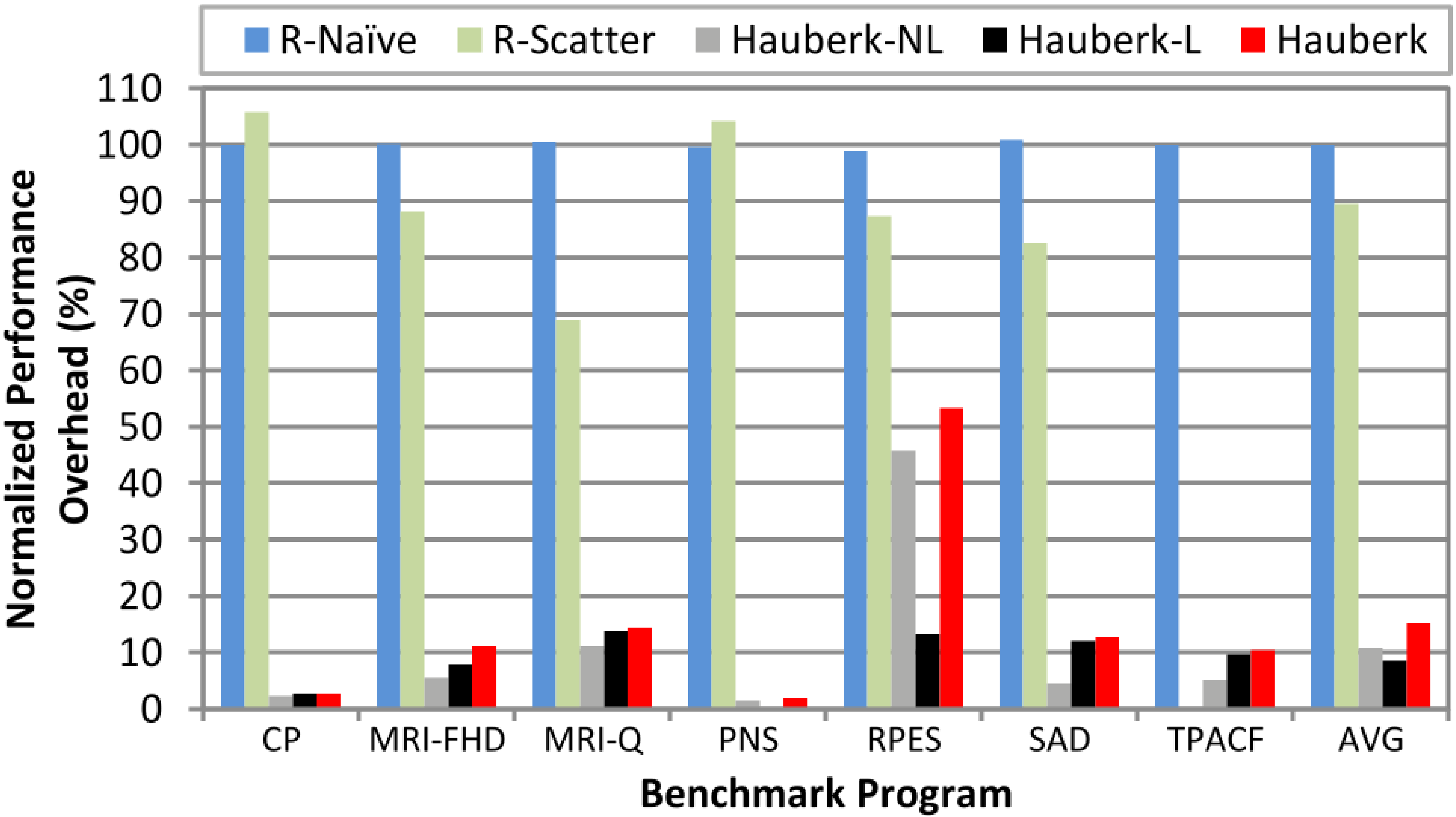


Figure 13. Performance overhead.

Experimental results

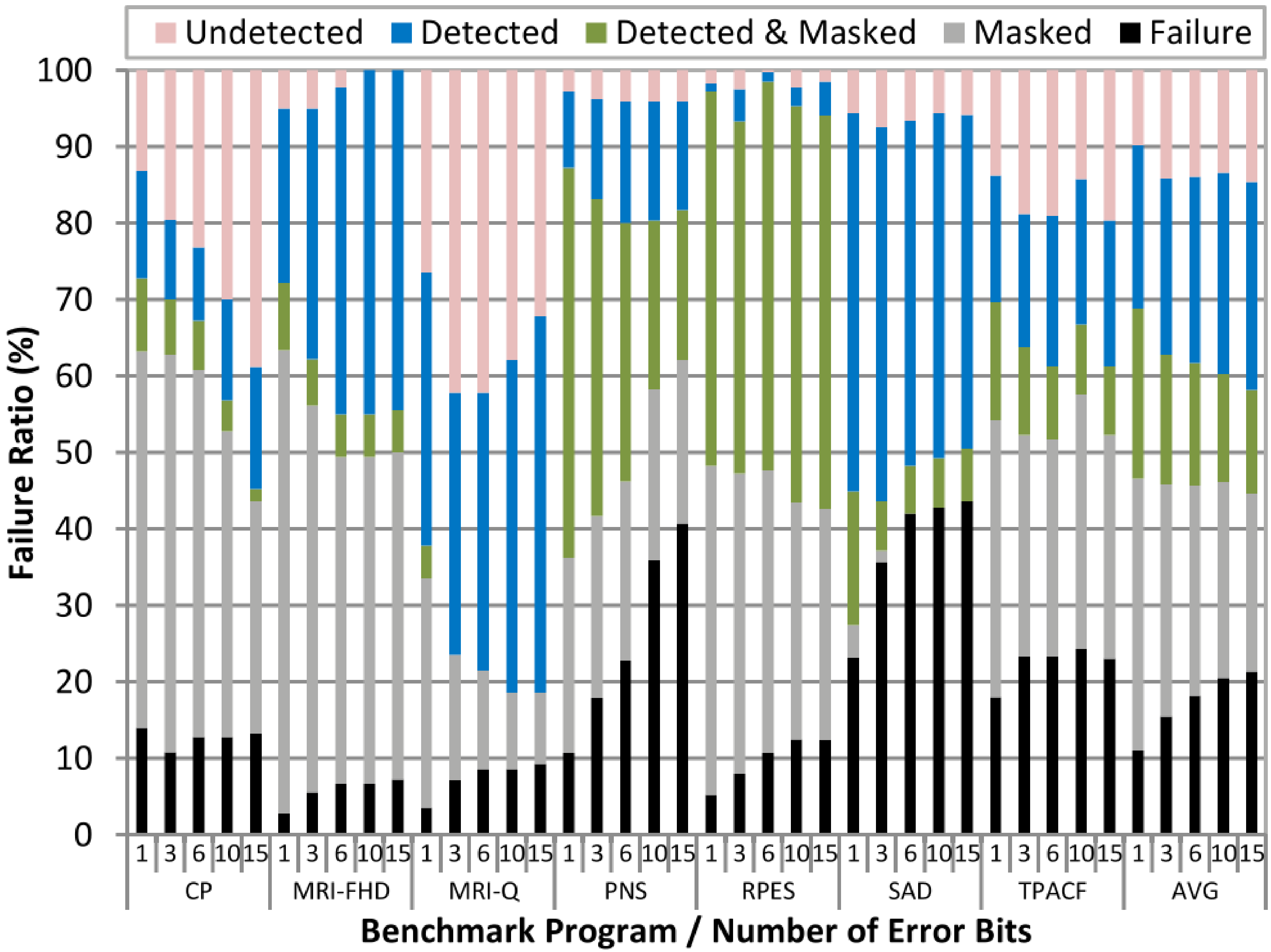


Figure 14. Error detection coverage of *HAUBERK*.

Experimental results

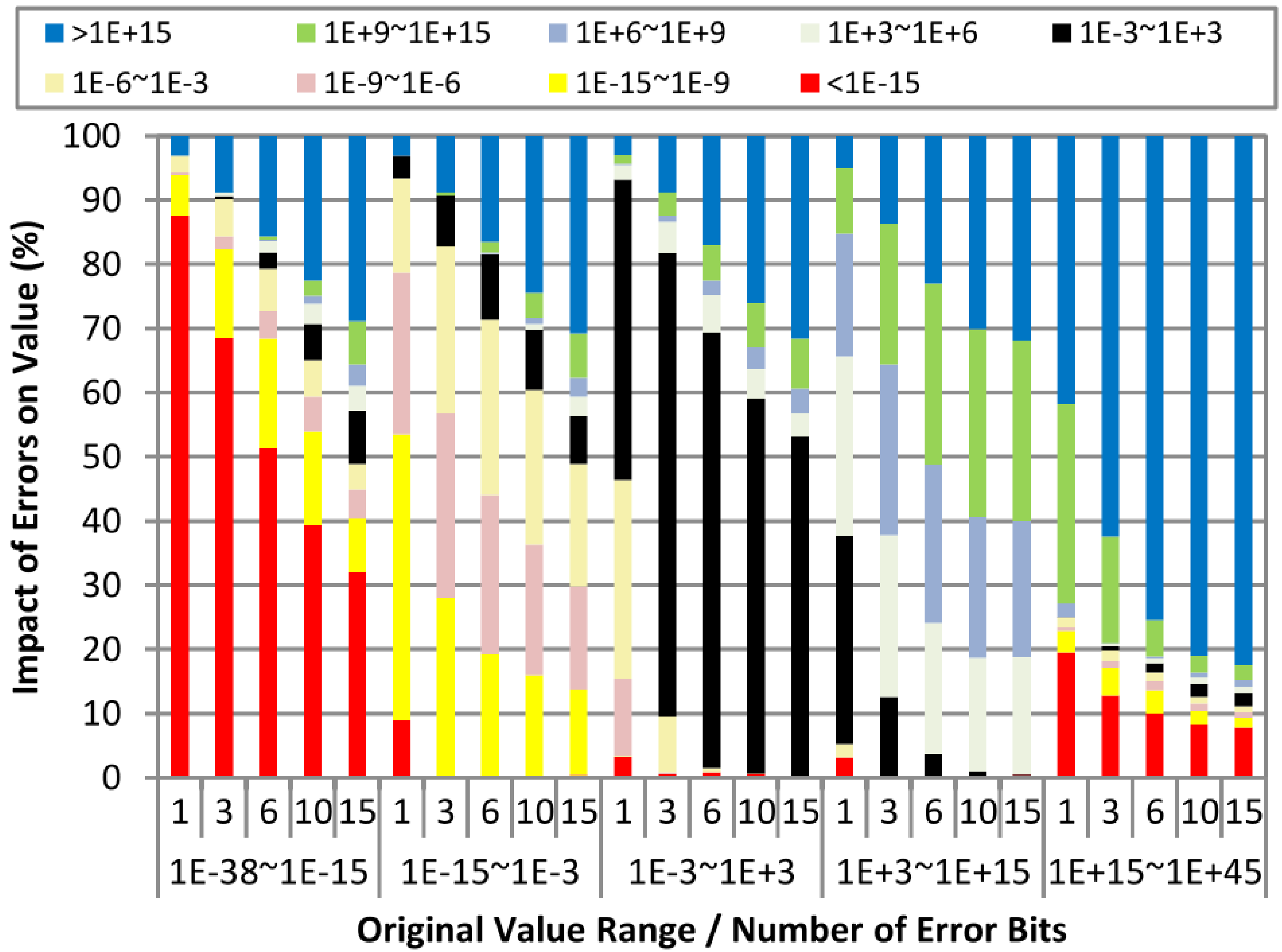


Figure 15. Changes in the magnitude of values after experiencing a fault depending on the orginal value range (of FP data) and error bit count.

Experimental results

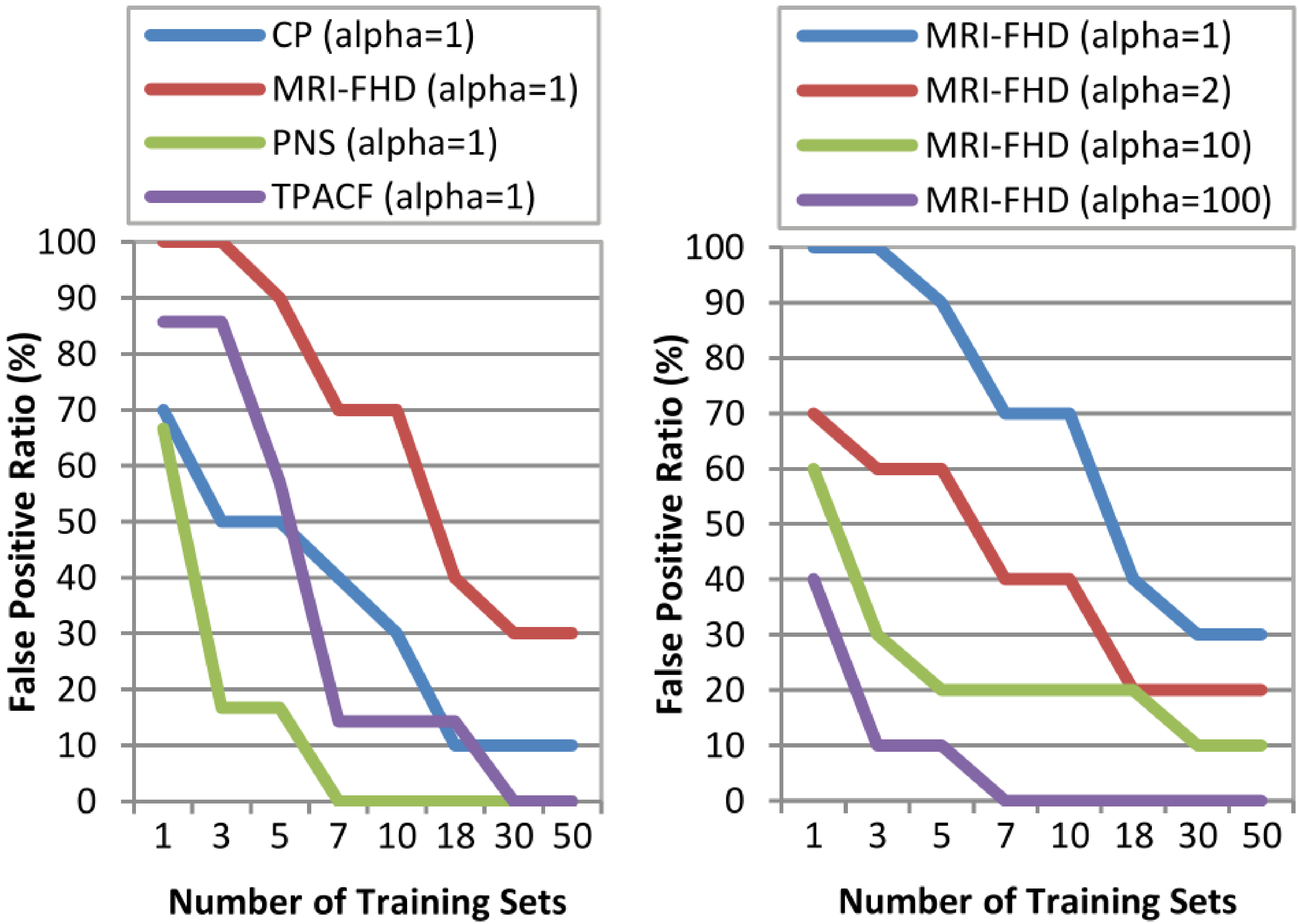


Figure 16. False positive ratio vs. Training count.

Conclusion

- This paper analyzed reliability problems in GPGPU platforms, focusing particularly on the design of efficient low-cost detection and recovery mechanisms for handling SDC (silent data corruption) errors.

- In order to tolerate SDC errors, customized error detection techniques are strategically placed in the source code of target GPU program so as to minimize performance impact and error propagation, and maximize recoverability.

- HAUBERK offers a high error detection coverage (~87%) with a small performance overhead (~15%).