

Smart-Hopping: Highly Efficient ISA-Level Fault Injection on Real Hardware

Horst Schirmeier, Lars Rademacher, and Olaf Spinczyk

Department of Computer Science 12, Technische Universität Dortmund, Germany
e-mail: {horst.schirmeier, lars.rademacher, olaf.spinczyk}@tu-dortmund.de

Abstract—Fault-injection experiments on the instruction-set architecture level are commonly used to analyze embedded software’s susceptibility to hardware faults, typically involving a vast number of experiments with systematically varying fault locations and times. Determinism and high performance are the predominant requirements on fault-injection platforms. Injecting faults into a real embedded hardware platform instead of a simulator is favorable for both workload execution speed and result accuracy. The most performance-critical part of such a fault-injection platform is the “fast forward” operation, which executes the target machine code without faults until the exact dynamic instruction is reached at which the execution must be stopped to inject the next fault. Unfortunately, most embedded CPUs do not support this operation efficiently.

In this paper we present an approach that speeds up fast-forwarding significantly for most workloads with minimal requirements on hardware support. Based on a previously recorded instruction trace – which is needed for systematic fault-injection experiment planning anyways – we use standard debugging hardware to advance to a chosen point in program execution with a minimal number of steps. We evaluate our FAIL* tool platform with two *MiBench* benchmark categories, and improve experiment throughput by up to several magnitudes compared to similar fault-injection tools in the field.

I. INTRODUCTION

Recent technology roadmaps [1], [2], [3] suggest that future hardware designs for embedded systems will exhibit an increasing rate of soft errors, trading reliability for smaller die sizes, lower supply voltage, and reduced production costs. This trend creates new challenges for embedded software development, which must application-specifically place error detection [4] and recovery mechanisms [5] (EDM/ERMs) that do not diminish all gains from these new hardware designs. In the future, critical parts of the software stack of reliable embedded systems must be hardened against hardware faults, while the remaining unprotected components economize resource consumption by occasionally tolerating incorrect results.

Fault injection (FI) on the instruction-set architecture (ISA) level has been the standard analysis and EDM/ERM evaluation technique in the software fault-tolerance community for at least two decades [6]. A FI campaign involves a vast number of experiments with different injected faults, deterministically repeating target software runs with injections at systematically chosen points in time and space [7], [8], [9]. In contrast to Monte-Carlo approaches [10], [11], deterministic FI is based on a previously¹ recorded instruction and memory-access trace that enables various techniques to dramatically reduce the number of experiments to be conducted [12], [9].

Injecting faults into a real embedded hardware platform instead of a hardware simulator is favorable for both workload execution speed and result accuracy [13], [14], [8]. Resorting to an FPGA implementation [15] is not possible in most cases due to the lack of accurate hardware models. The most performance-critical part of such a fault-injection platform is the “fast forward” operation, which deterministically executes the target machine code without faults until a specific dynamic instruction is reached. At this point in the instruction stream, which is also present in the previously recorded trace and identified by its position in that trace, the execution must be stopped to inject the next fault. Unfortunately, most embedded CPUs do not support this operation efficiently: Without further information, the target platform must single-step instructions until the specified instruction is reached. The widely used JTAG debugging interface [16], especially when controlled through low-cost USB debugger hardware, incurs significant round-trip times between each step. This results in extremely long experiment runtimes, dominated by the forwarding operation, and consequently a low FI experiment throughput endangering sufficient fault-space coverage.

After recapitulating the state of the art (Section II) we present an approach that *speeds up fast-forwarding significantly* for most workloads with minimal requirements on hardware support (Section III). Exploiting the information from a previously recorded instruction trace – which is needed for systematic fault-injection experiment planning anyways – we use standard debugging hardware (breakpoints, memory watchpoints) to advance to a chosen point in program execution with a minimal number of steps. We evaluate our FAIL* tool platform [17] with *MiBench*’s [18] *automotive* and *network* benchmark categories, and show an improvement in experiment throughput by up to several magnitudes compared to similar fault-injection tools in the field (Section IV). We discuss the evaluation results and other approaches in Section V, and conclude in Section VI.

II. STATE OF THE ART

The problem of fast-forwarding to a specific dynamic instruction of a deterministically executing program has been studied not only in the fault-injection domain, but is known especially in the context of debugging under the concept of “deterministic replay”. For example, King et al. [19] leverage checkpoints and Intel’s hardware performance counters to deterministically re-execute parts of the target to create an illusion of “time travel” backwards and forwards through the execution of a program. More recently, Patil et al. [20] extended a similar approach to multi-core replay without any specific hardware support requirements, but at the expense of slow execution in a virtual-machine environment with just-in-time

¹The recording takes place during a fault-free, so-called “golden” run.

```

1 for (i = 0; i < 64; ++i) {
2     for (j = 0; j < 64; ++j) {
3         sum += array[i+j];
4     } }

```

```

8: b    30 <func+0x30> ; (outer loop entry)
c: add r12, r0, r3 ; i+j
10: ldrb r12, [r12, r2] ; array[...]
14: add r2, r2, #1 ; ++j
18: cmp r2, #64 ; j < 64
1c: add r1, r1, r12 ; sum += ...
20: bne c <func+0xc> ; (inner loop back edge)
24: add r3, r3, #1 ; ++i
28: cmp r3, #64 ; i < 64
2c: beq 38 <func+0x38> ; (outer loop exit)
30: mov r2, #0 ; j = 0
34: b c <func+0xc> ; (outer loop back edge)

```

Figure 1: Example ARM machine code (initialization omitted) and the C code snippet it was compiled from: The highlighted *static* instruction at address 0x10 appears $64^2 = 4096$ times in the *dynamic* instruction stream.

compilation. In his *Jockey* [21] tool, Saito uses binary patching techniques and user-defined program annotations impossible to use for advancing to arbitrary instructions. The existing approaches are too slow to run on embedded hardware, modify the target software to a point where the *observer effect* becomes relevant, or depend on special hardware features such as cycle-accurate performance counters. In the following, we will outline the fast-forwarding approaches currently predominant in the fault-injection domain.

A. Single-stepping

The most straightforward approach to advance to the N^{th} dynamic instruction is *single-stepping* N times. This approach is feasible for very short target programs, but cannot be used for reasonably sized workloads, as fast-forwarding to the dynamic instruction N grows linearly: $t_{\text{ff}} = t_{\text{exec}} + N \times t_{\text{singlestep}}$. (t_{exec} is a negligible amount of time to actually execute the stepped instructions). For example, fast-forwarding to the highlighted instruction in the second iteration of the inner and the second iteration of the outer loop in Figure 1 ($i = 1, j = 1; N = 400$) takes about 28.0 s – for a single experiment out of potentially thousands – on contemporary ARM Cortex A9 development hardware connected through a USB JTAG debugger.

B. Simple-hopping with Hardware Breakpoints

A more sophisticated approach is to use hardware breakpoints to go forward in the instruction stream by multiple dynamic instructions at once. Fidalgo et al. [14] set a breakpoint to advance to the *first* instance of a static instruction (this would correspond to the instruction 0x10 in Figure 1, for the first time executed at position #5 in Figure 2) but do not discuss the general case of reaching an arbitrary dynamic instance. A dynamic instruction with the property that it represents the *first* occurrence of a static instruction can therefore be reached with a single breakpoint hop; unfortunately this property only holds for a infinitesimal fraction of the dynamic instructions in real-world programs with loops and branches.

Rebaudengo [22], Folkesson [13], Skarin [23], [8], and Hannius [24] also break on the static target instruction, but

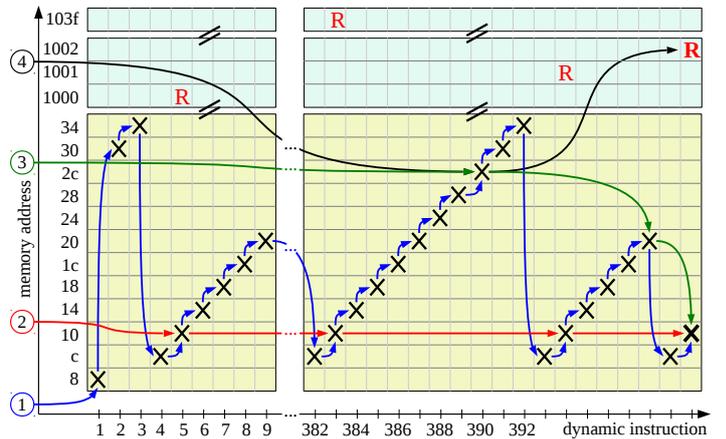


Figure 2: Pre-recorded program trace with dynamic instructions (eXecuted memory addresses), memory accesses (Reads and writes), and different fast-forwarding methods: (1) *Single-stepping*, (2) *simple-hopping*, and (3)/(4) *smart-hopping* with/without memory watchpoint usage.

continue running the program after each breakpoint trigger until the dynamic target instruction is reached. Figure 2 illustrates this *simple-hopping* procedure with the jump chain labeled (2): A breakpoint on the static instruction at address 0x10 triggers $N = 66$ times until the dynamic instruction #400 is reached. The fast-forwarding time can be calculated as $t_{\text{ff}} = t_{\text{exec}} + O \times t_{\text{BP}}$, O being the number of breakpoint triggers. Assuming that a breakpoint trigger and a subsequent *continue* takes about the same time as a single-step, and t_{exec} is negligible, the time needed for fast-forwarding of this example is reduced by a factor of 6 compared to single-stepping. This reduces the actual time overhead to about 4.6 s on the mentioned ARM platform.

C. Checkpointing

The usage of checkpoints (e.g., Berrojo et al. [7]), which represent loadable intermediate states at a priori chosen (and usually equidistant) dynamic instructions N_1, N_2, \dots , allows fast-forwarding to specific points in the execution trace. A checkpoint load costs a non-negligible amount of time, but more importantly *creating and storing* checkpoints consumes both time and storage space. Therefore, the total number of checkpoints in use is limited, making this technique orthogonal to fine-grained fast-forwarding techniques such as the aforementioned single-stepping or simple-hopping to navigate to the desired target instruction after the checkpoint load.

III. SMART-HOPPING: CHOOSING THE SHORTEST PATH

Unlike deterministic replay (cf. Section II), deterministic fault injection has the advantage of a complete instruction and memory access trace that is usually recorded for experiment planning purposes. Extending on the simple-hopping approach explained in the previous section, we propose a fast-forwarding method that exploits more trace knowledge and forwards to a specified dynamic instruction with a *minimal* number of breakpoint hops.

Instead of (simple-)hopping from one dynamic instance of the target instruction to the next, the basic idea of our *smart-hopping* approach is to iteratively set a breakpoint on the static instruction that *will trigger farthest in the future* in

Algorithm 1 The *smart-hopping* algorithm generates the optimal jump sequence for all possible trace positions, reusing the solution sequence for the previous position.

```

# Possible memory ACCESS TYPES in the trace:
type : enum ACESSTYPE { execute, read, write }
# Multiple ACCESSes may occur at one trace pos. (point in time):
type : class ACCESS { INT address, ACESSTYPE type }
# Always holds the optimal hop sequence to current pos. in trace:
var cur_solution : list of tuples (INT trace_pos, ACCESS a)
# Remember most recent occurrence of specific ACCESS in trace:
var access_last_seen : map ACCESS → INT trace_pos
# All memory ACCESSes at current trace position:
var cur_trace_events : list of type ACCESS

cur_trace_pos ← 0
while not at trace end do
  read trace events at cur_trace_pos into cur_trace_events
  if any ACCESS a ∈ cur_trace_events not in access_last_seen
  then
    clear cur_solution, add (cur_trace_pos, a) to it
  else
    new_hop ←
      {x|x ∈ cur_trace_events ∧ access_last_seen[x] minimal }
    last_seen ← access_last_seen[new_hop]
    while length of cur_solution is > 1 do
      (posa, a) ← rightmost entry in cur_solution
      (posa-1, a-1) ← 2nd to rightmost entry in cur_solution
      if last_seen ≤ posa-1 then
        remove (posa, a) from cur_solution
      else
        break
      end if
    end while
    add (cur_trace_pos, new_hop) to cur_solution
  end if
  print cur_solution
  for all ACCESS x ∈ cur_trace_events do
    access_last_seen[x] ← cur_trace_pos
  end for
  cur_trace_pos ← cur_trace_pos + 1
end while

```

the instruction trace. In the example (Figures 1 and 2), this reduces the number of breakpoint hops and debugger round-trips to three (variant (3) in Figure 2): One at static instruction 0x2c (dynamic instruction #390), completely skipping the first iteration of the outer loop construct, one at 0x20 (#398), and one at the target instruction 0x10 (#400). On the aforementioned ARM hardware, fast-forwarding time is reduced to about 0.21 s.

A. Embracing Memory Watchpoints

Although choosing an optimal breakpoint path significantly improves the example of reaching target instruction #400, this method degenerates when a late iteration of an instruction-wise homogeneous loop (i.e., no varying branch decisions in the loop body) contains the dynamic target instruction. For example, in a scenario trying to reach instruction 0x10 in the last iteration of the inner loop ($i = 0, j = 63$ in Figure 1), the optimal breakpoint path contains 64 hops – yielding a fast-forwarding time identical to using simple-hopping.

Fortunately, standard on-chip debugging hardware offers another feature exploitable for fast-forwarding purposes: memory-access watchpoints. Similar to breakpoints, watchpoints trigger when an instruction accesses memory, and can usually even

differentiate between read and write accesses. In the aforementioned pathologic example of the last iteration of the inner loop, the memory load instruction (0x10) accesses array element 63 exactly at the dynamic target instruction #400. This access can therefore be used as a watchpoint hop to directly forward to the desired target instruction, reducing the path length to one even for this scenario. The scenario in Figure 2 (4) is further reduced to two hops.

Algorithm 1 outlines our smart-hopping heuristic that generates the optimal breakpoint/watchpoint hop sequence for *all* possible trace positions. It iteratively consumes dynamic instructions (including the memory accesses they perform) from a sequentially read input trace, and reuses the (initially empty) solution hop sequence for the previous position. Although a solution sequence can grow to arbitrary lengths for input traces with adverse event patterns (such as homogeneous loops with extremely many iterations and no memory accesses, cf. Section IV), the algorithm can be implemented with modest memory footprint even for very large input traces.

B. Optimality

For homogeneous hop costs (i.e., identical timing properties for breakpoints and watchpoints at arbitrary addresses), the greedy smart-hopping heuristic is optimal due to an intuitive argument: *Not* picking the farthest-possible hop from the set of all reachable destinations cannot be better than picking the farthest, because the set of new potential hops reachable from a closer hop is a subset of those reachable from the farthest-possible one. More formally, it can be shown that always greedily choosing the farthest hop shortcuts suboptimal detours in Dijkstra’s algorithm [25], leveraging its property of locally optimal partial solutions.

IV. IMPLEMENTATION & EVALUATION

In order to evaluate the effectiveness of our fast-forwarding heuristic, we chose MiBench’s [18] *automotive* and *network* benchmarks as a source for instruction and memory access traces. They were compiled with ARM-GCC 4.6.1 (`-march=armv5te`) and run in the gem5 simulator [26] in ARM syscall emulation mode to generate the traces for both the “small” and “large” parameter sets. The left quarter of Figure 3 shows the benchmarks and their dynamic instruction and memory access counts for the two parameter sets.

We implemented the simple-hopping (for comparison with the state of the art) and smart-hopping heuristics as a C++ tool as part of our FAIL* fault-injection tool suite. The tool takes an instruction and memory access trace as an input, and calculates fast-forwarding breakpoint (and watchpoint, if enabled) hop lists for *every* dynamic instruction in the trace. Additionally, the forwarding costs are calculated. Note that an actual fault-injection campaign would not run an experiment for *every* dynamic instruction but carefully select a subset; nevertheless the calculated maximum and average costs are a good indication on how much faster the campaign gets with the different heuristics.

Our embedded evaluation platform is a PandaBoard ES with an ARM Cortex-A9 attached to a Linux PC through a USB 2.0 JTAG debugger (Flyswatter2). Measurements showed that the basic costs on this platform are not homogeneous as outlined in the previous sections, but differ by a factor of two in some cases: While a single-step and a normal hop between

MiBench benchmark properties on ARMv5				Simple-hopping			Smart-hopping (BP only)			Smart-hopping (BP+WP)			
Benchmark	Input	Dyn. Instr.	Mem. Accesses	max	avg	σ	max	avg	σ	max	avg	σ	
<i>automotive</i>	BASICMATH	large	3.37×10^9	1.05×10^9	1.76×10^7	3.62×10^6	4.00×10^6	12,889	215.2	1,246.2	8,367	114.5	702.9
		small	2.66×10^8	5.00×10^7	2.92×10^6	7.21×10^5	7.55×10^5	267	10.5	12.3	177	4.7	7.3
	BITCOUNT	large	6.26×10^8	4.28×10^7	6.75×10^7	2.30×10^7	2.08×10^7	301,372	150,200.6	86,956.6	301,372	124,995.0	92,278.2
		small	4.18×10^7	2.86×10^6	4.50×10^6	1.53×10^6	1.38×10^6	20,122	10,051.3	5,807.5	20,122	8,374.9	6,170.8
	QSORT	large	4.59×10^8	1.50×10^8	3.14×10^6	6.78×10^5	7.06×10^5	59,562	3,203.7	4,838.8	5	3.1	0.7
		small	1.78×10^7	7.12×10^6	4.63×10^5	6.73×10^4	8.85×10^4	6,318	708.9	879.8	8	2.7	0.7
SUSAN	large	3.92×10^8	1.52×10^8	9.95×10^6	2.54×10^6	2.93×10^6	28,849	116.0	689.5	40	2.9	1.0	
	small	2.42×10^7	8.94×10^6	6.50×10^5	1.74×10^5	1.93×10^5	1,883	55.3	57.5	40	2.8	0.8	
<i>network</i>	DIJKSTRA	large	2.03×10^8	6.94×10^7	1.85×10^7	6.76×10^6	5.11×10^6	534	179.8	96.3	9	4.6	1.0
		small	4.59×10^7	1.60×10^7	3.67×10^6	1.18×10^6	1.04×10^6	534	153.3	103.2	7	3.8	1.1
	PATRICIA	large	5.81×10^8	2.25×10^8	5.10×10^6	4.47×10^5	6.13×10^5	70	9.4	2.7	8	2.7	0.7
		small	9.45×10^7	3.65×10^7	7.84×10^5	6.84×10^4	9.30×10^4	70	7.9	2.5	8	2.6	0.7

Figure 3: Benchmarks from MiBench’s *automotive* and *network* benchmark categories (with both *small* and *large* input data sets) with dynamic instruction counts on ARMv5 and memory access counts, fed through the *simple-hopping* heuristic, *smart-hopping* with only breakpoints, and *smart-hopping* with both break- and watchpoints. The results include maximum and average costs for fast-forwarding to *every dynamic instruction* in the respective benchmark.

breakpoints and watchpoints take about 67 ms, hopping from a breakpoint to another breakpoint on the same static instruction (similar with watchpoints) takes 135 ms.² We subsequently implemented a slightly modified variant of Algorithm 1 that avoids the expensive hops (by taking the second-to-farthest hop in these cases if possible) while still yielding optimal results when assuming a factor of exactly 2. Note that the benchmarks were not actually run on the target hardware itself because only the fast-forwarding overhead – which can be calculated from the heuristics’ output and the measured basic costs for our platform – is relevant for this article.

The smart-hopping tool with enabled watchpoints processes about 1.5–2.3 million dynamic instructions per second (depending on the particular trace) on an Intel Xeon X5470 machine at 3.33 GHz, and its memory consumption peaks at 11.3 MiB when analyzing the “large” BITCOUNT trace. The table in Figure 3 shows the maximum and average fast-forwarding costs (multiply by 67 ms for the actual fast-forwarding time) for the chosen MiBench benchmarks:

- The commonly used simple-hopping heuristic is clearly not usable for programs of this size: even for the “small” input of QSORT it takes an average of 75 minutes to fast-forward to a chosen dynamic instruction, and the other benchmarks are worse by a factor of up to 342.
- Using the smart-hopping heuristic without watchpoints already improves the situation significantly: most benchmarks can on average be fast-forwarded within a matter of seconds, although the maximum forwarding costs are still relatively high, especially for the “large” inputs. The average fast-forwarding time improvement ranges between a factor of 94.9 for the “small” QSORT benchmark and 68,945 for the “small” BASICMATH. However, the BITCOUNT benchmark’s costs stand out: on average it still takes 168 minutes to fast-forward the “large” variant, the maximum takes even twice that long.

²The debugger implements this case by deleting the breakpoint, stepping a single instruction, and setting the breakpoint again. The single-step is not necessary if the new breakpoint is at a different address.

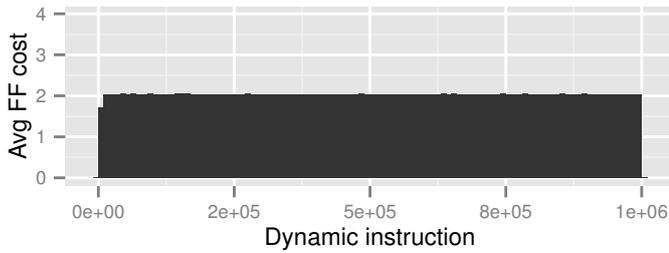
- Allowing smart-hopping to additionally use watchpoints as hop targets again dramatically improves fast-forwarding times for almost all benchmarks: the average fast-forward is below 320 ms for all but BITCOUNT and the “large” BASICMATH, and also the maximum costs are within reasonable bounds. The improvement compared to the simple-hopping heuristic now ranges between 183.2 for the “small” BITCOUNT to 1.48 million for the “large” DIJKSTRA benchmark. Unfortunately, the problematic BITCOUNT benchmark still has extremely high forwarding times; its maximum costs did not improve at all with the help of watchpoints.

Figure 4 shows the average fast-forwarding costs for the first 10^6 dynamic instructions of the DIJKSTRA and the problematic BITCOUNT benchmarks. While DIJKSTRA’s trace constantly offers trace features that ease fast-forwarding (static instructions that are rarely executed, or specific memory accesses), the costs for forwarding in the BITCOUNT benchmark linearly increase with the dynamic instruction count. A closer investigation of the C code behind BITCOUNT reveals that its nested loop constructs repeat almost identical operations 1.125 million times (for the “large” setup); only the use of a pseudorandom number generator seems to introduce some variation. We believe that this pathologic setting is relatively unrealistic for our application domain, as fault-injection experiments will not yield any new and interesting results after analyzing the first few of the million outer loop iterations.

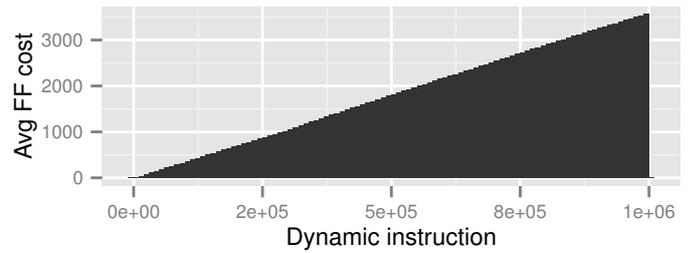
V. DISCUSSION

The evaluation showed that our smart-hopping heuristic improves fast-forwarding times massively for most target programs, as the round-trip time bottleneck between host PC and target platform dominates the total experiment time. For example, even for the shortest benchmark (QSORT “small”) the fault-injection campaign throughput³ improves from 19.2 experiments per day to 1781 without or 73,165 with watchpoints using smart-hopping on a single PandaBoard. This results in a

³Assuming that setup/reset of the target platform and actual program runtime t_{exec} add to 1 s per experiment.



(a) DIJKSTRA



(b) BITCOUNT

Figure 4: Average fast-forwarding costs (bin size: 10^4 dynamic instructions) for the first 10^6 dynamic instructions of the “small” DIJKSTRA and BITCOUNT benchmarks.

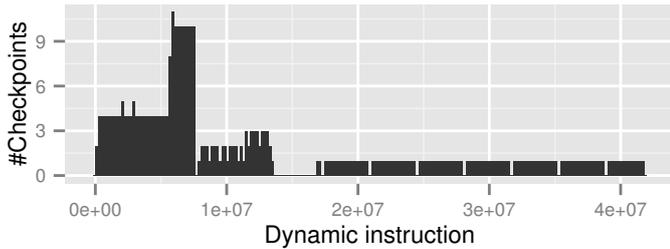


Figure 5: Distribution of 387 checkpoints (histogram bin size: 2×10^5 dynamic instructions) for the “small” BITCOUNT benchmark with a cost-capping checkpoint threshold of 500.

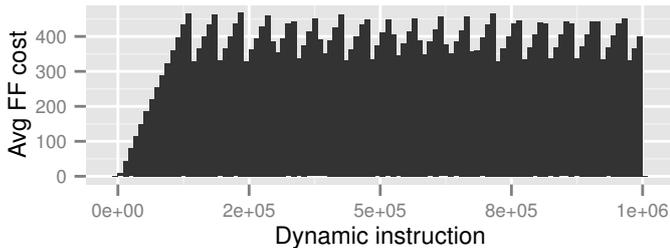


Figure 6: Average fast-forwarding costs for the first 10^6 dynamic instructions of the “small” BITCOUNT benchmarks after adding checkpoints (cf. Figure 5): Fast-forwarding costs are capped at the specified threshold of 500, transforming the linearly growing costs from Figure 4b to a saw-tooth growth curve.

fault-space coverage improvement of more than three orders of magnitude, with even more extreme improvements (e.g., DIJKSTRA “large” improves throughput by a factor of 346,217!) for longer-running programs.

On the other hand, the BITCOUNT benchmark demonstrates the limitations of smart-hopping: In the worst case, even optimal hop chains are too long on average for reasonable fast-forward times. Programs with very long loops that only operate on register contents, or on a very small set of data from memory, cannot be efficiently fast-forwarded using hardware breakpoints and watchpoints only. In the remaining paragraphs of this section we will discuss an extension and possible changes that would allow us to handle these pathological cases as well.

A. Extending the Approach: Checkpointing

For target programs that exceed reasonable average fast-forwarding costs, our approach can be complemented with

checkpointing (cf. Section II-C): We extended our heuristic by an automatism that creates a checkpoint whenever the fast-forwarding costs amounting in the `current_solution` variable (Algorithm 1) cross a specified threshold, effectively capping fast-forwarding costs there. This threshold must be chosen large enough that it exceeds the runtime for a checkpoint load, and should be tuned to even higher values to reduce the number of checkpoints that must be created and stored. Assuming checkpoint load costs of 300,⁴ and a threshold of 500, the “small” BITCOUNT benchmark’s fast-forwarding costs can be reduced to an average of 377.4 (max: 500) with only 387 checkpoints placed program-phase specifically by our heuristic. Figure 5 shows the distribution of these checkpoints over the benchmark’s runtime: While some program phases are very impervious to smart-hopping and therefore interspersed with checkpoints (up to 11 per 200,000 instructions), others can be fast-forwarded to without any need for additional checkpoints. Figure 6 shows the average cost curve from Figure 4b after adding checkpoints: The costs are capped at the specified threshold of 500.

B. Alternative and Complementary Improvements

An alternative for dealing with programs unfavorable for smart-hopping includes hardware breakpoint/watchpoint support that can be configured to trigger not at the first, but the n^{th} occurrence of the watched event. For example, the Lauterbach TRACE32 debugger attached to a Hitachi SH-4 or Freescale MPC5500 microcontroller offers this feature; unfortunately, these products are very expensive. Similarly, some CPUs come with hardware performance-counting units that can be exploited to fast-forward a specified number of, e.g., branch instructions. However, in many instances these counters only work “approximately accurate” [27], which counteracts the task of forwarding to an exact dynamic instruction. More widely available hardware support for fast-forwarding would certainly be desirable.

Another complementary approach is to reduce the debugger round-trip times ($t_{\text{singlestep}}, t_{BP}, \dots$), which directly improves all previously mentioned fast-forwarding techniques. Heinig et al. [28] recently showed that JTAG debugging round-trip times can be reduced to around 2 ms by implementing all timing-critical operations on a microcontroller, removing all USB and host-PC related delays from the fast-forwarding operation.

⁴Measurements on our ARM hardware indicate that it takes about 20 s, or ca. 300 cost units with 67 ms each, to load 2 MiB of data from the host PC into the target’s memory.

VI. CONCLUSIONS

In this article, we presented our *smart-hopping* heuristic, a trace-based fast-forwarding approach for running ISA-level fault-injection experiments on embedded platforms that only requires basic hardware support. We outlined algorithm details to ease implementation in other fault-injection platforms. A comparison to the prevalent simple-hopping approach, using instruction traces from the MiBench benchmark suite, showed that our heuristic improves experiment throughput by several orders of magnitude for most benchmarks.

A detailed analysis of a suboptimally performing input (the BITCOUNT benchmark) illustrated a principal limitation of smart-hopping using only breakpoints and watchpoints: without additional hardware support, programs with long-running, homogeneous loops and little to no memory accesses cannot be efficiently fast-forwarded. Subsequently we described an extension to our heuristic that automatically inserts checkpoints based on a fast-forwarding cost threshold, effectively capping costs. The resulting checkpoint positions are aligned with program-phase specific forwarding imperviousness, and the total checkpoint count – which directly results in recording time and hard-disk space consumption – can be tuned by increasing the threshold.

Taking into account the enormous performance gains and the satisfactory solution to even pathological cases, we conclude that the presented fast-forwarding mechanism offers a great potential: It gives developers of embedded systems the opportunity to systematically test the susceptibility of their software to hardware faults and to evaluate the effectiveness of fault-tolerance mechanisms on the real target hardware platform, which is the most accurate experiment platform possible.

Acknowledgments : This work was partly supported by the German Research Foundation (DFG) priority program SPP 1500 under grant no. SP 968/5-2. The authors would like to thank Björn Döbel for pointing us to related work on deterministic replay, and Björn Bönninghoff, Timon Kelter, Jan Kleinsorge and Richard Hellwig for lengthy discussions on path search and complementary solutions.

REFERENCES

- [1] S. Y. Borkar, "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.
- [2] M. Duranton, S. Yehia, B. de Sutter, K. de Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero, "The HiPEAC vision," HiPEAC, Tech. Rep., 2010.
- [3] V. Narayanan and Y. Xie, "Reliability concerns in embedded system designs," *IEEE Comp.*, vol. 39, no. 1, pp. 118–120, 2006.
- [4] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *42nd IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '12)*. IEEE, 2012, pp. 1–12.
- [5] C. Borchert, H. Schirmeier, and O. Spinczyk, "Generative software-based memory error detection and correction for operating system data structures," in *43rd IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '13)*. IEEE, Jun. 2013. [Online]. Available: <http://www.danceos.org/publications/DSN-2013-Borchert.pdf>
- [6] A. Benso and P. Prinetto, *Fault injection techniques and tools for embedded systems reliability evaluation*, ser. Frontiers in electronic testing. Boston, Dordrecht, London: Kluwer, 2003.
- [7] L. Berrojo, I. Gonzalez, F. Corno, M. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New techniques for speeding-up fault-injection campaigns," in *2002 Conf. on Design, Autom. & Test in Europe (DATE '02)*, 2002, pp. 847–852.
- [8] D. Skarin, R. Barbosa, and J. Karlsson, "GOOFI-2: A tool for experimental dependability assessment," in *40th IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '10)*. Los Alamitos, CA, USA: IEEE, Jul. 2010, pp. 557–562.
- [9] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *17th Int. Conf. on Arch. Support for Programming Languages and Operating Systems (ASPLOS '12)*. New York, NY, USA: ACM, 2012, pp. 123–134.
- [10] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: quantified error and confidence," in *2009 Conf. on Design, Autom. & Test in Europe (DATE '09)*. IEEE, 2009, pp. 502–506.
- [11] P. Ramachandran, P. Kudva, J. Kellington, J. Schumann, and P. Sanda, "Statistical fault injection," in *38th IEEE/IFIP Int. Conf. on Dep. Sys. & Netw. (DSN '08)*. IEEE, 2008, pp. 122–127.
- [12] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo, "Fault-list collapsing for fault-injection experiments," in *Annual Reliability and Maintainability Symposium*, Jan. 1998, pp. 383–388.
- [13] P. Folkesson, S. Svensson, and J. Karlsson, "A comparison of simulation based and scan chain implemented fault injection," in *28th Annual Int. Symp. on Fault-Tolerant Computing (FTCS '98)*. Los Alamitos, CA, USA: IEEE, 1998, pp. 284–293.
- [14] A. Fidalgo, M. Gericota, G. Alves, and J. Ferreira, "Using NEXUS compliant debuggers for real time fault injection on microprocessors," in *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design*. ACM, 2006, pp. 214–219.
- [15] A. Mohammadi, M. Ebrahimi, A. Ejlali, and S. G. Miremadi, "SCFIT: A FPGA-based fault injection technique for SEU fault model," in *2012 Conf. on Design, Autom. & Test in Europe (DATE '12)*, 2012, pp. 586–589.
- [16] C. M. Maunder and R. E. Tulloss, *The test access port and boundary-scan architecture*. IEEE, 1990.
- [17] H. Schirmeier, M. Hoffmann, R. Kapitza, D. Lohmann, and O. Spinczyk, "FAIL*: Towards a versatile fault-injection experiment framework," in *25th Int. Conf. on Arch. of Comp. Sys. (ARCS '12), Workshop Proceedings*, ser. Lecture Notes in Informatics, G. Mühl, J. Richling, and A. Herkersdorf, Eds., vol. 200. German Society of Informatics, Mar. 2012, pp. 201–210.
- [18] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *IEEE Int. W'shop. on Workload Characterization (WWC '01)*. Washington, DC, USA: IEEE, 2001, pp. 3–14.
- [19] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *USENIX TC*, 2005.
- [20] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *8th Int. Symp. on Code Gen. & Optimiz. (CGO '10)*. New York, NY, USA: ACM, 2010, pp. 2–11.
- [21] Y. Saito, "Jockey: a user-space library for record-replay debugging," in *6th Int. Symp. on Automated analysis-driven debugging*. ACM, 2005, pp. 69–76.
- [22] M. Rebaudengo and M. Sonza Reorda, "Evaluating the fault tolerance capabilities of embedded systems via BDM," in *17th IEEE VLSI Test Symposium*. IEEE, 1999, pp. 452–457.
- [23] R. Barbosa, J. Vinter, P. Folkesson, and J. Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency," in *5th Europ. Depend. Comp. Conf. (EDCC 2005)*, vol. 3463. Springer, Apr. 2005, p. 246.
- [24] O. Hannius and J. Karlsson, "Impact of soft errors in a jet engine controller," in *Computer Safety, Reliability, and Security*, ser. LNCS, F. Ortmeier and P. Daniel, Eds. Heidelberg, Germany: Springer, 2012, vol. 7612, pp. 223–234.
- [25] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.
- [26] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sadashti, R. Sen, K. Sewell, M. Shoaih, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comp. Arch. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.
- [27] ARM Limited, "ARM architecture reference manual, ARMv7-A and ARMv7-R edition (DDI 0406C)," Nov. 2011.
- [28] A. Heinig, I. Korb, F. Schmoll, P. Marwedel, and M. Engel, "Fast and low-cost instruction-aware fault injection," in *2nd GI W'shop on SW-Based Methods for Robust Embedded Sys. (SOBRES '13)*, 2013.