Short-Circuit Dispatch: Accelerating Virtual Machine Interpreters on Embedded Processors

Channoh Kim† Sungmin Kim† Hyeon Gyu Cho Dooyoung Kim Jaehyeok Kim Young H. Oh Hakbeom Jang Jae W. Lee
Sungkyunkwan University, Suwon, Korea
{channoh, rash4h, cho42me, dooyoungid, max250, younghwan, hakbeom, jaewlee}@skku.edu

Abstract—Interpreters are widely used to implement high-level language virtual machines (VMs), especially on resource-constrained embedded platforms. Many scripting languages employ interpreter-based VMs for their advantages over native code compilers, such as portability, smaller resource footprint, and compact codes. For efficient interpretation a script (program) is first compiled into an intermediate representation, or bytecodes. The canonical interpreter then runs an infinite loop that fetches, decodes, and executes one bytecode at a time. This bytecode dispatch loop is a well-known source of inefficiency, typically featuring a large jump table with a hard-to-predict indirect jump. Most existing techniques to optimize this loop focus on reducing the misprediction rate of this indirect jump in both hardware and software. However, these techniques are much less effective on embedded processors with shallow pipelines and low IPCs.

Instead, we tackle another source of inefficiency more prominent on embedded platforms—redundant computation in the dispatch loop. To this end, we propose Short-Circuit Dispatch (SCD), a low-cost architectural extension that enables fast, hardware-based bytecode dispatch with fewer instructions. The key idea of SCD is to overlay the software-created bytecode jump table on a branch target buffer (BTB). Once a bytecode is fetched, the BTB is looked up using the bytecode, instead of PC, as key. If it hits, the interpreter directly jumps to the target address retrieved from the BTB; otherwise, it goes through the original dispatch path. This effectively eliminates redundant computation in the dispatcher code for decode, bound check, and target address calculation, thus significantly reducing total instruction count. Our simulation results demonstrate that SCD achieves geometric speedups of 19.9% and 14.1% for two production-grade script interpreters for Lua and JavaScript, respectively. Moreover, our fully synthesizable RTL design based on a RISC-V embedded processor shows that SCD improves the EDP of the Lua interpreter by 24.2%, while increasing the chip area by only 0.72% at a 40nm technology node.

Keywords-Microarchitecture; Pipeline; Scripting Languages; Bytecodes; Interpreters; Dispatch; JavaScript; Lua

I. INTRODUCTION

Recently, the role of scripting languages has grown from a fast prototyping tool to a general-purpose programming environment that enables a variety of complex applications. For example, JavaScript is the default programming language for online web applications, and also being widely used for writing web servers (e.g., Node.js [1]) and standalone client applications (e.g., WebOS [2], Tizen [3]). Python [4], Ruby [5], PHP [6], and Lua [7] are also popular in various application domains. These scripting languages provide higher levels of abstraction with powerful built-in functions to allow the programmer to do more with fewer lines of code.

Scripting languages usually feature dynamic types, where the exact type of a variable is resolved only at runtime for a given program input. This makes it difficult for a static compiler to produce efficient code. Thus, interpreter-based virtual machines (VMs) are a popular execution environment for these languages, possibly augmented with dynamic code optimization techniques (e.g., Just-In-Time (JIT) compilation). However, JIT compilation is not practical on resource-constrained embedded devices for its large resource footprint and complexity of implementation, which leads to a longer time-to-market [8].

For efficient interpretation a script is first compiled into a platform-independent intermediate representation, or bytecodes, to eliminate the recurring cost of parsing the script. The canonical interpreter runs an infinite loop that fetches, decodes, and executes one bytecode at a time. Typically, the dispatcher code looks up a large jump table to retrieve the target address for a given bytecode, followed by an indirect jump, which often becomes the principal source of slowdown [8].

More specifically, two major sources of inefficiency exist in the dispatcher code. First, the target address of the indirect jump is difficult to predict with tens or even hundreds of potential targets, to cause frequent branch mispredictions. Second, it takes a large number of dynamic instructions to perform decode, bound check, and target address calculation for every bytecode, which are mostly redundant.

Most existing techniques tackle the first source to predict better the target address of the indirect jump in both hardware [9, 10, 11, 12, 13, 14] and software [8, 15,
The following contributions are made by this paper:

- We propose a novel architectural extension that enables fast, hardware-based bytecode dispatch, thus eliminating most of the redundant computation in the bytecode dispatch loop.
- We design and implement SCD, which efficiently overlays the bytecode dispatch onto the BTB with minimal hardware overhead.
- We provide a detailed evaluation of two production-grade script interpreters using a cycle-level simulator to demonstrate the effectiveness of SCD over the state-of-the-art.
- We run a fully synthesizable RTL model on FPGA to provide more realistic evaluation of SCD with larger inputs (executing over 2.29 trillion instructions in total) and more accurate estimation of area and energy consumption via synthesis using a TSMC 40nm standard cell library.

This paper makes the following contributions:

- We propose a novel architectural extension that enables fast, hardware-based bytecode dispatch, thus eliminating most of the redundant computation in the bytecode dispatch loop.
- We design and implement SCD, which efficiently overlays the bytecode dispatch onto the BTB with minimal hardware overhead.
- We provide a detailed evaluation of two production-grade script interpreters using a cycle-level simulator to demonstrate the effectiveness of SCD over the state-of-the-art.
- We run a fully synthesizable RTL model on FPGA to provide more realistic evaluation of SCD with larger inputs (executing over 2.29 trillion instructions in total) and more accurate estimation of area and energy consumption via synthesis using a TSMC 40nm standard cell library.

II. Motivation

A. Interpreters on Embedded Processors

Single-board computers are becoming more and more popular for so-called DIY electronics, including Arduino [21], Raspberry Pi [22], and Intel’s

1We were not able to build SpiderMonkey on RISC-V/Newlib successfully due to missing libraries at the time of this writing.
In the VM interpreter, a jump instruction contains the address of the bytecode handler to be executed. However, the target address is unknown until the bytecode fetch stage (Lines 2-5), which can be as many as hundreds of distinct bytecodes, a jump table is widely used. Thus, the indirect jump at the end is hard to predict as it has as many potential targets as the number of bytecodes. The conventional, PC-indexed BTB performs poorly for this jump instruction, which is a major source of inefficiency in the VM interpreter. Figure 2 confirms this with most branch mispredictions attributed to the indirect jump for dispatch in the Lua interpreter. (Refer to Section V for simulation methodology.)

Figure 1(a) shows an Alpha assembly code for the canonical dispatch loop for VM interpreters. At every iteration the dispatcher code fetches a new bytecode (e.g., ADD R1, R2, R3 in Lua) and increments the virtual PC (Line 3), decodes it to extract an opcode (e.g., 6-bit numeric code (0x06) representing ADD in Lua), calculates the target address to the handler (embedded in the switch statement), and jumps to the target address (e.g., to Line 11 for ADD) to execute the bytecode. Although simple, this code represents a common pattern among script interpreters and instruction-set simulators.

Figure 1(b) shows an Alpha assembly code for the dispatcher code (corresponding to Lines 2-6 in Figure 1(a)) generated by gcc with a -03 flag. It consists of four components: bytecode fetch (Lines 2-5), decode (Line 8), bound check (Lines 10-11), and target address calculation and jump (Lines 13-18). Since script interpreters typically have tens or even hundreds of distinct bytecodes, a jump table is widely used. Thus, the indirect jump at the end is hard to predict as it has as many potential targets as the number of bytecodes. The conventional, PC-indexed BTB performs poorly for this jump instruction, which is a major source of inefficiency in the VM interpreter. Figure 2 confirms this with most branch mispredictions attributed to the indirect jump for dispatch in the Lua interpreter. (Refer to Section V for simulation methodology.)

Figure 2: Branch MPKI breakdown for Lua interpreter.
higher $CPI_{\text{Base}}$ (denoted by (4)) due to narrower issue width, smaller caches, in-order scheduling, and so on.

Our simulation with Value-Based BTB Indexing (VBBI) predictor [9], representing the state-of-the-art, also confirms this point. VBBI aims to improve the indirect branch prediction accuracy for switch statement and virtual function calls. To index the BTB, VBBI uses a hash of the PC and a hint value that controls the branch (e.g., opcode), instead of the PC alone to effectively eliminates most of the branch mispredictions from the dispatcher code (with a <0.1% misprediction rate). However, this leads to only modest speedups at best on an embedded processor, with geometric speedups of 8.8% and 5.3% for Lua and JavaScript, respectively. (See Figure 7 in Section VI for more details.) Therefore, we need alternative solutions to accelerate VM interpreters on embedded processors.

B. Redundant Computation in Interpreters

Figure 3 shows a fraction of dispatch instructions (as in Figure 1) out of the entire interpreter loop on the Lua interpreter. All instructions between the interpreter loop header and the indirect jump to a handler are counted as dispatcher code. More than 25% of total instructions are spent on the dispatcher code. Rohou et al. [8] report a similar range of numbers for other VM interpreters measured on a x86_64 architecture: 16-25% for Python, 27% for JavaScript, and 33% for CLI (Common Language Infrastructure, or .NET framework).

We identify a major fraction of the dispatcher code as redundant. More specifically, the instructions for decode, bound check, and target address calculation (Lines 8-18 shaded in gray in Figure 1(b)) implement a pure function with no side effect, which always produces the same output value (i.e., jump target address) for the same input (i.e., bytecode).

This code block makes an ideal target for memoization, which effectively eliminate redundant computation, reducing dynamic instruction count significantly. Furthermore, bypassing this code block eliminates 10 cache accesses (for both instructions and data) to save energy consumption. Thus, we investigate a non-prediction-based technique to capitalize on this opportunity to improve both performance and energy efficiency of the VM interpreter.

III. Short-Circuit Dispatch

This section introduces Short-Circuit Dispatch (SCD), an ISA extension and microarchitectural organization, which effectively eliminates the two sources of inefficiency in the VM interpreter discussed in Section II (i.e., frequent branch mispredictions and redundant dynamic instructions). The key idea of SCD is to use part of the BTB to cache a software-created jump table for the dispatcher to be short-circuited to the correct target address upon fetching a bytecode. Although we assume only one critical jump table (i.e., the one for the dispatcher code) in this section, SCD can be easily extended to support multiple jump tables, which is discussed in Section IV. We have the following three design goals for SCD:

- **Broad applicability**: The ISA extension should be flexible enough to be applicable to multiple popular VM interpreters.
- **High performance**: The proposed design should yield significant speedups for those interpreters to outperform prior work.
- **Low cost**: The cost of implementation should be minimal in terms of area and power.

A. ISA Extension

To perform a jump table lookup on the BTB, SCD introduces three new registers as follows:

- **Rop (Opcode Register)**: This register holds an opcode to dispatch and is composed of two fields: one-bit
(branch-on-opcode) looks up the BTB with decode.

In addition to the three instructions discussed in Figure 4, the bytecode must be written not only to the target address (from the source register) in the Execute stage. Since a BTB entry can be invalidated by resetting the valid flag (Rop.v) and 32-bit data field (Rop.d). The data field is used as key for a BTB lookup.

- **Rmask (Mask Register)**: This register holds 32 mask bits to extract an opcode (numeric code that encodes the operation to perform like ADD) from a bytecode. Typically, the value of this register is set just once when the interpreter is launched.

- **Rbop-pc (BOP-PC Register)**: This register holds the PC value of the indirect jump instruction at the end of the dispatcher code. When the indirect jump instruction is fetched, the BTB is looked up for fast dispatch using the opcode stored in Rop.

**Figure 4: Transformed dispatch loop (original code taken from Figure 1(b))**

![Figure 4](image)

**Datapath for <inst>.op.** To implement <inst>.op (suffix-for-Rop-update), the following three components are newly added: 32-bit Mask Register (Rmask), 32-bit Opcode Register (Rop), and 32-bit AND gate. Once execution of <inst> is completed in the Execute stage, the result is masked with Rmask and stored into Rop. By providing Mask Register SCD saves (at least) one instruction as the opcode is automatically extracted from the bytecode just calculated. The opcode stored in Rop serves as a VM instruction (and Rop as virtual instruction register (IR)), which will later be used by bop and jru instructions.

**Datapath for bop.** Rbop-pc is used to store the address of the critical indirect jump instruction that dispatches bytecodes. At every cycle the value of PC is compared with that of Rbop-pc to see if this jump instruction is being fetched. If they match (i.e., the bop? signal is true), Rop.d is used as input for BTB lookup instead of PC. If it hits, the cache jump table entry (JTE) is retrieved, and PC is redirected to its target address in the following cycle; if not, PC is just incremented by 4 (assuming 32-bit instructions). Note that, in case of a hit Rop.v is reset to zero. Finally, Rbop-pc is updated to hold the pointer to the bop instruction.

**Datapath for jru and jte_flush.** A new JTE is inserted into the BTB by jru, which replaces the critical jump (jmp) instruction in the original dispatcher code. A JTE is formed by putting together a valid opcode (from Rop) and its target address (from the source register) in the Execute stage. Since a BTB entry can
The value of BTB entries are already the norm in today’s processor designs, there is still enough headroom. Moreover, a JTE is generally more likely to be reused than a BTB entry.

Example walk-through. To demonstrate the operations of SCD, we walk through an example scenario shown in Figure 6. Figure 6(a) depicts the initial state of the BTB with the tag field omitted for brevity. Initially, there are two valid BTB entries but no JTEs. We assume $R_{bop-pc}$ is already set correctly to point to the address of bop in the dispatch loop. After each step affected parts are shaded in gray.

Step 1 (slow path): Figure 6(b) shows the operation of the slow path (bop miss) inserting a new JTE into the BTB. 1. When a new bytecode, say OP_LOAD, enters the pipeline (marked by a *.op suffix), it is masked with $R_{mask}$ and stored into $R_{op}$. A BTB lookup with $R_{op}$, d at bop fails as no JTE is cached yet, hence falling through to the slow path. Then jru inserts a new JTE for OP_LOAD into the BTB.

Step 2 (fast path): Figure 6(c) illustrates the operation of the fast path (bop hit). 1. If an OP_LOAD bytecode is loaded into the pipeline again, a BTB lookup by bop hits this time. 2. Then the dispatcher code is short-circuited to immediately jump to the target address for OP_LOAD, which constitutes the fast path for dispatch.

Step 3 (jte_flush): Figure 6(d) shows the operations of jte_flush, which is called when the interpreter exists from the dispatch loop. jte_flush invalidates all JTEs in the BTB (but not BTB entries) by resetting their valid bits to zero.

C. Applying SCD to Popular Interpreters

To demonstrate the practicality of SCD, we apply SCD to two popular open-source script interpreters: Lua [7] and SpiderMonkey [29]. Lua is the most popular programming language for game programming, especially for writing plug-ins. SpiderMonkey is the default JavaScript engine for the Firefox web browser.
While SpiderMonkey provides a JIT compiler, we turn it off to run in interpreter mode. We produce SCD-augmented script versions as follows.

**Lua** [7]. We use Lua-5.3.0. The interpreter loop starts at Line 661 in `src/1vm.c`. In fact, the canonical interpreter loop code in Figure 1(a) is a stripped-down version of the Lua interpreter loop, and we omit the code due to their similarity. Likewise, the transformed assembly code is almost the same as the code shown in Figure 4. **SpiderMonkey** [29]. The main interpreter loop of SpiderMonkey-17.0 begins at Line 1322 in `js/src/jsinterp.cpp`. SpiderMonkey adopts variable-length bytecodes, and the program control takes different paths before reaching the common dispatcher code depending on the length of the bytecode. SpiderMonkey fetches a bytecode not only at the common dispatcher code block but also at the end of some bytecodes, such as `FUNCALL`, `BRANCH`, `LT`, and so on. Thus, we apply an `.op` suffix to the load instructions at three different locations: Line 1329 (default), Line 2480 (`FUNCALL`) and Line 1199 (common macro for the others). Otherwise, the transformed dispatch loop would look similar to Lua.

**IV. Discussion**

**Supporting multiple jump tables.** SCD can be applied to any jump table-based indirect jumps beyond the bytecode dispatch loop and can be easily extended to track multiple jumps. To support $n$ indirect jumps simultaneously, we need to replicate the set of three registers ($R_{op}$, $R_{mask}$, and $R_{op\_pc}$) by $n$ times and expand the J/B bit to an $n$-bit vector. A preferred implementation uses a one-hot encoding for branch ID to simplify hardware. All instructions in Table I are also extended to take a branch ID as immediate or register value to specify which indirect branch they are referring to. In the Fetch stage PC is compared with $n$ $R_{op\_pc}$'s in a way similar to the tag comparison in a $n$-way set-associative cache. If any $R_{op\_pc}$ hits, its ID (one-hot-encoded) will be used for BTB lookup. Likewise, jru also uses the branch ID value when inserting a new JTE into the BTB.

**OS Interactions.** In a realistic setup we should consider OS context switching. There are a spectrum of policies with regard to how we handle the newly introduced registers. Unlike BTB entries, which are used for prediction, JTEs directly affect the program execution path such that they should be either saved or flushed at the context switch. Our preferred way of handling it is to flush JTEs (and $R_{op}$) to minimize changes to the OS code. This is achieved by simply inserting a `jte_flush` instruction. Even so, we should save the Mask Register ($R_{mask}$) as the value of $R_{mask}$ must be preserved until the end of execution to ensure correctness. Once a process is scheduled again after a context switch, there will be no JTEs in the BTB, and it will take some cycles to populate JTEs again by taking the slow path.

**Contentions between BTB and JT entries.** Since JTEs have higher priority than BTB entries, cold JTEs might occupy most of the BTB capacity to degrade overall branch predictor performance. This problem is more pronounced with small BTBs and many distinct bytecodes used for a workload. In such cases the cost of extra branch target misses for direct branches can outweigh the benefit of short-circuit dispatch. A practical solution to the problem is to cap the maximum number of JTE in the BTB at any given time. We implement and evaluate this solution with a small BTB in Section VI-C.

**Application to high-end processors.** While SCD is also applicable to high-end processors, its benefits are most pronounced on low-end processors, where JIT is not practical. A typical single-board computer features a single core running at tens to low hundreds of MHz with memory size ranging from KBs to low hundreds of MBs. In such a resource-constrained environment JIT is not a viable option. Besides, the effectiveness of JIT depends highly on the existence of a handful of hot methods dominating total execution time, which may not be the case in real workloads [30, 31]. Unlike JIT, SCD is applicable to low-end processors and to workloads without hotspots.
V. EXPERIMENTAL SETUP

We use both a cycle-level simulator and FPGAs to evaluate SCD on embedded processors. For simulation we extend gem5 [32], and the architectural parameters are summarized in Table II. We use the MinorCPU processor model included in gem5 and take most of architectural parameters from ARM Cortex-A5 [33], a popular embedded application processor. We also implement the VBBI [9] predictor and jump threading [8] for comparison, which represent the state-of-the-art hardware and software techniques, respectively.

For FPGA emulation we have written a fully synthesizable RTL model for SCD, to demonstrate its ISA independence and effectiveness with large inputs (executing over 2.29 trillion instructions in total). Our model is based on a open-source 64-bit RISC-V v2 Rocket core [34] written in Chisel language, whose parameters are also summarized in Table II. This model is compiled into Verilog RTL, which is then synthesized for FPGA emulation and area/power estimation. Xilinx ZC706 FPGAs are used for execution. We use the default RISC-V/Newlib version. We only use the Lua interpreter for FPGA emulation since we fail to build a SpiderMonkey binary due to missing libraries on RISC-V/Newlib.

We synthesize the same RTL model using Synopsys Design Compiler (Version B-2008.09-SP5-1) to report an realistic estimation of area and power. We use 5 TSMC CLN40G technology libraries at a 40nm technology node, which are a 9-track standard cell library (SC9) and 4 SRAM libraries generated by ARM Artisan memory compilers. A standard cell library for most-typical corner is chosen (rvt_tt typical max 0p90v 25c). SRAM libraries of tag and data arrays of I- and D-caches are generated by high density 1-port regfile, high density 1-port SRAM and high speed 2-port regfile memory compilers.

As for workloads we experiment with the following two popular open-source interpreters: Lua-5.3.0 [7] and SpiderMonkey-17.0 [29]. Both script interpreters are compiled with gcc -O3, and we turn off garbage collection to not disturb the mutator (main) code. Lua has 47 distinct bytecodes and the dispatch loop consists of 35 native instructions. SpiderMonkey is written in C++. We turn off the JIT compiler to run in the interpreter mode. It has 229 distinct bytecodes, and the dispatch loop takes 29 native instructions. We implement custom jump threaded versions for both interpreters, for which we preserve the (almost) same code layout for

2 We use the same small input for both the simulator and the FPGA due to a system failure with a large input on FPGA.
non-dispatcher code. In this way, we can compare the performance impact of different dispatching schemes in a fair manner.

We initially take the same set of 11 scripts from recent work [19], but end up replacing four of them for the following reasons. The replaced benchmarks are either not working on SpiderMonkey (fasta and meteor) or Lua (reverse–complement) or spending most of the time on native library code rather than bytecodes (regex–dn$. Instead, four new benchmarks, n–sieve, random, fibo, and ackermann, are taken from Computer Language Benchmarks Game [35], where the 11 benchmarks in [19] originate from. We run all benchmarks to completion with inputs summarized in Table III and measure the cycle count from the beginning and the end of the interpreter loop.

VI. EVALUATION

A. Overall Speedups

1) Speedups on Simulator: Figure 7 shows the overall performance speedups of SCD, along with jump threading [8] and VBBI [9], normalized to the out-of-the-box baseline. SCD achieves a geomean speedup of 19.9% and 14.1% for Lua and SpiderMonkey, respectively, with maximum speedups of 38.4% and 37.2%. This compares favorably to the other techniques with 1.16% and 7.3% speedups for jump threading, and 8.8% and 5.3% for VBBI.

These performance gains with SCD are mainly attributed to the following two factors. First, as Figure 8 shows, the normalized total dynamic instruction counts of SCD are reduced by 10.2% and 9.6% on average for Lua and SpiderMonkey, respectively. Furthermore, the branch misprediction rates, represented in misses in kilo-instructions (MPKI), are also reduced by 70.6% and 28.1% (Figure 9). We will provide more detailed discussion for the two interpreters in the following.

Lua. VBBI and SCD achieve geomean speedups of 8.8% and 19.9% with maximum speedups of 16.8% and 38.4% for mandelbrot, respectively. While both have comparable branch misprediction rates and instruction cache miss rates (shown in Figure 10), SCD has significantly lower instruction count. However, jump threading shows 2% worse performance than baseline in the instruction cache miss rate increases from 0.28 MPKI (baseline) to 4.80 MPKI (jump threading). The
branch misprediction rates are decreased for all three schemes by 70.5%, 77.5%, and 24.4% for SCD, VBBI, and jump threading, respectively.

**SpiderMonkey.** VBBI and SCD achieve geometric speedups of 5.3% and 14.1%, respectively, with maximum speedups of 14.6% and 37.2% for *fannkuch-redux*. Like Lua the primary source of improvement of SCD over VBBI is a lower dynamic instruction count. Jump threading achieves a geometric speedup of 7.3% with the maximum speedup of 18.7% for *fannkuch-redux*. While instruction count and branch misprediction rate are decreased by 13.8% and 4.4%, respectively, over the baseline, jump threading increases the instruction cache miss rate by 25.3%. Overall, the performance gains of SpiderMonkey are lower than Lua partly due to the existence of multiple paths to the dispatcher code, and SCD is not applicable to all paths.

2) **Speedups on FPGA:** Table IV summarizes the cycle count and instruction count of the Lua interpreter running on FPGA. We compare three versions: the baseline, jump threading, and SCD. Columns 8 and 10 represent a reduction ratio in instruction count over the baseline with jump threading and SCD, respectively. Columns 9 and 11 represent the speedups over the baseline, respectively. SCD achieves geometric speedup of 12.0% with maximum 22.7% for *mandelbrot*. SCD reduces instruction count by 10.4% on average over the baseline, which is comparable to the simulator results.

Jump threading shows a negligible geometric speedup of 0.01% on FPGA with a maximum speedup of 7.5%. It reduces instruction count by 4.8% on average with a maximum of 5.9%. For *n-sieve*, jump threading experiences an 11.1% slowdown. This is likely caused by increased instruction cache misses as we have discussed in Section VI-A1.

### B. Area and Energy Consumption

Table V reports the area and power estimation of SCD implemented on a RISC-V Rocket Core. The target frequency is 500 MHz, and both the baseline and SCD satisfy this constraint. With SCD, the total area and power are increased by 0.72% and 1.09%, respectively. Combined with speedup numbers in Section VI-A2, these numbers translate to a 24.2% improvement in energy-delay product (EDP). According to the area/power breakdown, the BTB accounts for 33% and 7% of total area and power, respectively, which is increased by 21.6% and 11.7% with SCD.
Overall, the BTB module is responsible for an increase of 0.59% and 0.90% for area and power by integrating SCD. According to the timing reports SCD does not affect the critical timing path of the original design as the critical path is in the FPU module before and after integrating SCD. Note that, we have not performed any microarchitectural optimization, so there is still significant room for improvement with this result.

C. Sensitivity Study

1) Sensitivity to BTB size and maximum cap on the number of JTEs: Since the software jump table is overlaid onto the BTB, normal BTB entries and jump table entries (JTEs) compete for the BTB space. Our default policy gives a higher priority to JTEs, and regular directed branches can be penalized for this. In the worst case the cost of extra branch target misses for direct branches can outweigh the benefit of short-circuit dispatch. This problem is likely to be more pronounced for small BTBs.

Therefore, we perform a sensitivity study with varying BTB capacity. Figures 11 (a) and (b) represent the result of the sensitivity study with varying BTB size for Lua and SpiderMonkey, respectively. In the graphs X-axis represents the number of BTB size. Y-axis represents speedups of SCD over baseline for each BTB size. While the performance benefit decreases with smaller BTBs, SCD still significantly outperforms the baseline even with a small BTB size (64).

Figure 11 (c) and (d) show the effects of capping the maximum number of JTEs in the BTB at any given time with the smallest BTB size. X-axis represents the maximum cap on the number of JTEs. While capping brings only modest speedups compared to the baseline (denoted by “∞”), some programs get significant boost of performance (e.g., n–sieve). We will leave selecting an optimal cap value for future work.

2) Performance on a higher-end core: We evaluate SCD on a higher-performance in-order core based on ARM Cortex-A8 [33]. We adopt a dual-issue pipeline and increase the size of L1 I-cache to 32KB with 4 ways, L2 cache to 256KB, and BTB to 512 entries. SCD still achieves comparable performance on this core with geometric speedups of 17.6% and 15.2% for Lua and SpiderMonkey, respectively, and reduced instruction counts by 10.2% and 9.2%.

Table IV: Cycle count and instruction count of Lua interpreter using RISC-V Rocket Core on FPGA

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>binary-trees</td>
<td>2.40</td>
<td>3.68</td>
<td>2.39</td>
<td>3.68</td>
<td>2.29</td>
<td>3.68</td>
<td>2.29</td>
<td>3.68</td>
<td>2.29</td>
<td>3.68</td>
</tr>
<tr>
<td>lannicka-redux</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
<td>595.3B</td>
</tr>
<tr>
<td>k–nucleotide</td>
<td>88.7B</td>
<td>155.3B</td>
<td>85.4B</td>
<td>159.0B</td>
<td>82.4B</td>
<td>142.9B</td>
<td>3.9%</td>
<td>4.3%</td>
<td>3.9%</td>
<td>4.3%</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>218.7B</td>
<td>210.5B</td>
<td>214.1B</td>
<td>214.1B</td>
<td>3.9%</td>
<td>3.9%</td>
<td>3.9%</td>
<td>3.9%</td>
<td>3.9%</td>
<td>3.9%</td>
</tr>
<tr>
<td>n–body</td>
<td>426.4B</td>
<td>341.2B</td>
<td>345.4B</td>
<td>313.1B</td>
<td>2.7%</td>
<td>2.7%</td>
<td>2.7%</td>
<td>2.7%</td>
<td>2.7%</td>
<td>2.7%</td>
</tr>
<tr>
<td>spectral–norm</td>
<td>507.2B</td>
<td>741.2B</td>
<td>485.1B</td>
<td>702.3B</td>
<td>461.5B</td>
<td>627.6B</td>
<td>3.9%</td>
<td>1.2%</td>
<td>3.9%</td>
<td>1.2%</td>
</tr>
<tr>
<td>n–sieve</td>
<td>4.0B</td>
<td>3.4B</td>
<td>3.8B</td>
<td>4.3B</td>
<td>3.6B</td>
<td>4.3B</td>
<td>6.2%</td>
<td>11.3%</td>
<td>6.2%</td>
<td>11.3%</td>
</tr>
<tr>
<td>random</td>
<td>3.5B</td>
<td>4.8B</td>
<td>3.3B</td>
<td>4.3B</td>
<td>3.2B</td>
<td>4.3B</td>
<td>7.2%</td>
<td>3.9%</td>
<td>7.2%</td>
<td>3.9%</td>
</tr>
<tr>
<td>fpio</td>
<td>3.8B</td>
<td>3.4B</td>
<td>3.2B</td>
<td>3.5B</td>
<td>3.0B</td>
<td>3.6B</td>
<td>5.0%</td>
<td>5.3%</td>
<td>5.0%</td>
<td>5.3%</td>
</tr>
<tr>
<td>pingpads</td>
<td>598.2B</td>
<td>799.2B</td>
<td>564.3B</td>
<td>816.2B</td>
<td>545.2B</td>
<td>790.2B</td>
<td>5.4%</td>
<td>2.4%</td>
<td>5.4%</td>
<td>2.4%</td>
</tr>
</tbody>
</table>

Table V: Hardware overhead breakdown (area, power)
There are many existing proposals to improve interpreter performance. Hoogerbrugge et al. [38] propose software pipelining for inter-

VII. RELATED WORK

Hardware-based techniques. Indirect branches are heavily used in modern high-level programming languages for function pointers, switch-type statements, etc. Naturally, there are a number of attempts to optimize them using specialized hardware. ARM Thumb2 ISA includes complex table branch instructions, such as tbh and tbb, which combine jump table lookup, target address calculation, and jump into a single instruction. However, these instructions only reduce instruction count but do not eliminate redundant computation, hence yielding limited speedups. According to ARM’s software optimization guides [20], these instructions take at least 6 cycles (i.e., equivalent to 6 instruction slots on a single-issue processor) before fetching the correct target instruction. Kaeli and Emma [28] propose Case Block Table (CBT) to optimize switch-type statements, which is perhaps the most similar in spirit with SCD, but with very different interface and organization. Their ISA extension marks only the beginning and end of a switch statement and rely on a very specific pattern of generated code to identify a key-target address pair, which limits applicability. Furthermore, CBT is an auxiliary hardware table without overlaying on the BTB, incurring higher cost than SCD. Finally, their proposal lacks realistic evaluation and many design details.

Another avenue of proposals aim to improve indirect branch prediction in general. The conventional branch target buffer (BTB) [36] is not effective for an indirect branch with multiple targets. There are many proposals to address this [9, 10, 11, 12, 13]. Chang et al. [11] propose history-based Tagged Target Cache (TTC), which stores multiple targets in a target cache for predicting the indirect jumps. Driesen et al. [12, 13] propose Cascaded Predictor, a hybrid predictor combining a simple first-stage predictor with a complex second-stage one. VPC prediction [10] exploits the existing direct predictor for predicting indirect branches. Recently, Farooq et al. [9] propose VBBI, which demonstrates the effectiveness of value correlation for indirect branches. Sezne and Michaud [37] propose the IT-TAGE predictor, which is the most accurate branch predictor and relies on multiple predictor tables indexed with global history. However, these techniques mostly target high-end processors with high misprediction cost and are less effective on thin embedded processors (as discussed in Section II). Instead, SCD not only improves the accuracy of prediction but also eliminates redundant computation, to significantly speed up VM interpreters on embedded systems.

Software-based techniques. There are many existing proposals to improve interpreter performance. Hoogerbrugge et al. [38] propose software pipelining for inter-
Short-Circuit Dispatch (SCD), a low-cost hardware-based technique to accelerate the VM interpreter on embedded platforms. The key idea of SCD is to overlay the software-created bytecode jump table on a branch target buffer (BTB), realizing efficient memoization for eliminating redundant computation in the dispatcher code. Our cycle-level simulation with gem5 demonstrates the effectiveness of SCD with geomean speedups of 19.9% and 14.1% for two production-grade script interpreters for Lua and JavaScript, respectively. Moreover, our fully synthesizable RTL design based on a RISC-V embedded processor shows that SCD improves the EDP of the Lua interpreter by 24.2%, while increasing the chip area by only 0.72% at a 40nm technology node.

IX. Acknowledgments

This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1501-07.

References