2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

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, vash4h, 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 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.

***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,

[†]These authors contributed equally to this work.

1063-6897/16 $31.00 © 2016 IEEE
DOI 10.1109/ISCA.2016.34

```
for (;;) {
  // fetch next bytecode
  Bytecode bc = *(VM.pc++);
  // decode and jump
  int opcode = bc & mask;
  switch (opcode) {
  // execute bytecode
  case: LOAD
    do_load(RA(i),RB(i));
    break;
  case: ADD
  ...
  default:
    error();
  }
}
```

(a)

```
Fetch:
  ldq     r5,40(r14)       # r5 = Mem[r14 + 40]
  ldl     r9,0(r5)         # r9 = Mem[r5]
  lda     r5,4(r5)         # r5 = r5 + 4
  stq     r5,40(r14)       # Mem[r14 + 40] = r5
Decode-and-Jump:           ## slow path ##
  ## decode
  and     r9,63,r2         # r2 = r9 & 63
  ## bound check
  cmpule  r2,45,r1         # r1 = (r2 <= 45)
  beq     r1,Default       # branch if (r1 == 0)
  ## target address calculation and jump
  ldah    r7,T(r3)         # r7 = r3+(T[31:16]<<16)
  lda     r7,T(r7)         # r7 = r7 + T[15:0]
  s4addq  r2,r7,r2         # r2 = (r2 << 2) + r7
  ldl     r1,0(r2)         # r1 = Mem[r2]
  addq    r3,r1,r1         # r1 = r3 + r1
  jmp     r31,(r1),Load    # jump to r1
```

(b)

```
// save the labels
void *labels[] =
  {&&LOAD, &&ADD, ...};
// fetch; decode; jump
Bytecode bc = *(VM.pc++);
int opcode  = bc & mask;
goto *labels[opcode];

LOAD:
  do_load(RA(i), RB(i));
  // fetch; decode; jump
  bc = *(VM.pc++);
  opcode = bc & mask;
  goto *labels[opcode];
ADD:
  ...
Default:
  error();
```

(c)

Figure 1: (a) Canonical dispatch loop in C; (b) Alpha assembly code; (c) Jump-threaded dispatch loop

16, 17, 18, 19]. However, our analysis (in Section II-A) demonstrates that these techniques are less effective on embedded processors with shallow pipelines and low IPCs. To mitigate the second source of inefficiency, some ISAs include complex instructions, which combine multiple simpler instructions into a single instruction to make the dispatcher code more compact (e.g., table branch instructions in ARM [20]). However, these instructions only reduce instruction count but do not eliminate redundant computation, hence yielding limited speedups.

This paper proposes Short-Circuit Dispatch (SCD), a low-cost architectural extension for fast, hardware-based bytecode dispatch. The key idea of SCD is to overlay the software-created bytecode jump table on a branch target buffer (BTB). Using the BTB as an efficient hardware lookup table, SCD caches most frequently used jump table entries in a portion of the 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 for the bytecode; otherwise, it goes through the original dispatch path. Unlike multi-target indirect branch predictors [9, 10, 11, 12, 13, 14], where the target address must still be computed in software, SCD is the *first* to use the BTB as a cache for the bytecode jump table, bypassing most of the redundant instructions for target address calculation with minimal hardware cost.

We evaluate SCD on two production-grade script interpreters for Lua and JavaScript with 11 scripts for each, using both a cycle-level simulator and a synthesizable RTL model on FPGA. Our simulation shows that SCD achieves geomean speedups of 19.9% and 14.1% with maximum speedups of 38.4% and 37.2% for Lua and JavaScript, respectively, while a state-of-the-art indirect branch predictor [9] yields only 8.8% and 5.3% geomean speedups. Moreover, our fully synthesizable RTL model based on a RISC-V embedded processor shows a geomean speedup of 12.0% with a maximum speedup of 22.7% for the Lua interpreter running on FPGA[1]. According to our synthesis results using a TSMC 40nm standard cell library, SCD improves the EDP of the Lua interpreter by 24.2%, while increasing the chip area by only 0.72%.

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 jump table 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

[1] We were not be able to build SpiderMonkey on RISC-V/Newlib successfully due to missing libraries at the time of this writing.

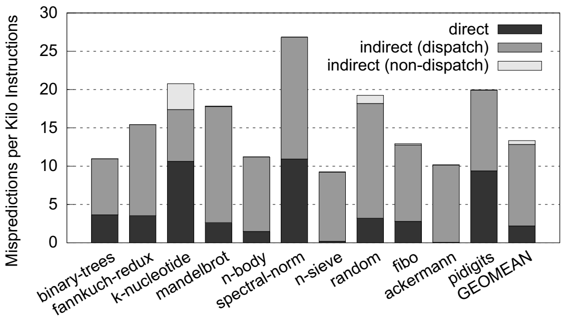


Figure 2: Branch MPKI breakdown for Lua interpreter

Galileo [23], to name a few. These computers commonly feature small form factors, low cost, flexible I/O interfaces, and open-source hardware/software stack, to enable a variety of IoT applications. Scripting languages are already widely adopted in those platforms for their productivity benefits. For example, Raspberry Pi promotes Python [24] as default programming language, and Intel supports JavaScript for IoT programming through Intel XDK [23].

Many single-board computers [21, 22, 23, 25, 26, 27] employ single-core embedded processors with a shallow pipeline running at tens to low hundreds of MHz and small memory size ranging from KBs to low hundreds of MBs. In such a resource-constrained environment interpretation is often a more viable option to run scripts than JIT compilation due to its smaller resource footprint and ease of development [8]. Unfortunately, interpreted codes are far too slower than compiled codes, and this *efficiency gap* should be adequately addressed for scripting languages to be more widely used not only for prototyping but also for production.

Figure 1(a) shows a simplified C code of 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 (`0x0E`) 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 `-O3` 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.)

Naturally, most existing techniques for accelerating interpreters focus on reducing the misprediction rate of the indirect jump instruction in both hardware and software. Hardware techniques propose more sophisticated indirect branch predictors that support multi-target prediction from a single indirect jump [9, 10, 11, 12, 13, 14]. In contrast, software techniques apply code transformations to reduce misprediction rate. Figure 1(c) illustrates a popular example of such techniques, called *jump threading* [8]. Jump-threaded dispatch requires a compiler support to store the addresses of all labels into a static array (Lines 2-3), such as Labels-as-Values GNU extension. In essence, jump threading replicates the dispatcher code at the end of each bytecode handler (Lines 12-14) so that each of the replicated indirect jumps occupies a distinct BTB entry. A downside of jump threading is that it can cause a code bloat to increase instruction cache misses.

Although these techniques significantly reduce the misprediction rate, they are not effective on embedded processors. To make this point we present a simple analysis of CPI improvement when a new, sophisticated indirect branch predictor is employed. The amount of CPI improvement (α) can be represented as follows:

$$\alpha = \frac{CPI_{\text{Base}}}{CPI_{\text{New}}} = \frac{1}{1-\frac{\Delta CPI}{CPI_{\text{Base}}}} = \frac{1}{1-\beta}$$

where

$$\beta = \frac{\Delta CPI}{CPI_{\text{Base}}} = \frac{\frac{\#\ bytecodes}{Total\ inst.\ count}^{(1)} * \Delta mispred\_rate^{(2)} * miss\_penalty^{(3)}}{CPI_{\text{Base}}{}^{(4)}}$$

According to the formula, the CPI improvement (α) is usually smaller on an embedded processor than a desktop/server processor. The first two terms in the numerator of β (denoted by (1) and (2); $0 \leq \beta \leq 1$) remain relatively constant for the two, assuming the same ISA and branch predictors. However, the embedded processor generally has lower branch misprediction penalty (denoted by (3)) due to shallower pipeline and

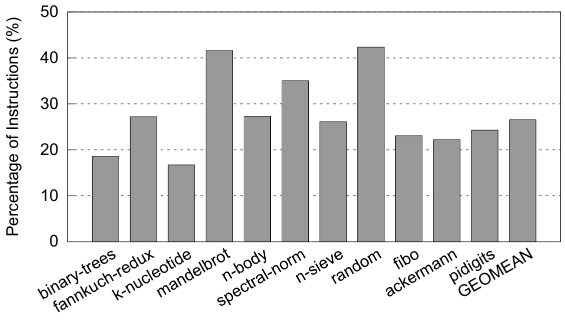


Figure 3: Fraction of dispatch instructions for Lua

higher CPI_{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 geomean speedups of 8.8% and 5.3% for Lua and JavaScript interpreters, 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.

Instruction	Description
setmask R_n (Set-mask)	$R_{\mathtt{mask}} \leftarrow R_n$
<inst>.op (Suffix-for-R_{op}-update)	execute <inst> $R_{op}.\mathtt{d} \leftarrow R_{\mathtt{mask}}$ & result of <inst> $R_{op}.\mathtt{v} \leftarrow 1$

bop
(Branch-on-opcode)
	if ($R_{\mathtt{bop-pc}}$ == PC && $R_{op}.\mathtt{v}$
	&& BTB.valid($R_{op}.\mathtt{d}$))
	PC ← BTB.targetaddr($R_{op}.\mathtt{d}$)
	$R_{op}.\mathtt{v} \leftarrow 0$
	else PC ← PC + 4
	$R_{\mathtt{bop-pc}}$ ← PC of bop

jru R_n
(Jump-register-with-jte-update)
	PC ← R_n
	if ($R_{op}.\mathtt{v}$)
	BTB.update($R_{op}.\mathtt{d}$, R_n)
	$R_{op}.\mathtt{v} \leftarrow 0$

jte_flush (Flush-all-jte)	Flush all jump table entries in BTB $R_{op}.\mathtt{v} \leftarrow 0$

Table I: ISA extension with SCD

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:

- R_{op} (*Opcode Register*): This register holds an opcode to dispatch and is composed of two fields: one-bit

```
1  Fetch:
2    ldq     r5,40(r14)
3    ldl.op  r9,0(r5)
4    lda     r5,4(r5)
5    stq     r5,40(r14)
6  Branch-on-opcode:
7    bop
8  Decode-and-Jump:          ## slow path ##
9    ## decode
10   and     r9,63,r2
11   ## bound check
12   cmpule  r2,45,r1
13   beq     r1,Default
14   ## target address calculation and jump
15   ldah    r7,T(r3)
16   lda     r7,T(r7)
17   s4addq  r2,r7,r2
18   ldl     r1,0(r2)
19   addq    r3,r1,r1
20   jru     r31,(r1),Load
```

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

valid flag (R_{op}.v) and 32-bit data field (R_{op}.d). The data field is used as key for a BTB lookup.

- R_{mask} (*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.
- $R_{bop\text{-}pc}$ (*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 R_{op}.

Figure 4 illustrates how the original dispatcher code in Figure 1(b) is re-targeted to SCD for efficient dispatch. The modified lines are shown in gray. In Line 3 the load instruction (ldl) is suffixed with .op, which indicates the bytecode must be written not only to r9 but also to R_{op} (Opcode Register) after masking with R_{mask}. The masking operation corresponds to the decode instruction in Line 8 of Figure 1(b). Then bop (branch-on-opcode) looks up the BTB with R_{op}.d as key to see if a jump table entry (JTE) already exists for the opcode (Line 7). If it hits, PC is redirected to the target address retrieved from the BTB; otherwise, it will just fall through to the slow path starting from Line 8. The original indirect jump (jmp) instruction is replaced with jru (jump-register-with-jte-update) in Line 20, which jumps to the target address (r1) and inserts a (R_{op}.d, r1) pair into the BTB. Note that, once a dispatch is completed by either bop (fast path) or jru (slow path), R_{op} is invalidated by resetting R_{op}.v to zero.

Table I summarizes the new instructions introduced by SCD. In addition to the three instructions discussed with Figure 4 (<inst>.op, bop, jru), there are two more instructions. setmask sets the value of Mask Register (R_{mask}), which is set before the interpreter loop begins. For example, since the opcode field of a Lua bytecode is placed at the 6 least-significant bits (LSBs), the mask bits are set to 0x0000003F. jte_flush is used to invalidate all jump table entries currently residing in the BTB. This instruction is invoked at a context switch or exit of the interpreter loop to prevent erroneous operations. Interactions with OS are discussed in greater details in Section IV.

B. SCD Organization

Figure 5 shows a pipeline structure to implement the extended ISA introduced by SCD in Section III-A. To minimize hardware cost we overlay (part of) the software-created jump table onto the BTB. This is different from Case Block Table (CBT) [28], which is similar in spirit but requires an auxiliary on-chip table to store jump target addresses. A BTB entry is extended with a new flag, called $J/\bar{B}$ bit. If this bit is set to one (J), the entry holds a jump table entry (JTE) for a bytecode; if zero ($\bar{B}$), it holds a normal BTB entry. The rest of this section covers various issues in microarchitectural design.

Datapath for <inst>.op. To implement <inst>.op (suffix-for-R_{op}-update), the following three components are newly added: 32-bit Mask Register (R_{mask}), 32-bit Opcode Register (R_{op}), and 32-bit AND gate. Once execution of <inst> is completed in the Execute stage, the result is masked with R_{mask} and stored into R_{op}. 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 R_{op} serves as a VM instruction (and R_{op} as virtual instruction register (IR)), which will later be used by bop and jru instructions.

Datapath for bop. $R_{bop\text{-}pc}$ 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 $R_{bop\text{-}pc}$ to see if this jump instruction is being fetched. If they match (i.e., the bop? signal is true), R_{op}.d is used as input for BTB lookup instead of PC. If it hits, the cached 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 R_{op}.v is reset to zero. Finally, $R_{bop\text{-}pc}$ 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 R_{op}) and its target address (from the source register) in the Execute stage. Since a BTB entry can

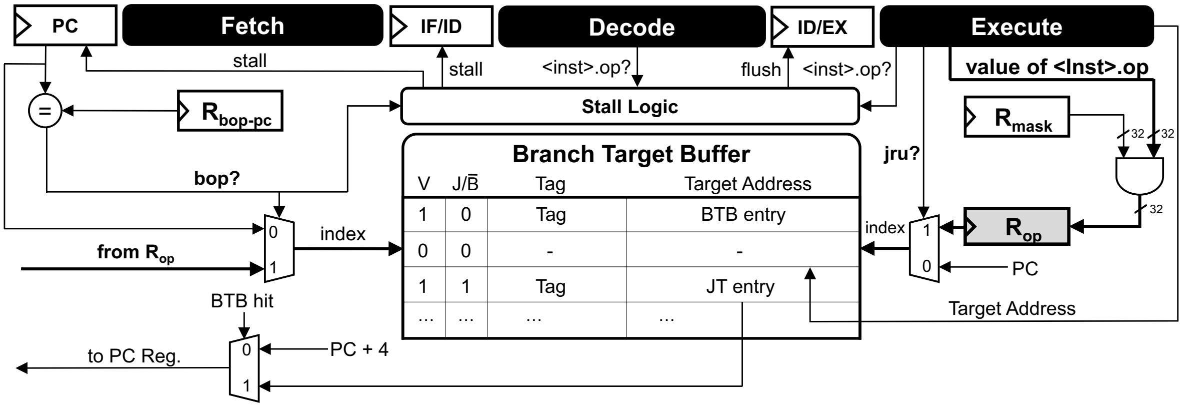


Figure 5: Pipeline structure augmented with SCD

now hold both an original BTB entry (providing PC-to-target address *prediction*) and an JTE (providing opcode-to-target address *translation*), it is extended with a new flag, JTE/$\overline{BTB}$ (J/$\bar{B}$) bit, to indicate which one it currently holds. A valid JTE in the BTB has the J/$\bar{B}$ bit set to one; if the valid bit (V) is zero, this flag is ignored. When a short-circuited dispatch is attempted with an opcode in R_{op}, only JTEs (but not BTB entries) will be searched for a match. Likewise, `jte_flush` invalidates (i.e., resets the valid bit to zero) only JTEs but not BTB entries.

Stall logic. There may be cases when the value of R_{op} is not available yet when a `bop` enters the Fetch stage. There are two solutions to this. First, the pipeline normally proceeds with the `bop` falling through to the slow path. In this case there is no performance improvement from short-circuited dispatch. Second, the pipeline detects any in-flight instruction to update R_{op}, and stalls the `bop` at the Fetch stage until the value of R_{op} becomes available. This inserts several bubbles (`nop`'s) into the pipeline, but can benefit from short-circuited dispatch. Since we target embedded processors featuring shallow pipelines, the benefit of fast dispatch generally outweighs the cost of bubbles. Thus, SCD adopts the second (stalling) scheme by default, which is implemented by stall logic in Figure 5.

JT/BTB entry replacement. Since both JT and BTB entries share the same BTB, a replacement policy should be carefully designed. The default policy is to give a higher priority to the JTE. In other words, an incoming JTE can evict a BTB entry, but not the other way around. This policy is justified with the following facts. Our evaluation reveals that, even if an interpreter defines tens or even hundreds of distinct bytecodes, only a small fraction of them, say at most low tens of them, are actually used. Since high tens, or hundreds 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.

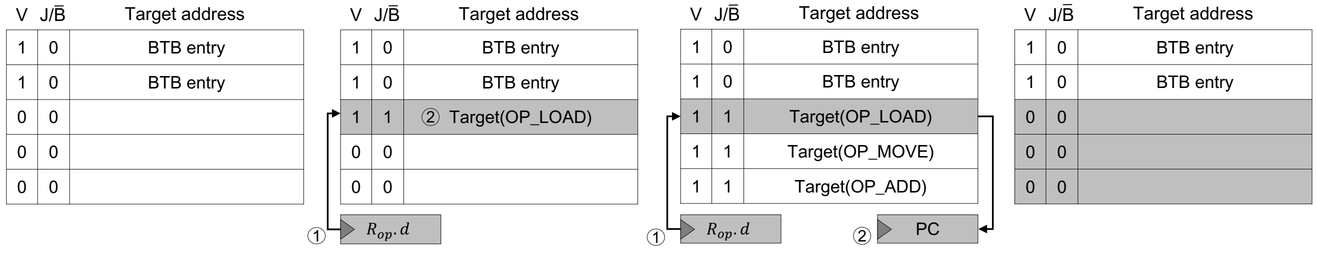
Step 1 (slow path): Figure 6(b) shows the operation of the slow path (`bop` miss) inserting a new JTE into the BTB. ① When a new bytecode, say `OP_LOAD`, enters the pipeline (marked by a `*.op` suffix), it is masked with R_{mask} and stored into R_{op}. A BTB lookup with R_{op}`.d` at `bop` fails as 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.



(a) Initial state **(b) Slow path - bop miss** **(c) Fast path - bop hit** **(d) jte_flush**

Figure 6: Running example of SCD operations

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/lvm.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 ($\texttt{R}_{\texttt{op}}$, $\texttt{R}_{\texttt{mask}}$, and $\texttt{R}_{\texttt{bop-pc}}$) by n times and expand the J/$\bar{B}$ 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 $\texttt{R}_{\texttt{bop-pc}}$'s in a way similar to the tag comparison in a n-way set-associative cache. If any $\texttt{R}_{\texttt{bop-pc}}$ 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 $\texttt{R}_{\texttt{op}}$) 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 ($\texttt{R}_{\texttt{mask}}$) as the value of $\texttt{R}_{\texttt{mask}}$ 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.

	Simulator	**FPGA**
ISA	64-bit Alpha	64-bit RISC-V v2
Architecture	Single-Issue In-Order, 1GHz	Single-Issue In-Order, 50MHz (Synthesized)
Pipeline	Fetch1/Fetch2/Decode/Execute (4 stages)	Fetch/Decode/Execute/Memory/Writeback (5 stages)
Branch Predictor	Tournament predictor (512-entry for global and 128-entry for local) 256-entry, 2-way BTB with RR replacement policy 8-entry return address stack 3-cycle branch miss penalty	32B predictor (128-entry gshare) 62-entry, fully-associative BTB with LRU replacement policy 2-entry return address stack 2-cycle branch miss penalty
Caches	16KB, 2-way, 2-cycle L1 I-cache 32KB, 4-way, 2-cycle L1 D-cache 10-entry I-TLB, 10-entry D-TLB 64B block size with LRU replacement policy	16KB, 4-way, 1-cycle L1 I-cache 16KB, 4-way, 1-cycle L1 D-cache 8-entry I-TLB, 8-entry D-TLB 64B block size with LRU replacement policy
Memory	512MB, DDR3-1600, 2 rank, tCL/tRCD/tRP = 11/11/11	1GB, DDR3-1066, 1 rank, tCL/tRCD/tRP = 7/7/7
Workloads	Lua-5.3.0, JavaScript (SpiderMonkey-17)	Lua-5.3.0

Table II: Architectural parameters

Input script	Input parameter		Description
	Simulator	FPGA	
binary-trees[2]	12		Allocate and deallocate many binary trees
fannkuch−redux	9	11	Indexed-access to tiny integer-sequence
k−nucleotide	250,000	2,500,000	Repeatedly update hashtables and k-nucleotide strings
mandelbrot	250	4,000	Generate Mandelbrot set portable bitmap file
n−body	500,000	5,000,000	Double-precision N-body simulation
spectral−norm	500	3,000	Eigenvalue using the power method
n−sieve	7	8	Count the prime numbers from 2 to M (Sieve of Eratosthenes algorithm)
random	300,000	600,000	Generate random number
fibo	12	32	Calculate Fibonacci number
ackermann	7	10	Use of the Ackermann function to provide a benchmark for computer performance
pidigits	500	6,000	Streaming arbitrary-precision arithmetic

Table III: Benchmarks

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.

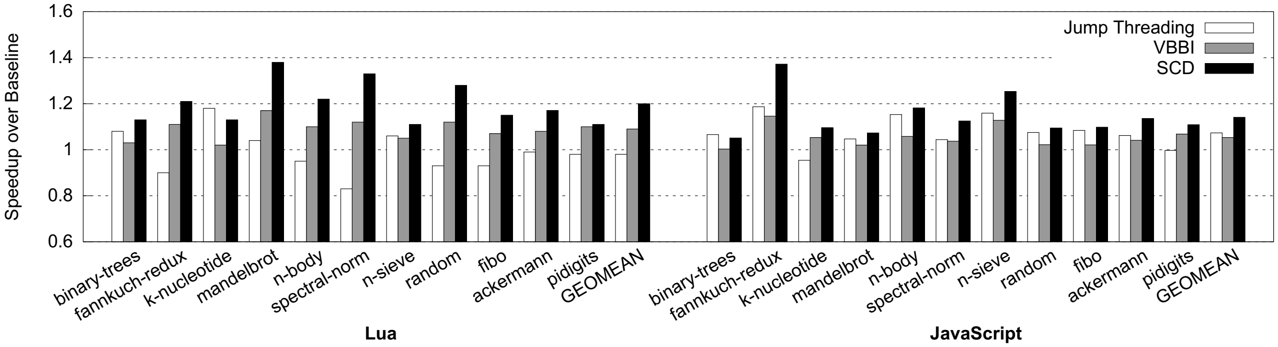


Figure 7: Overall speedups for Lua and JavaScript interpreters (the higher, the better)

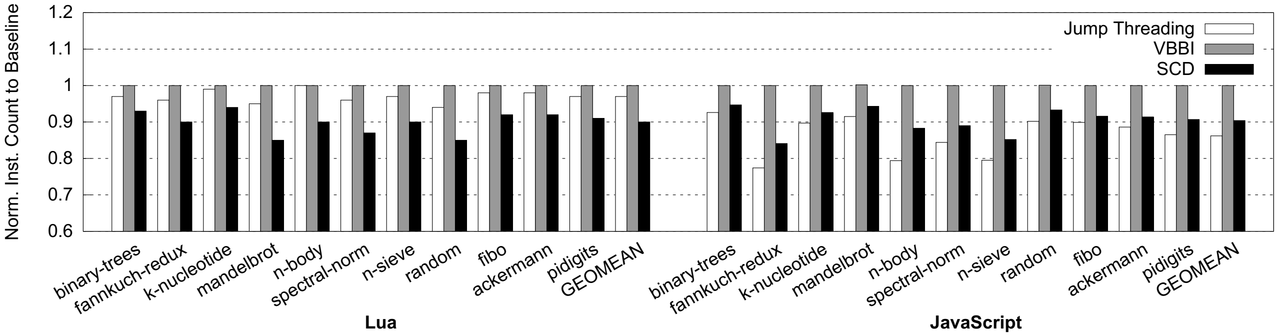


Figure 8: Normalized dynamic instruction count (the lower, the better)

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–dna*). 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 −1.6% 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 miss prediction 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 as the instruction cache miss rate increases from 0.28 MPKI (baseline) to 4.80 MPKI (jump threading). The

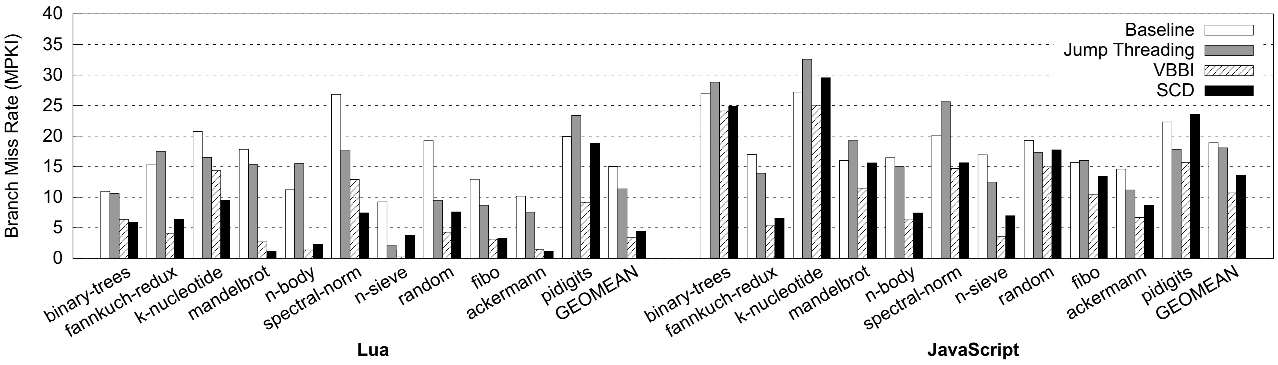


Figure 9: Branch misprediction rate in misses per kilo-instructions (MPKI) (the lower, the better)

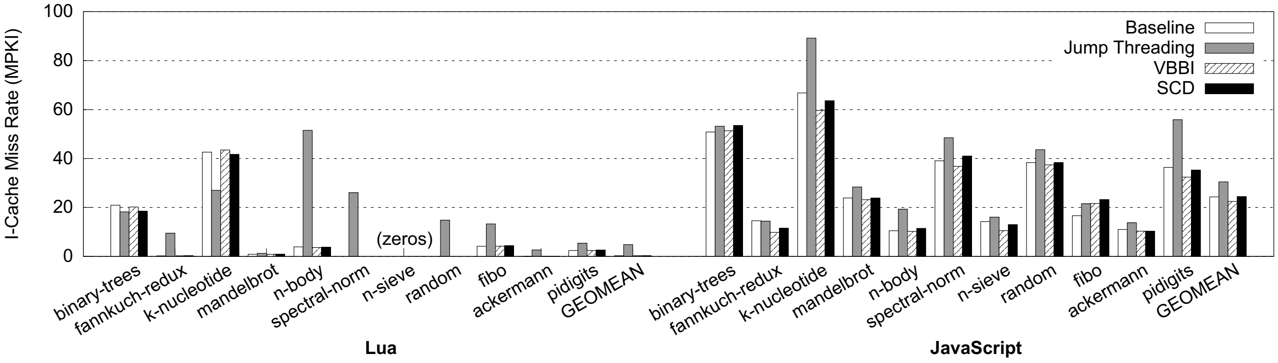


Figure 10: Instruction cache miss rates in misses per kilo-instructions (MPKI) (the lower, the better)

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 geomean 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 geomean 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 geomean 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 geomean 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 ˜3% and ˜7% of total area and power, respectively, which is increased by 21.6% and 11.7% with SCD.

	Baseline		**Jump Threading**		**SCD**		**Jump Thr./Baseline**		**SCD/Baseline**	
Benchmark	Inst. Count	Cycle Count	Inst. Count	Cycle Count	Inst. Count	Cycle Count	Inst. Savings	Speedup	Inst. Savings	Speedup
binary−trees	2.4B	3.8B	2.3B	3.6B	2.2B	3.4B	5.39%	4.33%	9.06%	10.43%
fannkuch−redux	595.4B	737.5B	565.9B	796.3B	534.1B	631.5B	5.22%	-7.38%	11.48%	16.78%
k−nucleotide	88.7B	155.3B	85.4B	159.0B	82.4B	142.9B	3.92%	-2.30%	7.69%	8.67%
mandelbrot	218.3B	263.0B	210.5B	249.6B	185.1B	214.4B	3.69%	5.37%	17.95%	22.67%
n−body	267.4B	341.2B	260.1B	345.4B	246.1B	313.1B	2.77%	-1.21%	8.65%	8.97%
spectral−norm	507.2B	711.2B	488.1B	702.3B	461.1B	627.6B	3.91%	1.27%	10.00%	13.87%
n−sieve	4.0B	5.4B	3.8B	6.1B	3.6B	4.9B	6.27%	-11.13%	10.95%	12.03%
random	0.8B	1.1B	0.8B	1.1B	0.7B	1.0B	7.21%	3.97%	11.92%	13.01%
fibo	3.5B	4.8B	3.3B	4.5B	3.2B	4.3B	7.49%	5.93%	10.56%	11.48%
ackermann	23.4B	34.5B	22.3B	32.8B	21.3B	31.6B	5.02%	5.32%	10.09%	9.30%
pidigits	578.2B	796.3B	564.3B	816.2B	541.2B	750.2B	2.46%	-2.45%	6.83%	6.14%
GEOMEAN							4.84%	0.01%	10.44%	12.04%

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

	Baseline				**SCD**			
Module Hierarchy	**Area (mm^2)**		**Power (mW)**		**Area (mm^2)**		**Power (mW)**	
Top	**0.690**	**100.0%**	**18.46**	**100.0%**	**0.695**	**100.0%**	**18.66**	**100.0%**
- Tile	0.649	94.0%	14.66	79.4%	0.654	94.0%	14.86	79.6%
| - Core	0.044	6.3%	2.86	15.5%	0.044	6.3%	2.87	15.4%
| | - CSR	0.013	1.9%	1.07	5.8%	0.013	1.9%	1.07	5.7%
| | - Div	0.006	0.9%	0.17	0.9%	0.006	0.9%	0.17	0.9%
| - FPU	0.087	12.7%	3.19	17.3%	0.088	12.7%	3.21	17.2%
| - ICache	0.251	36.4%	3.58	19.4%	0.255	36.7%	3.75	20.1%
| | **- BTB**	**0.019**	**2.7%**	**1.40**	**7.6%**	**0.023**	**3.3%**	**1.56**	**8.4%**
| | - Array	0.229	33.2%	1.91	10.3%	0.229	32.9%	1.91	10.2%
| | - ITLB	0.003	0.5%	0.28	1.5%	0.003	0.5%	0.27	1.5%
| - DCache	0.248	36.0%	3.70	20.0%	0.248	35.7%	3.70	19.8%
| - Uncore	0.018	2.6%	1.34	7.2%	0.018	2.6%	1.33	7.2%
| | - HTIF	0.006	0.8%	0.41	2.2%	0.006	0.8%	0.41	2.2%
| | - Memsys/L2Hub	0.012	1.8%	0.92	5.0%	0.012	1.8%	0.92	4.9%
- Wrapping	0.041	6.0%	3.80	20.6%	0.042	6.0%	3.80	20.4%

Table V: Hardware overhead breakdown (area, power)

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. Figure 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 geomean speedups of 17.6% and 15.2% for Lua and SpiderMonkey, respectively, and reduced instruction counts by 10.2% and 9.2%.

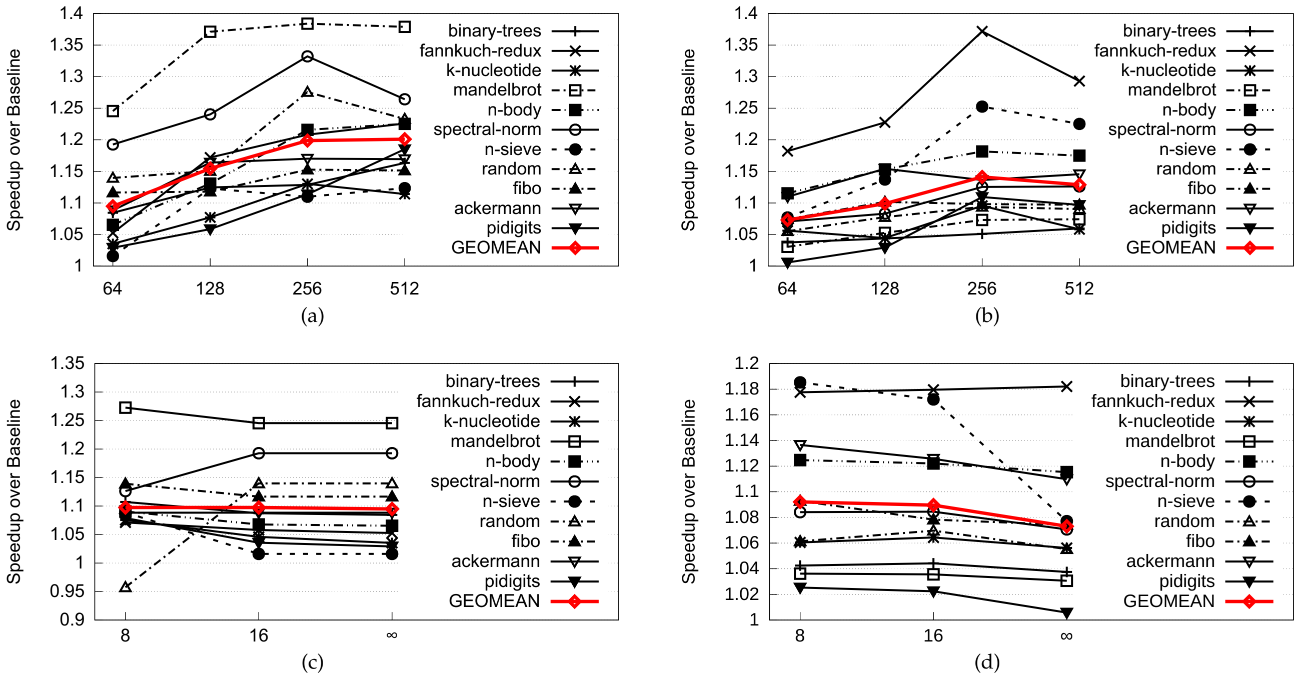


Figure 11: Sensitivity of overall speedups to varying BTB size (a) for Lua and (b) SpiderMonkey. Sensitivity to the the maximum cap imposed on the number of JTEs (c) for Lua and (d) SpiderMonkey

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 el al. [9] propose VBBI, which demonstrates the effectiveness of value correlation for indirect branches. Seznec and Michaud [37] propose the ITTAGE 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 interpreters to reduce dispatch branch cost on a VLIW processor. However, these techniques require a large number of registers, not suitable in the embedded domain. Berndl et al. [15] propose context threading, which completely eliminates indirect branches from the dispatch of virtual machine instructions via JIT compilation, requiring a significant number of CPU cycles. Jia et al. [39] propose Direct TPC Table (DTT), a program structure-aware indirect branch handling technique replacing target calculation with a table lookup in software. However, the software table lookup itself takes a significant fraction of total execution time in the VM interpreter. Another class of techniques reduce the cost of indirect branches via code transformation, including bytecode replication and superinstructions [16], Compiler-guided Value Pattern (CVP) [18], techniques by Mccandless and Gregg [17], and so on. These techniques still do not reduce the redundant computation in the dispatcher code, hence yielding limited gains for VM interpreters on embedded processors.

VIII. Conclusion

This paper proposes 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

[1] "Node.js - open-source runtime environment for developing server-side web applications." https://nodejs.org

[2] "WebOS - linux kernel-based multitask operating system for smart devices." http://www.openwebosproject.org

[3] "TIZEN." https://www.tizen.org

[4] "Python." https://www.python.org

[5] "Ruby." https://www.ruby-lang.org

[6] "PHP." https://secure.php.net

[7] "The Programming Language Lua." http://www.lua.org

[8] E. Rohou, B. N. Swamy, and A. Seznec, "Branch Prediction and the Performance of Interpreters: Don'T Trust Folklore," in *Proc. of CGO*, 2015.

[9] M. Farooq, L. Chen, and L. John, "Value Based BTB Indexing for indirect jump prediction," in *Proc. of HPCA*, 2010.

[10] H. Kim *et al.*, "VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-based Dynamic Devirtualization," in *Proc. of ISCA*, 2007.

[11] P.-Y. Chang, E. Hao, and Y. N. Patt, "Target Prediction for Indirect Jumps," in *Proc. of ISCA*, 1997.

[12] K. Driesen and U. Hölzle, "Accurate Indirect Branch Prediction," in *Proc. of ISCA*, 1998.

[13] K. Driesen and U. Hölzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction," in *Proc. of MICRO*, 1998.

[14] A. Seznec, "A New Case for the TAGE Branch Predictor," in *Proc. of MICRO*, 2011.

[15] M. Berndl *et al.*, "Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters," in *Proc. of CGO*, 2005.

[16] M. A. Ertl and D. Gregg, "Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters," in *Proc. of PLDI*, 2003.

[17] J. Mccandless and D. Gregg, "Compiler Techniques to Improve Dynamic Branch Prediction for Indirect Jump and Call Instructions," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, Jan. 2012.

[18] M. Tan *et al.*, "CVP: An Energy-efficient Indirect Branch Prediction with Compiler-guided Value Pattern," in *Proc. of ICS*, 2012.

[19] T. Oh *et al.*, "Practical Automatic Loop Specialization," in *Proc. of ASPLOS*, 2013.

[20] "ARM Cortex M series." http://www.arm.com/products/processors/cortex-m

[21] "Arduino. an open-source electronics platform based on easy-to-use hardware and software." https://www.arduino.cc

[22] "Raspberry Pi." https://www.raspberrypi.org

[23] "Intel Galileo." https://software.intel.com/en-us/iot/hardware/galileo

[24] "Raspberry Pi. Usage documentation of Python." https://www.raspberrypi.org/documentation/usage/python

[25] "Espruino. JavaScript on board." http://www.espruino.com

[26] "Tessel 2." https://tessel.io

[27] "TI LaunchPad." http://www.ti.com/ww/en/launchpad/launchpad.html

[28] D. R. Kaeli and P. G. Emma, "Improving the Accuracy of History-Based Branch Prediction," *IEEE Trans. Comput.*, vol. 46, no. 4, Apr. 1997.

[29] "SpiderMonkey 17." https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Releases/17

[30] S.-W. Lee and S.-M. Moon, "Selective just-in-time compilation for client-side mobile javascript engine," in *Proc. of CASES*, 2011.

[31] P. Ratanaworabhan *et al.*, "JSMeter: Characterizing Real-World Behavior of JavaScript Programs," Tech. Rep. MSR-TR-2009-173, Dec. 2009.

[32] N. Binkert *et al.*, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, Aug. 2011.

[33] P. Gavin, D. Whalley, and M. Själander, "Reducing Instruction Fetch Energy in Multi-issue Processors," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, Dec. 2013.

[34] K. Asanović *et al.*, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr. 2016.

[35] "Computer Language Benchmarks Game." http://benchmarksgame.alioth.debian.org

[36] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer*, vol. 17, no. 1, Jan. 1984.

[37] A. Seznec and P. Michaud, "A case for (partially) TAgged GEometric history length branch prediction," *Journal of Instruction Level Parallelism*, vol. 8, Feb. 2006.

[38] J. Hoogerbrugge *et al.*, "A Code Compression System Based on Pipelined Interpreters," *Softw. Pract. Exper.*, vol. 29, no. 11, Sep. 1999.

[39] N. Jia *et al.*, "DTT: Program Structure-aware Indirect Branch Optimization via direct-TPC-table in DBT System," in *Proc. of CF*, 2014.