Typed Architectures: Architectural Support for Lightweight Scripting

Channoh Kim[1*] Jaehyeok Kim[1*] Sungmin Kim[1] Dooyoung Kim[1] Namho Kim[2]
Gitae Na[1] Young H. Oh[1] Hyeon Gyu Cho[1] Jae W. Lee[2]

[1]Sungkyunkwan University, Suwon, Korea [2]Seoul National University, Seoul, Korea
{channoh, max250, vash4h, dooyoungid, hire1021, younghwan, cho42me}@skku.edu
{kkjknh2, jaewlee}@snu.ac.kr

Abstract

Dynamic scripting languages are becoming more and more widely adopted not only for fast prototyping but also for developing production-grade applications. They provide high-productivity programming environments featuring high levels of abstraction with powerful built-in functions, automatic memory management, object-oriented programming paradigm and dynamic typing. However, their flexible, dynamic type systems easily become the source of inefficiency in terms of instruction count, memory footprint, and energy consumption. This overhead makes it challenging to deploy these high-productivity programming technologies on emerging single-board computers for IoT applications.

Addressing this challenge, this paper introduces *Typed Architectures*, a high-efficiency, low-cost execution substrate for dynamic scripting languages, where each data variable retains high-level type information at an ISA level. Typed Architectures calculate and check the dynamic type of each variable *implicitly* in hardware, rather than explicitly in software, hence significantly reducing instruction count for dynamic type checking. Besides, Typed Architectures introduce *polymorphic instructions* (e.g., `xadd`), which are bound to the correct native instruction at runtime within the pipeline (e.g., `add` or `fadd`) to efficiently implement polymorphic operators. Finally, Typed Architectures provide hardware support for flexible yet efficient type tag extraction and insertion, capturing common data layout patterns of tag-value pairs. Our evaluation using a fully synthesizable RISC-V RTL design on FPGA shows that Typed Architectures achieve geomean speedups of 11.2% and 9.9% with maximum speedups of 32.6% and 43.5% for two production-grade scripting engines for JavaScript and Lua, respectively. Moreover, Typed Architectures improve the energy-delay product (EDP) by 19.3% for JavaScript and 16.5% for Lua with an area overhead of 1.6% at a 40nm technology node.

CCS Concepts • **Computer systems organization** → **Serial architectures**; • **Software and its engineering** → **Scripting languages**

Keywords Instruction Set Architecture, Microarchitecture, Pipeline, Internet of Things (IoT), Interpreters, Performance, Scripting languages, JavaScript, Lua, Type checking

1. Introduction

Dynamic scripting languages are being widely used for a variety of complex applications. For example, Lua [28] is a lightweight scripting language adopted for game programming (e.g., World of Warcraft [22] and Angry Birds [1]) and writing plug-ins (e.g., Adobe's Photoshop Lightroom [13]). JavaScript [7] is the default language for programming the web, enabling billions of web pages. Python [14], Ruby [19], R [15], MATLAB [10], and Perl [12] are also popular in various application domains. They provide high-productivity programming environments featuring high levels of abstraction, automatic memory management, object-oriented programming paradigm, and dynamic typing.

These scripting languages are also being embraced by emerging single-board computers for so-called DIY electronics, such as Raspberry Pi [16], Arduino [24], and Intel's Galileo and Edison [3, 5], to name a few. For example, Raspberry Pi promotes Python as the main programming language [18]; Arduino offers a variant based on JavaScript [4]; Galileo and Edison provide a JavaScript-based IDE through Intel XDK [23]. These computers feature small form factors, low cost, and open-source software stack to make them an ideal platform for Internet-of-Things (IoT) applications. Scripting languages can provide many productivity benefits for IoT programming: ease of testing (e.g., JavaScript runnable on any web browser), natural support for event-

* These authors contributed equally to this work.

ASPLOS '17, April 08 - 12, 2017, Xi'an, China

ISBN 978-1-4503-4465-4/17/04...$15.00

DOI: http://dx.doi.org/10.1145/3037697.3037726

	SAMA5D3 [20] (Atmel)	**Galileo Gen 2** [5] (Intel)	**Arduino Yun** [2] (Atmel)	**LaunchPad** [8] (TI)	**ARM mbed** [29] (STMicro)
Processor	ARM Cortex-A5	Intel Quark SoC X1000	MIPS 24K	ARM Cortex-M4	ARM Cortex-M0
ISA	ARMv7-A	x86 (IA32)	MIPS32	ARMv7-M	ARMv6-M
Clock Frequency	536MHz	400MHz	400MHz	80MHz	48MHz
L1 Cache	64KB	16KB	0 $\sim$ 64KB	-	-
Main Memory	256MB DDR2 DRAM	256MB DDR3 DRAM	64MB DDR2 DRAM	32KB SRAM	8KB SRAM
Flash Memory	256MB	8MB	16MB	256KB	32KB
OS	Linux	Yocto Linux	Linux (OpenWrt)	TI RTOS	ARM mbed OS
Power	0.25 $\sim$ 1.85W	2.6 $\sim$ 4W	700 $\sim$ 1500mW	75 $\sim$ 225mW	100 $\sim$ 110mW
Price (2016)	$159	$64.99	$74.95	$12.99	$10.32

Table 1: IoT device platforms

driven programming model (e.g., for sensors), seamless web-based integration with servers (using JavaScript and Node.js [11]), and mature user community.

However, scripting languages are still much slower than native programming languages (e.g., C/C++), to make it impractical to use them for developing production-grade applications on IoT devices. Table 1 summarizes some of the IoT devices available today [2, 5, 8, 20, 29]. Typically, those devices have a single-core, in-order processor with small main memory whose size ranges from tens of kilobytes to hundreds of megabytes. Many devices in this class cannot afford the high runtime cost of a scripting language. Conventional dynamic code optimization techniques (e.g., Just-In-Time (JIT) compilation) may not be viable, either, as the JIT compiler itself requires a significant amount of additional memory space and CPU cycles. As a result, many scripting engines running on these devices (e.g., Lua [28], Duktape [26]) employ interpreter-based virtual machines (VMs) without JIT compilation.

Dynamic types are one of the major sources of inefficiency for scripting languages. With dynamic typing programmers do not have to specify statically the exact type of a variable, and an object can be easily extended to integrate new fields and methods or override existing ones during execution. However, this flexibility comes with a cost. Since type checking and method dispatch are performed with regard to a given program input at runtime, each variable must carry a *type tag*, and a *type guard* must be executed before any operation that can be overloaded. This significantly increases dynamic instruction count, memory footprint, and hence energy consumption, compared to statically typed languages. A recent study [39] estimates about 25% of total execution time spent for dynamic type checking on the V8 JavaScript engine.

While hardware support for dynamic type checking dates back to 1970s-80s (e.g., LISP machines), existing proposals have either limited applicability targeting a specific language [30, 33, 46, 50, 52], or a relatively narrow coverage of type checking operations [30, 39], or both. For lightweight scripting on resource-constrained IoT devices, low-cost hardware-based acceleration of type checking with broad coverage is the key.

To address this, we propose *Typed Architectures*, a high-efficiency, low-cost execution substrate for dynamic scripting languages. The key idea of Typed Architectures is to retain the high-level type information for each variable at an ISA level. Then dynamic type checking is performed *implicitly* within the pipeline in parallel with instruction execution, not explicitly in software, hence significantly reducing dynamic instruction count. Besides, Typed Architectures introduce polymorphic instructions, which are bound to the correct native instruction at runtime based on the operand types. Finally, Typed Architectures provide hardware support for flexible type tag extraction and insertion to accelerate multiple scripting engines with different data layouts for tag-value pairs.

We evaluate our proposal by running two production-grade scripting engines for Lua and JavaScript on FPGA, using a synthesizable RTL model based on RISC-V Rocket Core [17]. Typed Architectures achieve maximum speedups of 43.5% and 32.6% with geomean speedups of 9.9% and 11.2% for Lua and JavaScript, respectively. This compares favorably to 7.3% and 5.4% geomean speedups by a state-of-the-art hardware-based type checking technique [30]. Moreover, our synthesis results using a TSMC 40nm standard cell library report only a 1.6% increase in chip area, while improving the energy-delay product (EDP) by 16.5% for Lua and 19.3% for JavaScript, respectively.

In summary this paper makes the following contributions:

- We propose a novel ISA extension to efficiently manage type tags in hardware, which can be flexibly applied to multiple scripting languages and engines.
- We design and implement the Typed Architecture pipeline, which effectively reduces the overhead of dynamic type checking at low hardware cost.
- We prototype the proposed processor architecture using a fully synthesizable RTL model to execute two production-grade scripting engines with large inputs on FPGA (executing over 274 billion instructions in total) and provide a more accurate estimate of area and power using a TSMC 40nm standard cell library.

2. Motivation

Figure 1 illustrates the usages and implementation of bytecode `ADD`, which requires type guards. The "+" operator is polymorphic, hence it must be properly guarded to invoke the right version of the operator function depending on the

```
1  function add(x,y) return x+y end
2  add(1,2)        --(INTEGER) 3
3  add(1,2.2)      --(FLOAT) 3.2
4  add(1.1,2)      --(FLOAT) 3.1
5  add(1.1,2.2)    --(FLOAT) 3.3
6  add("1","2")    --(FLOAT) 3.0
7  add("a","b")    --error
```

(a)

```
1  for (;;) {
2   // dispatch next bytecode
3   Bytecode bc = *(VM.pc++);
4   switch(OPCODE(bc)) {
5      ...
6    case ADD:
7     // load pointers of RB and RC to rb and rc
8     Value *rb = RB(bc); Value *rc = RC(bc);
9     Number nb, nc;
10    // check if rb and rc is integer
11    // if so, calculate sum and set type
12    if(isInt(rb) && isInt(rc)){
13     type(ra) = INTEGER;
14     ival(ra) = ival(rb) + ival(rc);
15    // convert rb and rc to float
16    // if successful, caculate sum and set type
17    }else if(toNumber(rb,&nb)&&toNumber(rc,&nc)){
18     type(ra) = FLOAT;
19     fval(ra) = nb + nc;
20    }else{
21     /* handle exception cases */
22    }
23   break;
24  }
25 }
```

(b)

```
1  ADD:
2   ## load pointers of ra, rb and rc to s14, s10 and s9
3   ...
4  ADD_fast:
5  isInt_Rb:
6   lw      a2,8(s10)         # load type(rb)
7   li      a4,19             # check if type(rb) is int
8   bne     a2,a4,isFloat_Rb  # if not, jump to isFloat_Rb
9  isInt_Rc:
10  lw      a5,8(s9)          # load & check type(rc) is int
11  bne     a5,a4,isFlt_Rc    # if not, jump to isFlt_Rc
12 ADD(int, int):
13  ld      a2,0(s10)         # load ival(rb)
14  ld      a5,0(s9)          # load ival(rc)
15  add     a5,a5,a2          # add rb and rc
16  sw      a4,8(s14)         # store type(ra)
17  sd      a5,0(s14)         # store value(ra) to mem
18  j       .loop_header      # return to loop header
19 isFloat_Rb:
20  li      a4,3              # check if type(rb) is float
21  bne     a2,a4,ADD_slow    # if not, jump to ADD_slow
22 isFloat_Rc:
23  lw      a5,8(s9)          # load & check type(rc) is float
24  bne     a5,a4,ADD_slow    # if not, jump to ADD_slow
25 ADD(flt, flt):
26  fld     f2,0(s10)         # load fval(rb)
27  fld     f5,0(s9)          # load fval(rc)
28  fadd.d  f5,f5,f2          # add rb and rc
29  sw      a4,8(s14)         # store type(ra)
30  fsd     f5,0(s14)         # store value(ra) to mem
31  j       .loop_header      # return to loop header
32 isFlt_Rc:
33  li      a4,3              # check if type(rc) is float
34  beq     a5,a4,cvt_Rb      # if then, jump to cvt_Rb
35 ADD_slow:
36  ## convert rb and rc to float
37  ...
```

(c)

Figure 1: (a) Usages of polymorphic ”+” (add) operator in Lua; (b) Bytecode `ADD` in C; (c) RISC-V assembly code

operand types. Before adding two operators, say `x` and `y` in Figure 1(a), their types must be checked. Figure 1(b) shows an excerpt from the bytecode interpreter loop in Lua [28], which fetches and executes one bytecode at a time. Since the `ADD` operation is polymorphic, type guards (shaded in gray) must be executed to bind to the correct function. Figure 1(c) shows an RISC-V assembly code of the bytecode generated by `gcc -O3`. The overhead of dynamic type checking is known to be significant and can be broken down into the following components:

- *Tag extraction*: The type tag of an operand should be extracted for a given value (e.g., Line 6 in the RISC-V assembly) for type checking. Although a load instruction suffices in this example, it may take additional instructions to extract this field (e.g., shift and mask).
- *Tag checking*: Type checking examines the types of the input operands and dispatches to the right version of the operation being performed. Typically, this is realized by multiple type guards, each of which consists of type tag comparison followed by a conditional branch (e.g., Line 12 in Figure 1(b), which corresponds to Lines 5-11 in Figure 1(c)).
- *Tag insertion*: This is an inverse operation of type tag extraction. When a new value is produced, the type tag must be stored together with it (e.g., Line 13 in Figure 1(b)). Again, it may take additional instructions to insert this field like shift and mask.

Figure 2(a) shows the breakdown of dynamic bytecodes for 11 Lua scripts. While Lua defines 47 distinct bytecodes, a small number of bytecodes ($<$10) dominates the total dynamic bytecode count. Among them Figure 2(b) shows the number of dynamic instructions per bytecode for the five most frequently used bytecodes: `ADD`, `SUB`, `MUL`, `GETTABLE`, and `SETTABLE`. Note that `GETTABLE` and `SETTABLE` are used for table lookup and update, respectively, and can use both an integer and a string as key. Since the five bytecodes are all polymorphic, they require type guards to select the right function depending on the operand types. In Figure 2(b) we count instructions separately for different type pairs as their numbers of instructions for type guards may vary. A significant fraction of instructions are spent executing type guards for the five bytecodes. According to Dot et al. [39] type guards account for about 25% of total execution time for the V8 engine.

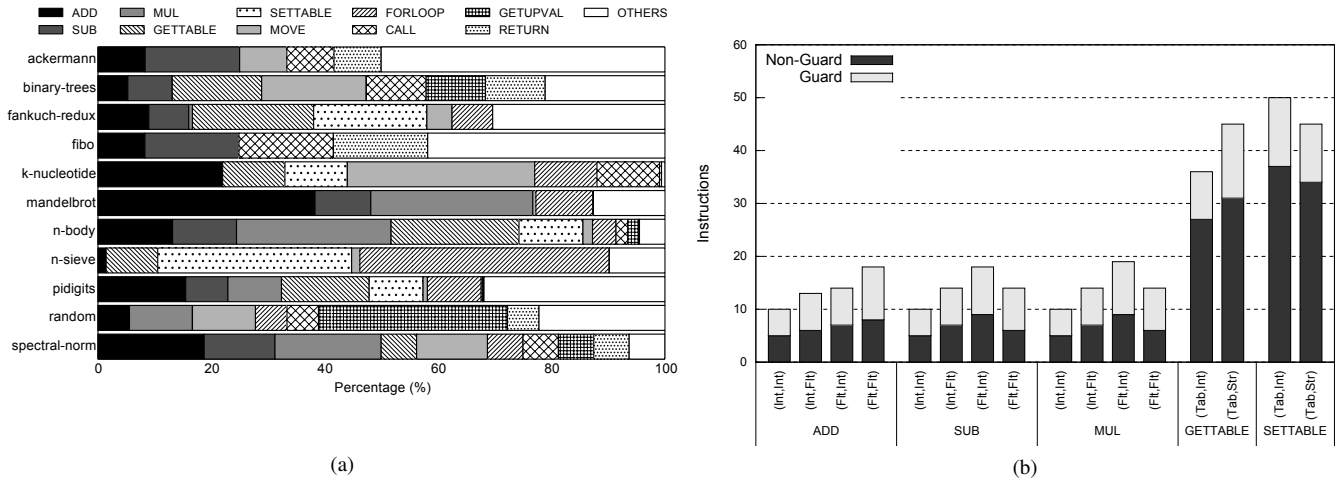

Figure 2: (a) Breakdown of dynamic bytecodes in Lua; (b) Dynamic instruction count per bytecode for top five bytecodes

Hardware support for type checking can reduce this overhead. Existing techniques mostly focus on introducing complex instructions that effectively capture common patterns of the type checking code [30, 33, 46, 50, 52]. In contrast, our approach bridges the semantic gap between bytecodes and the native hardware instructions by (1) maintaining the high-level operand type at an ISA level for hardware-accelerated type checking and (2) raising the abstraction level of the native instructions with polymorphic instructions to implement polymorphic bytecodes more efficiently.

3. Typed Architectures

Low-cost, high-coverage architectural support for dynamic typing is the key to enable efficient scripting on resource-constrained IoT processors, where JIT compilation may not be viable. We have the following three design objectives for Typed Architectures:

- *High performance*: Significant speedups should be achieved by offloading type checking operations to hardware.
- *Flexibility*: The ISA extension should be flexible enough to support multiple, production-grade scripting engines.
- *Low cost*: The proposed mechanism should incur minimal cost in terms of area and power.

3.1 ISA Extension

Typed Architecture extends the baseline RISC ISA with the following three capabilities: unified register file, tagged ALU instructions, and tagged memory instructions. The rest of this section describes each of them in details.

Unified register file. Typed Architectures extend the register file with two new fields: 8-bit type and 1-bit F/$\bar{I}$ fields. The type field stores the type tag for a value as defined by the script engine. The 8-bit type field can represent 256 distinct types, and we believe it can accommodate most cases, possibly with simple re-encoding of type tags to fit in 8 bits. The F/$\bar{I}$ bit is a flag indicating whether the value is of an integer subtype (0) or floating-point (FP) subtype (1). This bit can be provided either by software (e.g., extending the original type field by one bit) or by hardware (e.g., registering a specific set of type values in a table to identify FP subtypes). With this extension a register entry has three fields, value, type, and F/$\bar{I}$ bit, which are denoted by `R.v`, `R.t`, and `R.f`, respectively. This register file is *unified* as it can hold both integer and FP values.

Tagged ALU instructions. Typed Architectures introduce three tagged ALU instructions (`xadd`, `xsub`, and `xmul`) to perform type checking in parallel with value calculation within the pipeline, as well as a Handler Register (R_{hdl}) for handling type mispredictions. When a tagged ALU instruction is executed, Typed Architecture looks up *Type Rule Table* with the two source type tags and the instruction's opcode as key. If it hits, the pipeline executes normally to write back the output type tag retrieved from the Type Rule Table together with the output value to the destination register. If not, a type misprediction has happened, the PC is redirected to the slow path pointed to by R_{hdl} to go through the original software-based type checking. We assume the Type Rule Table is pre-loaded only once at program launch.

The three instructions with a prefix `x` (`xadd`, `xsub`, `xmul`) are *polymorphic* instructions. These instructions are bound to the correct native instruction at runtime according to the types of the source operands. For example, an `xadd` instruction is bound to (integer) `add` instruction if both operands are integers, or to (FP) `fadd` instruction if both operands are FP values; otherwise, it will jump to the slow path pointed to by R_{hdl}.

Tagged memory instructions. Typed Architectures introduce two instructions for memory operations: `tld` (tagged load) and `tsd` (tagged store). `tld` not only loads a requested value from memory but also its type tag and F/$\bar{I}$ bit. `tsd` works similarly, but to the opposite direction. A type tag is extracted from an adjacent 64-bit double-word (or the

Instruction	Operation	Type Tag Handling	Description
Memory Instructions			
tld Rc, imm(Ra)	Rc.v ← Mem[Ra.v+imm]	Rc.t ← extract(Mem[Ra.v+imm+R_{offset}])	Load dword with tag
tsd Rc, imm(Ra)	Mem[Ra.v+imm] ← Rc.v	Mem[Ra.v+imm+R_{offset}] ← insert(Rc.t)	Store dword with tag
Arithmetic and Logical Instructions			
xadd Ra,Rb,Rc	$Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} + Rc.v_{[63:0]}$	**if** (type hits)	Add (dword)
xsub Ra,Rb,Rc	$Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} - Rc.v_{[63:0]}$	Rc.t ← OutputType(op, Ra.t, Rb.t)	Subtract (dword)
xmul Ra,Rb,Rc	$Ra.v_{[63:0]} \leftarrow Rb.v_{[63:0]} \times Rc.v_{[63:0]}$	**else** NextPC ← R_{hdl}	Multiply (dword)
Configuration Instructions			
setoffset Ra	R_{offset} ← Ra.v	-	Set Ra to R_{offset}
setmask Ra	R_{mask} ← Ra.v	-	Set Ra to R_{mask}
setshift Ra	R_{shift} ← Ra.v	-	Set Ra to R_{shift}
set_trt Ra	TypeRuleTable.push.data(Ra.v)	-	Push Ra to Type Rule Table (TRT)
flush_trt	TypeRuleTable.flush()	-	Flush TRT
Miscellaneous			
thdl label	R_{hdl} ← NextPC + (disp << 2)	-	Store `label` addr to R_{hdl}
tchk Rb,Rc	-	NextPC ← (type hits) ? PC + 4 : R_{hdl}	Look up TRT with two source type tags (Ra.t and Rb.t)
tget Ra,Rb	Ra.v ← ZeroExt64(Rb.t)	-	Copy the type field of Rb to Ra.v (64-bit zero extended)
tset Ra,Rb	Rb.t ← $Ra.v_{[7:0]}$	-	Copy the least significant byte of Ra.v to Rb.t

Table 2: Description of Extended ISA (64-bit)

same double-word with the value) by applying shift-and-mask. Since the exact location and length of a type tag is implementation-specific, Typed Architectures introduce three special-purpose registers to flexibly control tag extraction and insertion.

- R_{offset} (Offset Register): This register holds a 3-bit flag divided into two fields. The two LSBs indicate which double-word the tag will be extracted from. Since script engines commonly place a value-tag pair close to each other, we only offer three choices: next double-word (`01`), previous double-word (`11`), or the same double-word with the value (`00`). The MSB controls whether Not-a-Number (NaN) detection for a FP number is enabled or not. Some scripting engines exploit NaN values to represent non-FP values as well as FP values using a single 64-bit double-word. The SpiderMonkey JavaScript engine is such an example, and we discuss more about how it uses the NaN detection mechanism in Section 4.2.
- R_{shift} (Shift Amount Register): This register encodes the starting position (bit) of the type field within the double-word indicated by the R_{offset}. R_{shift} takes 6 bits to point to any of the 64 bits within the double-word.
- R_{mask} (Mask Register): This register holds an 8-bit mask to extract a type tag of the same width. Typically, the value of R_{offset}, R_{mask}, and R_{shift} is set only once at initialization.

Miscellaneous instructions. Finally, Typed Architecture add four more instructions: `thdl`, `tchk`, `tget`, and `tset`. `thdl` sets the value of R_{hdl}, which points to the starting address of a type miss handler (i.e., slow path). `tchk` only performs type checking without value calculation. It looks up the Type Rule Table with the two source operand types and the opcode (i.e., `tchk`) as key. If it hits, the program proceeds to the next instruction; if not, it jumps to the slow path. `tget` and `tset` are used to read the type tag of a register (`tget`) or write to it (`tset`). With these instructions we can explicitly manipulate register type tags.

```
1  case: ADD
2    Value *rb = RB(bc); Value *rc = RC(bc);
3    uint64_t tmp1, tmp2;
4    asm volatile(
5      "tld   %0,0(%2)\n\t"      // load rb (v,t)
6      "tld   %1,0(%3)\n\t"      // load rc (v,t)
7      "thdl  ADD_slow\n\t"      // set err handler
8      "xadd  %0,%1,%0\n\t"      // ra = rb + rc
9      "tsd   %0,0(%4)\n\t"      // store ra(v,t)
10     "j     .loop_header\n\t" // go to loop header
11     : "=&r"(tmp1), "=&r"(tmp2)
12     : "r"(rb), "r"(rc), "r"(ra)
13     : "memory"
14   );
15 ADD_slow:
16   Number nb, nc;
17   // convert rb and rc to float
18   // if successful, calculate sum and set type
19   if(toNumber(rb,&nb) && toNumber(rc,&nc)){
20     type(ra) = FLOAT;
21     fval(ra) = nb + nc;
22   }else{
23     /* handle exception cases*/
24   }
25   break;
```

Figure 3: Transformed bytecode ADD

Figure 3 illustrates how the original bytecode ADD in Figure 1(b) is transformed with the ISA extension using inline assembly. The modified lines are shown in gray. In Line 5 and 6, Typed Architecture loads the type-value pair of `rb` and `rc`, respectively. Before calculation, R_{hdl} is set to the address of `ADD_slow` using `thdl` in Line 7. `xadd` looks up the Type Rule Table with the source operand types and opcode as key (e.g., (`Int`, `Int`, `ADD`)). If it hits, `xadd` executes the addition of source registers (e.g., `add` instruction) and the type tag of destination register is set to the output from the Type Rule Table (e.g., `Int`). If not, `PC` is set to the address held by R_{hdl} (e.g., `ADD_slow`). Typed Architecture stores a value-type pair into memory by using `tsd` in Line 9. Table 2 summarizes the operations of the extended ISA.

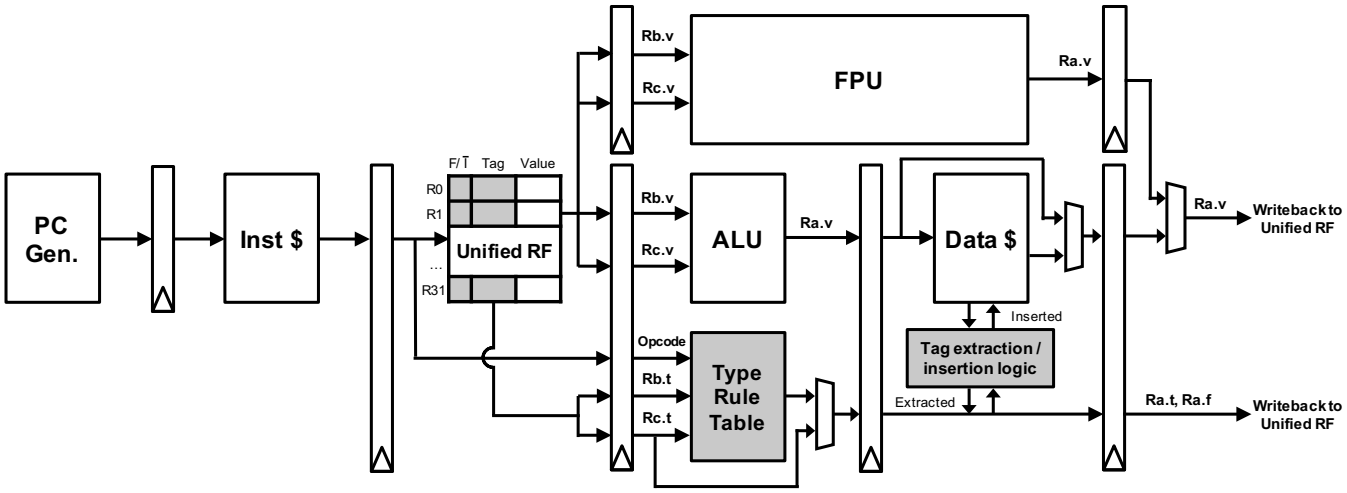

Figure 4: Pipeline structure augmented with Typed Architecture

3.2 Pipeline Organization

Figure 4 shows a pipeline structure that implements the extended ISA introduced in Section 3.1. We add a unified register file, a Type Rule Table, and a tag extract/insert logic to the baseline (shaded in gray). The unified RF replaces the original untyped register file, since the value from an untyped load instruction can be simply marked *untyped* to bypass type checking. When a polymorphic ALU instruction is issued, say `xadd`, the instruction is bound to either (integer) `add` or (FP) `fadd` depending on the value of F/$\bar{I}$ bit.

The Type Rule Table is a small content-addressable memory with three inputs and one output. It takes the tags of the two source operands and the opcode of the instruction as input and generates the output type tag as output. The contents of the Type Rule Table are initialized based on the type encoding of the scripting engine being executed.

Datapath for `xadd`, `xsub`, and `xmul`. The execution path of `xadd` is different from static add instructions in three ways. First, `xadd` should select the calculation path at the decode stage: integer ALU or FP ALU. Second, it accesses the Type Rule Table for type checking in hardware. If it hits, the output tag is propagated to writeback stage and finally to the type tag of the destination register. If not, it goes through the type misprediction path. Finally, there are type mispredictions with tagged ALU instructions. This misprediction is different from a system-level exception (e.g., divide-by-zero), which is handled by OS. The type misprediction handler is nothing but the original code with software-based type checking, and there is no need to return to the tagged instruction in question for retry.

If an overflow is detected from execution of a polymorphic instruction, Typed Architecture generates a type misprediction to redirect execution flow to the slow path. It is because, in some engines, a type tag is co-located with the data value within the same double-word and an overflow may corrupt the type tag. If it is safe to ignore this case, we can simply turn off overflow detection to prevent unnecessary switches to the slow path.

3.3 Memory Access Path

The tagged load instruction (`tld`) loads both value and tag into the destination register. However, the data layout for storing a tag-value pair may vary depending on languages and implementations. However, most script engines co-locate a tag-value pair close to each other, and we exploit this in designing reconfigurable tag extraction/insertion logic. To balance flexibility and efficiency, we limit the location of the tag to be located in either one of the two adjacent double-word or the same double-word with the data value. This logic is implemented by combining shift and mask operations, which is reconfigurable using three special-purpose registers: R_{offset}, R_{mask}, and R_{shift}. If the distance between the type tag and the value is two or more double-words, one can first load them using two load instructions, and then use a `tset` instruction to pack them into a single register.

Datapath for `tld`. In most script engines the address of a type tag has a constant offset from that of the associated value. For example, Lua has an 8-byte value followed by a 1-byte tag (i.e., least-significant byte of the next higher double-word). This 1-byte tag is extracted from memory by the extractor. The extractor first selects the double-word that contains the tag using R_{offset}. Then, the double-word is shifted and masked using R_{shift} and R_{mask} to get the tag.

Datapath for `tsd`. Tag insertion of `tsd` is an inverse operation of tag extraction of `tld`. Tags must be extended to fit in memory layout. This process is performed by the tag insertion logic using the same set of the three configuration registers. The insertion logic make tags sign- or zero-extended and shifted to restore the original layout. The shifter logic is shared by both the tag extraction and insertion logic.

VM	Bytecode	Description
Lua	ADD	R(A) := R(B) + R(C)
	SUB	R(A) := R(B) − R(C)
	MUL	R(A) := R(B) × R(C)
	GETTABLE	R(A) := R(B)[R(C)]
	SETTABLE	R(A)[R(B)] := R(C)
SpiderMonkey	ADD	St[-2] := St[-1] + St[-2]
	SUB	St[-2] := St[-1] − St[-2]
	MUL	St[-2] := St[-1] × St[-2]
	GETELEM	St[-2] := St[-2][St[-1]]
	SETELEM	St[-2][St[-1]] := St[-2]

Table 3: Modified bytecodes in Lua and SpiderMonkey

	Lua	SpiderMonkey
R_{offset} (3 bits)	0b001	0b100
R_{shift} (6 bits)	0b000000	0b101111
R_{mask} (8 bits)	0b11111111	0b00001111

Table 4: Special-purpose register settings

4. Code Transformation

This section describes how we apply the ISA extension of Typed Architecture to two popular open-source scripting engines. Based on bytecode profiling, we identify five hot bytecodes, which execute type guards, as summarized in Table 3. `R()` represents a virtual register, where `A`, `B`, and `C` are operand IDs. `St[-1]` and `St[-2]` denote the top of stack (TOS) and the second on stack (SOS), respectively.

The five bytecodes are divided into (1) arithmetic operations (`ADD`, `SUB`, and `MUL`) and (2) table access operations (`GETTABLE` and `SETTABLE` for Lua; `GETELEM` and `SETELEM` for JavaScript). The arithmetic operations do arithmetic calculation using polymorphic instructions if both operands have the same type; otherwise, it goes to the slow path to convert them into a FP type (`Float`) before calculation. The table access operations are used to access an array or a dictionary table. It takes two source operands: table and key. If the type of the key is `Int`, the address of the target element is simply a sum of the base address of the table and the key; if it is `String`, a hash table is used to retrieve the requested element.

4.1 Lua

We use Lua-5.3.0 [28], which is a register-based virtual machine. It has 47 distinct bytecodes. The bytecode in Lua consists of a 6-bit opcode, a 8-bit register field, and two 9-bit register fields. Lua defines eight primitive types (`NIL`, `Boolean`, `Number`, `String`, `Table`, `Function`, `Thread`, and `User data`). Although there is only one number type (`Number`), Lua internally maintains two subtypes (`Int` and `Float`) to optimize the common case of integer arithmetic. A `struct` is defined to store a tag-value pair, where an 8-byte value is followed by a one-byte tag. Thus, a single Lua variable occupies 16 bytes with the remaining 7 bytes unused for alignment. As shown in Table 4, the three special-purpose registers for tag extraction, R_{offset}, R_{mask}, R_{shift} are set to be 0b001, 0xFF, and 0b000000, respectively. We extend the original type tag by one bit to use its MSB as F/$\bar{I}$ bit.

Opcode	$Type_{in1}$	$Type_{in2}$	$Type_{out}$
xadd	Int	Int	Int
	Float	Float	Float
xsub	Int	Int	Int
	Float	Float	Float
xmul	Int	Int	Int
	Float	Float	Float
tchk	Table	Int	Table
	Int	Table	Table

Table 5: Type Rule Table settings for Lua and SpiderMonkey

We retarget the five bytecodes in Table 3 to Typed Architecture. In fact, bytecode ADD in Figure 1(b) is a simplified version the bytecode ADD in Lua, and we omit the code due to their similarity. Likewise, the transformed C code is almost the same as the code shown in Figure 3. We also transform the other two arithmetic bytecodes, `SUB` and `MUL`, in the same manner. Table 5 shows the contents of the Type Rule Table for Lua.

Table access bytecodes consists of two parts: address calculation of an element of the table and element access/update with boundary checking. Before address calculation, the operand types are checked. We only check the common case of the `Table-Int` pair using a `tchk` instruction. If it passes, Typed Architecture calculates the address of the indexed element; if not (e.g., the table is indexed by a `String` key), Typed Architecture jumps to execute the slow path, whose starting address is stored in R_{hdl}.

4.2 SpiderMonkey: JavaScript Engine for FireFox

SpiderMonkey is the default JavaScript engine for the FireFox web browser, and we use SpiderMonkey-17.0.0 [21]. It is a stack-based virtual machine having 229 distinct bytecodes. The opcode in SpiderMonkey has a variable length, and the bytecode pops the top of stack (TOS) values as sources. SpiderMonkey defines five primitive types (`Undefined`, `Null`, `Boolean`, `String`, and `Number`).

To represent a number, SpiderMonkey uses the double-precision IEEE 754 standard floating point representation. To represent non-FP type numbers, such as `Int` and `String`, SpiderMonkey exploits NaN values, whose exponent bits are all set to one (i.e., infinity) and fraction bits to a non-zero value. More specifically, SpiderMonkey sets 13 most-significant bits (MSBs) to one and uses the following 4 bits to represent a type. For example, for an `Int` value, the 32 LSBs are used for data, the 4-bit type field is set to 0b0001, and 13 MSBs to all ones. To extract the 4-bit type field for non-FP values, our current implementation uses the NaN

detection logic (Section 3.1) and sets R_{mask} to 0x0F. If NaN detection is enabled, the F/$\bar{I}$ bit is set to one if the value is not NaN.

For tagged load and store instructions the three special registers are set as follows: R_{offset} to 0b100, R_{shift} to 0b101111 (47), and R_{mask} to 0x0F. For FP values both tagged load and store instructions behave in the same way as normal load and store instructions except that the F/$\bar{I}$ flag is set to one. To extract the type field from a non-FP load value (i.e., a NaN value), it is shifted right by 47 bits and masked with R_{mask}. To store a non-FP value, say in `Ra`, to memory, the correct NaN value is reconstructed by putting together 47 LSBs with the register value ($\texttt{Ra.v}_{[46:0]}$), 4-bit type tag (R_{mask} AND `Ra.t`), followed by all ones for the remaining 13 MSBs. Note that tagged ALU instructions like `xadd` are executed normally using the F/$\bar{I}$ bit.

5. Discussion

OS interactions. In a realistic setup, OS context switching should be considered. We extend the register file with F/$\bar{I}$ bit and tag fields. The two field should be preserved across context switches as register state. The special registers, R_{offset}, R_{shift}, R_{mask} and R_{hdl} also must be saved at a context switch. Finally, the contents of the Type Rule Table must be preserved as well. When execution of a script is finished, a `flush_trt` instruction flushes all Type Rule Table entries. To insert a new entry, one can use a `set_trt` instruction.

Legacy code execution. The energy/power tax for legacy code execution should be low with Typed Architectures for the following reasons. First, as for dynamic power, the additional switching activities caused by the typed datapath will be minimal as legacy code does not use typed instructions and the type tag remains constant (untyped). Second, as for leakage/idle power, our proposal incurs only a small area overhead (1.6%), hence small power overhead. It is feasible to apply well-known power management techniques (e.g., clock and power gating) to the type-handling path to further reduce power.

Deoptimizing the fast path. Since frequent type mispredictions can cause significant performance penalty, we can place a path selector instruction at the header of the fast path to revert to the slow path if the miss rate is high. For this path selector instruction one can add a new instruction for slow path prediction or override the functionality to the `thdl` instruction. It is because `thdl` has a simple task of updating R_{hdl} with an immediate value. However, there is little room for reordering the `thdl` instruction without paying stall cycles due to data hazards. Thus, either way, the path selector instruction trades the performance of the fast path for accelerating the slow path.

Application to high performance core. While Typed Architecture is also applicable to high-end processors, its benefits are most pronounced on low-end processors where JIT is not practical. In such a resource-constrained environment JIT may 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 [49]. Unlike JIT, Typed Architecture is applicable to low-end processors and to workloads without hotspots.

ISA	64-bit RISC-V v2
Architecture	Single-Issue In-Order, 50MHz (Synthesized)
Pipeline	Fetch/Decode/Execute/Memory/Writeback (5 stages)
Branch Predictor	32B predictor (128-entry gshare) 62-entry, fully-associative BTB 2-entry RAS, 2-cycle branch miss penalty
Caches	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	1GB, DDR3-1066, 1 rank, tCL/tRCD/tRP = 7/7/7
Workloads	Lua-5.3.0, JavaScript (SpiderMonkey-17.0.0)

Table 6: Evaluation parameters

6. Experimental Setup

System parameters. Our model is based on open-source 64-bit RISC-V v2 Rocket core with the default RISC-V/Newlib target [17]. We have integrated custom performance counters for performance analysis, such as I-cache miss rate, branch misprediction rate, and so on. It is a fully synthesizable RTL model written in Chisel language. This model is compiled into Verilog RTL, and then synthesized for FPGA emulation and area/power estimation. We use Xilinx ZC706 FPGAs for instruction and cycle counts. Table 6 summarizes the parameters used for evaluation.

Synthesis. We use Synopsys Design Compiler (Version I-2013.12-SP5) to synthesize the same RTL model for realistic estimation of area and power. Five TSMC CLN40G technology libraries at a 40nm technology node are used, which consist of a 9-track standard cell library (SC9) and 4 SRAM libraries generated by ARM Artisan memory compilers. The most typical corner is selected (rvt_tt_typical_max_0p90v_25c). The SRAM arrays for tags and data of L1 caches are generated by memory compilers for high-density single-port regfile and SRAM, and high-speed dual-port regfile.

Scripting Engines. We use two popular open-source script interpreters: Lua (Version 5.3.0) [28] and JavaScript (SpiderMonkey Version 17.0.0) [21]. We compile both script interpreters using `gcc` version 5.2.0, built by the RISC-V toolchain, with `-O3` flag. For Lua we turn off garbage collection to not interfere the mutator (main) code. However, we were not able to turn off garbage collection for SpiderMonkey as there is no simple way to do it. We also implement an in-house version of Checked Load [30] running on FPGA, which combines (tag) load, compare, and branch into a single instruction. Checked Load represents a state-of-the-art hardware-based technique for reducing type checking overhead.

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

Table 7: Benchmarks

Benchmark summary. We take the benchmark set from recent work [47], initially. However, some benchmarks are replaced because `fasta` and `meteor` are not working on SpiderMonkey and `reverse-complement` is not working on Lua. These benchmarks simply do not run to completion on our FPGA. Moreover, `regex-dna` spends most of the time on native library code rather than bytecodes. Instead, we measure four other benchmarks: `n-sieve`, `random`, `fibo`, and `ackermann`. These benchmarks are taken from Computer Language Benchmarks Game [25], where the original 11 benchmarks originate from [43, 47]. We run all benchmarks to completion and report the instruction and cycle counts from the beginning and the end of the main interpreter loop. Input parameters are summarized in Table 7.

7. Evaluation

7.1 Overall Speedups

Figure 5 shows the overall performance speedups of Typed Architecture over the baseline. Typed Architecture achieves geomean speedups of 9.9% and 11.2% for Lua and SpiderMonkey, respectively, with maximum speedups of 43.5% and 32.6%. This compares favorably to 7.3% and 5.4% geomean speedups by Checked Load [30], a state-of-the-art hardware-based type checking technique. Note that the checked load instruction (`chklb`) in [30] is difficult to integrate into a RISC-style ISA with fixed instruction width (e.g., 32 bits) as it requires an additional 8-bit immediate field for a load instruction. Thus, their original design uses a variable-length x86_64 ISA. We sidestep this problem by introducing a new instruction that sets a type value to a special register, which is used by `chklb` for type checking.

The first source of performance improvement with Typed Architecture is the reduction in dynamic instruction count. As shown in Figure 6, the dynamic instruction count is reduced by 11.2% and 4.4% for Lua and SpiderMonkey, respectively. Besides, Typed Architecture also reduces resource pressures to the branch predictor, instruction cache, registers, and so on. This also brings significant performance benefits to some benchmarks. We will discuss the performance of each scripting engine in greater details.

Lua. The performance improvement of Lua is highly correlated with the amount of reduction in dynamic instruction count. For example, Typed Architecture reduces the instruction count for `fannkuch-redux` and `n-sieve` by 32.9% and 31.8%, respectively. These programs have frequent table accesses and the type hit rate is very high (as shown in Figure 9). However, executing fewer instructions is not the only factor that improves performance. While `fibo` has a very small amount of reduction in instruction count, it achieves a disproportionate speedup due to reduced pressure on the branch predictor. Figure 7 compares the branch misprediction rates of the three designs.

There are other benchmarks with smaller performance improvements even if their type hit rates (Figure 9) are high, such as `mandelbrot`, `pidigits`, and `spectral-norm`. They execute `CALL` bytecodes frequently, which are used to invoke system calls like `printf`. `k-nucleotide`, `ackermann` and `random` also execute many `CALL` bytecodes for function calls and file I/O. Thus, these programs have relatively small room for improvement due to Amdahl's Law. For `n-body`, `GETTABLE` is the most frequently used bytecode, but there are also frequent table lookups using a string as key, which should go through the slow path.

Checked Load [30] achieves a geomean speedup of 7.3% with maximum speedup 40.8% for `fannkuch-redux`. Although the maximum speedup is comparable, the geomean speedup of Checked Load is lower than Typed Architecture. It is because the type combination for the fast path of each bytecode is fixed at compile time in Checked Load to optimize a single pair of input types, so it cannot handle both integer- and FP-oriented workloads at the same time. Instead, Typed Architecture can adapt to both types of workloads by using polymorphic operators, hence achieving much more robust performance without requiring recompilation. Both `mandelbrot` and `n-body` heavily use FP arithmetic bytecodes, but the Lua VM for Checked Load is compiled to optimize integer arithmetic, which is the common case in the other cases.

SpiderMonkey. The reduction in dynamic instruction count with Typed Architecture is a major source of performance improvement for JavaScript as well. `fannkuch-redux`,

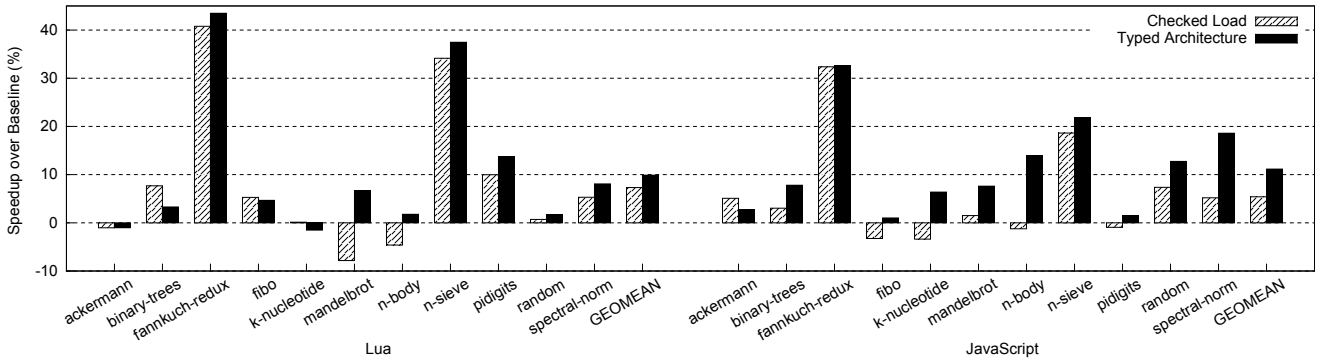

Figure 5: Overall speedups for Lua and JavaScript interpreters (the higher, the better)

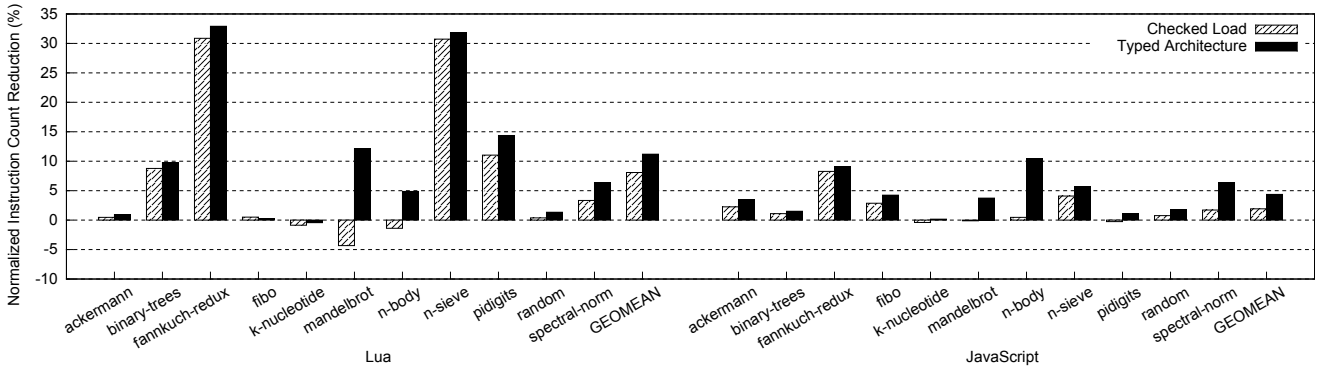

Figure 6: Reduction of dynamic instruction count (the higher, the better)

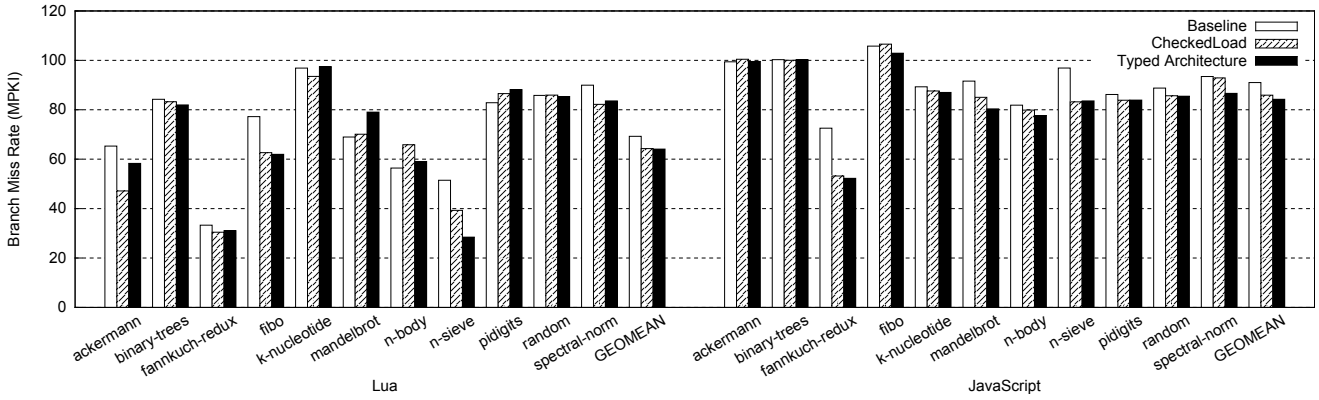

Figure 7: Branch miss rates in misses per kilo-instructions (MPKI) (the lower, the better)

`n-sieve`, and `spectral-norm` achieve speedups of 32.6%, 21.8%, and 18.6%, with dynamic instruction reduction of 9.0%, 5.7%, and 6.3%, respectively. These programs frequently access tables using `GETELEM` and `SETELEM` bytecodes, featuring high type hit rates (Figure 9). For `n-sieve` and `fannkuch-redux` the branch misprediction rate (shown in Figure 7) is also reduced by 13.7% and 28.0%, respectively, which is another source of improvement.

In contrast, `binary-trees`, `k-nucleotide`, and `random` achieve sizable speedups even if the amount of reduction in both instruction count and branch misprediction rate is relatively small. It is because Typed Architecture reduces the instruction cache miss rate by 20.7%, 11.6%, and 50.8%, for the three programs, respectively, as shown in Figure 8. Among them `k-nucleotide` shows a smaller performance gain due to high type miss rate in table accesses (Figure 9). SpiderMonkey should detect an overflow as a tag-value pair is co-located within a double-word, in which case Typed Architecture cannot execute the fast path. (The number of overflows is not included in Figure 9.)

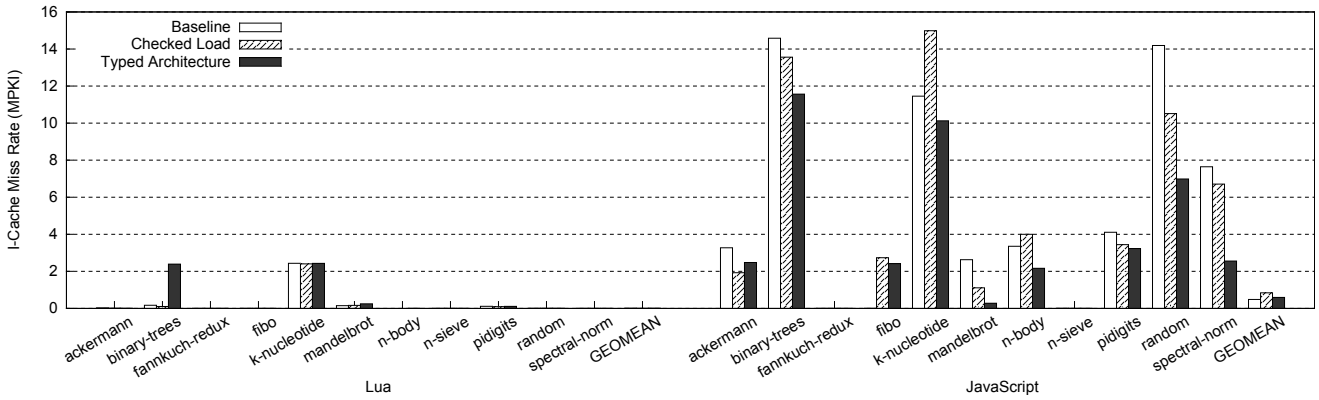

Figure 8: Instruction cache miss rates in misses per kilo-instructions (MPKI) (the lower, the better)

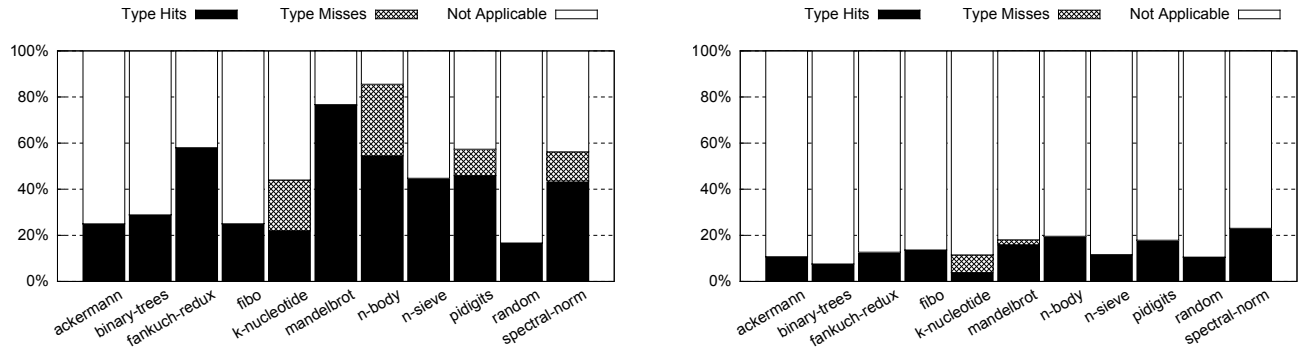

Figure 9: Type hit and miss rates normalized to dynamic bytecode count for (a) Lua and (b) SpiderMonkey

Checked Load [30] achieves a geomean speedup of 5.4% with a maximum speedup of 32.4% for `fannkuch-redux`. `n-body` gets worse performance than the baseline due to a high type miss rate.

7.2 Area and Energy Efficiency

Table 8 shows the area and power estimation of Typed Architecture implemented on a RISC-V Rocket Core. The total area and power of Rocket Core augmented with Typed Architecture are increased by 1.6% and 3.7%, respectively. Combined with the speedups in Section 7.1, the EDP is improved by 16.5% for Lua and by 19.3% for JavaScript. According to the area/power breakdown, Typed Architecture increases the area and power of the *core* module, which accounts for 6.7% of total area and 14.1% of total power. The core module integrates 8-bit tags into the register file, datapath for tag propagation, and an 8-entry Type Rule Table. According to the timing report Typed Architecture does not affect the critical path as the critical path is in the FPU module for both the baseline and Typed Architecture.

8. Related Work

Hardware-based acceleration of type checking. LISP machines in 1980's, such as Symbolics 3600 [46], TI Explorer [31, 33], and SPUR [52], provide hardware support for runtime type checking. Extending this idea, Steenkiste et al. [50] propose hardware extensions to the general-purpose MIPS-X architecture to accelerate LISP workloads. They show that an average of one fourth of execution time would be spent in handling tags if type checking is turned off. In the same spirit we advocate hardware support for type checking. In contrast to their proposal, which is LISP-specific (e.g., assuming the MSBs of a data word are used as type tag), our proposal aims to flexibly support multiple languages and implementations with low hardware cost to make them viable even for resource-constrained IoT platforms.

Recently, Anderson et al. [30] and Dot et al. [39] propose an ISA extension to support JavaScript type checking with new instructions and register. For accelerating type checking, they propose new complex instructions to control program flow depending on type information. However, their approaches have several limitations. First, they provide hard-

	Baseline				Typed Architecture			
Module Hierarchy	**Area (mm^2)**		**Power (mW)**		**Area (mm^2)**		**Power (mW)**	
Top	**0.684**	**100.0%**	**18.72**	**100.0%**	**0.695**	**100.0%**	**19.41**	**100.0%**
- Tile	0.627	91.6%	12.60	67.3%	0.638	91.7%	13.29	68.5%
| - Core	0.038	5.5%	2.22	11.8%	0.047	6.7%	2.74	14.1%
| | - CSR	0.008	1.2%	0.57	3.0%	0.009	1.3%	0.60	3.1%
| | - Div	0.006	0.9%	0.17	0.9%	0.006	0.9%	0.18	0.9%
| - FPU	0.089	13.0%	3.18	17.0%	0.089	12.9%	3.23	16.6%
| - ICache	0.251	36.7%	3.49	18.7%	0.251	36.1%	3.50	18.0%
| - DCache	0.249	36.4%	3.71	19.8%	0.250	36.0%	3.82	19.7%
- Uncore	0.046	6.8%	4.75	25.3%	0.046	6.7%	4.74	24.4%
- Wrapping	0.011	1.6%	1.38	7.4%	0.011	1.6%	1.38	7.1%

Table 8: Hardware overhead breakdown (area, power)

ware support only for type checking operations; still, tag calculation, insertion, removal, and extraction are explicitly performed by software with significant overhead. Second, Checked Load assumes a specific tag-value layout (i.e., the MSB of a word is reserved for tag), hence having limited applicability. Our proposal provides a more comprehensive solution with broader applicability as well as higher efficiency by managing tags mostly in hardware.

Hardware support for metadata processing. Recent work on hardware-based tag processing, most notably for taint tracing and memory bound checking, can be viewed as limited forms of hardware-based type checking [6, 9, 32, 34, 35, 37, 41, 44, 45, 48, 51, 54]. For example, DIFT [51] proposes to attach a one-bit tag to every variable, indicating authentic or spurious, to dynamically track the information flow of spurious data (originally through I/O) to thwart a broad range of security exploits. Hardbound [37] augments every pointer variable in C with their legitimate bounds to perform memory bound checking at runtime, hence improving security. While some of the hardware features in these proposals might be relevant to hardware-based type management, their hardware designs are specialized for different goals and too inflexible for our use.

To provide flexible tag processing capabilities, some hardware designs offer configurable options to users [36, 38, 53]. As one of the most recent studies, Dhawan et al. [38] propose the PUMP architecture (PUMP stands for "Programmable Unit for Metatdata Processing"), which can accommodate both DIFT and HardBound [37], for example. Their design focuses on generality to support complex tag propagation and checking rules, which are possibly unbounded. However, the PUMP architecture incurs high hardware and energy cost (110% on-chip storage overhead with 60% energy overhead) for this flexibility, hence not suitable for resource-constrained IoT platforms. In addition, while aiming to be a general-purpose tag processing architecture, PUMP only demonstrates relatively simple use cases such as taint tracing and memory bound checking.

JIT-based Type Specialization A popular approach to accelerate dynamic scripts is to generate type-specialized codes with minimal type checks, either statically or dynamically. WebKit's FTL JIT [27] represents the state-of-the-art of dynamic, profile-directed JIT compilation for JavaScript, which builds on the LLVM JIT infrastructure. Some proposals [40, 42] exploit both static and dynamic (or profiling-based) analysis strategies for type specialization of JIT compiled code. Kedlaya et al. [42] propose two-pass type inference: purely static first pass and profile-guided second pass. In this way they significantly improves the coverage of type specialization to have fewer type guards. Another interesting approach is to exploit the type information passed from the original program written in a statically typed language. For example, Emscripten [55] generates type-specialized JavaScript codes by compiling statically typed C programs.

While these software-only techniques have achieved some successes, profile-guided dynamic compilation techniques are not a viable option on emerging IoT devices due to the high cost of profiling and compilation in terms of CPU cycles and memory usage. Also, it is difficult to generate efficient codes with static analyses only, due to the dynamic nature of modern scripting languages with the lack of type information at compile time.

9. Conclusion

This paper introduces *Typed Architectures*, a new class of processor architectures in which each data value retains high-level type information at an ISA level. Typed Architectures calculate and check the dynamic type of each variable *implicitly* in hardware, rather than explicitly in software, hence significantly reducing instruction count for dynamic type checking. According to our experiment using an RTL prototype on FPGA, Typed Architectures achieve geomean speedups of 11.2% and 9.9% for two popular scripting engines for JavaScript and Lua, respectively. Moreover, according to our synthesis results, Typed Architectures improves the EDP of JavaScript by 19.3% and Lua by 16.5% with an area overhead of 1.6% at a 40nm technology.

10. Acknowledgments

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

References

[1] Angry Birds. https://www.angrybirds.com/games/.

[2] Arduino Yun. https://www.arduino.cc/en/Main/ArduinoBoardYun.

[3] Intel Edison. https://software.intel.com/en-us/iot/hardware/edison.

[4] Espruino. http://www.espruino.com.

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

[6] Introduction to Intel Memory Protection extensions. http://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[7] JavaScript. https://developer.mozilla.org/en/docs/Web/JavaScript.

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

[9] LowRISC. http://www.lowrisc.org.

[10] Matlab. http://www.mathworks.com/products/matlab/.

[11] Node.js–open-source, cross-platform runtime environment for developing server-side web applications. https://nodejs.org/.

[12] Perl. https://www.perl.org.

[13] Adobe's Photoshop Lightroom Developer Center. http://www.adobe.com/devnet/photoshoplightroom.html.

[14] Python. https://www.python.org.

[15] The R Project for Statistical Computing. http://www.r-project.org.

[16] Raspberry Pi. https://www.raspberrypi.org/.

[17] Rocket Core. http://riscv.org/download.html.

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

[19] Ruby. https://www.ruby-lang.org/en/.

[20] SAMA5D3 Series Datasheet. http://www.atmel.com/Images/Atmel-11121-32-bit-Cortex-A5-Microcontroller-SAMA5D3_Datasheet.pdf.

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

[22] World of Warcraft interface AddOn Kits. https://us.battle.net/support/en/article/download-the-world-of-warcraft-interface-addon-kit.

[23] Intel XDK. https://software.intel.com/en-us/intel-xdk.

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

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

[26] Duktape. http://duktape.org.

[27] FTL: WebKit's LLVM based JIT. http://blog.llvm.org/2014/07/ftl-webkits-llvm-based-jit.html.

[28] The Programming Language Lua. http://lua.org.

[29] ARM mbed. https://www.mbed.com/en/.

[30] O. Anderson, E. Fortuna, L. Ceze, and S. Eggers. Checked Load: Architectural Support for JavaScript Type-checking on Mobile Processors. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2011.

[31] P. W. Bosshart, C. R. Hewes, M. D. Ales, M. C. Chang, K. K. Chau, K. Fasham, C. C. Hoac, T. W. Houston, V. Kalyan, S. L. Lusky, S. S. Mahant-Shetti, D. J. Matzke, K. N. Ruparel, J. F. Sexton, C. H. Shaw, T. Shridhar, D. Stark, and A. L. Lee. A 553 k-transistor lisp processor chip. *IEEE Journal of Solid-State Circuits (JSSC)*, 1987.

[32] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. K. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005.

[33] C. J. Corley and J. A. Statz. LISP workstation brings AI power to a user's desk. In *Computer Design, January*, 1985.

[34] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim. (TACO)*, 2006.

[35] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[36] D. Y. Deng and G. E. Suh. High-performance parallel accelerator for flexible and efficient run-time monitoring. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2012.

[37] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[38] U. Dhawan, C. Hritcu, R. Rubin, N. Vasilakis, S. Chiricescu, J. M. Smith, T. F. Knight, Jr., B. C. Pierce, and A. DeHon. Architectural Support for Software-Defined Metadata Processing. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015.

[39] G. Dot, A. Martínez, and A. González. Analysis and optimization of engines for dynamically typed languages. In *Proceedings of the 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2015.

[40] B. Hackett and S.-y. Guo. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[41] M. E. Houdek, F. G. Soltis, and R. L. Hoffman. Ibm system/38 support for capability-based addressing. In *Proceedings of the 8th Annual Symposium on Computer Architecture (ISCA)*, 1981.

[42] M. N. Kedlaya, J. Roesch, B. Robatmili, M. Reshadi, and B. Hardekopf. Improved Type Specialization for Dynamic Scripting Languages. In *Proceedings of the 9th Dynamic Languages Symposium (DLS)*, 2013.

[43] C. Kim, S. Kim, H. G. Cho, D. Kim, J. Kim, Y. H. Oh, H. Jang, and J. W. Lee. Short-circuit dispatch: Accelerating virtual machine interpreters on embedded processors. In *Proceedings of the 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[44] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, Jr., and A. DeHon. Low-fat pointers: Compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2013.

[45] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[46] D. A. Moon. Architecture of the symbolics 3600. In *Proceedings of the 12nd Annual International Symposium on Computer Architecture (ISCA)*, 1985.

[47] T. Oh, H. Kim, N. P. Johnson, J. W. Lee, and D. I. August. Practical Automatic Loop Specialization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[48] F. J. Pollack, G. W. Cox, D. W. Hammerstrom, K. C. Kahn, K. K. Lai, and J. R. Rattner. Supporting ada memory management in the iapx-432. In *Proceedings of the 1st International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1982.

[49] P. Ratanaworabhan, B. Livshits, and B. G. Zorn. JSMeter: Comparing the behavior of JavaScript benchmarks with real web applications. In *Proceedings of the 2010 USENIX Conference on Web Application Development (WebApps)*, 2010.

[50] P. Steenkiste and J. Hennessy. Tags and Type Checking in LISP: Hardware and Software Approaches. In *Proceedings of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems (ASPLOS)*, 1987.

[51] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[52] G. S. Taylor, P. N. Hilfinger, J. R. Larus, D. A. Patterson, and B. G. Zorn. Evaluation of the SPUR Lisp Architecture. In *Proceedings of the 13rd Annual International Symposium on Computer Architecture (ISCA)*, 1986.

[53] G. Venkataramani, I. Doudalis, Y. Solihin, and M. Prvulovic. Flexitaint: A programmable accelerator for dynamic taint propagation. In *Proceedings of the 14th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2008.

[54] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014.

[55] A. Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011.