Received September 5, 2020, accepted October 7, 2020, date of publication October 12, 2020, date of current version October 21, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3030214

Segment-Based Multiple-Base Compressed Addressing for Flexible JavaScript Heap Allocation

GYEONGHWAN HONG AND DONGKUN SHIN, (Member, IEEE)
Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon 16419, South Korea

Corresponding author: Dongkun Shin (dongkun@skku.edu)

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2017-0-00914).

ABSTRACT Many Internet-of-Things (IoT) systems use lightweight JavaScript engines to support easy programming in microcontrollers. Lightweight JavaScript engines use several techniques for memory optimization, such as static heap reservation and compressed addressing. Recent IoT systems also use several external libraries and a larger on-chip memory to support abundant functionalities, such as machine learning and connectivity. However, as the JavaScript heap space is not resizable owing to the memory optimizations, the JavaScript engine or an external library is prone to fail the memory allocation in these devices. To address this problem, we propose a flexible memory optimization technique, *segment-based multiple-base compressed addressing (SMBCA)*, which compresses a pointer indicating a JavaScript object allocated to a resizable heap based on multiple base addresses. SMBCA comprises two components: a dynamic segment allocator (DSA) and a multiple-base compressed address translator (MBCAT). DSA dynamically allocates the JavaScript heap in segment units. Meanwhile, MBCAT converts a low-bitwidth address into a full-bitwidth address and vice versa, based on the multiple base addresses. To reduce the address compression overhead of MBCAT, we propose a software cache technique, *reverse map cache (RMC)*. We found that the SMBCA reduces average memory usage by 43.9% compared to the existing lightweight JavaScript engines when running SunSpider benchmarks, V8 benchmarks, and real-world applications. We also showed that the RMC reduces the average address compression latency of MBCAT by 34.9% when running the SunSpider benchmarks.

INDEX TERMS Compressed addressing, Internet of Things, JavaScript, memory management, microcontroller.

I. INTRODUCTION

Internet-of-things (IoT) systems require low power consumption, low cost, and small form factors. Thus, many IoT systems use low-end devices, such as microcontrollers (MCUs). As presented in Table 1, the MCUs usually include an ARM Cortex-M CPU or ARM Cortex-R CPU and a wide range of on-chip memory (SRAM: from 128 KB to 4.5 MB).

Recently, two trends exist in these low-end devices. First, for the easy programming of IoT applications, MCUs run a lightweight interpreter engine that can support dynamic programming languages, such as JavaScript and Python. For example, Espruino [1] operates on various boards, such as Espruino Pico with 96 KB of SRAM and Espruino WiFi with 128 KB of SRAM. Further, IoT.js [2], Duktape [3], and MicroPython [4] also support various boards.

Second, to support abundant functionality, such as DNN-based machine learning [5]–[8] or IoT connectivity standards [9], [10], several external libraries have been used by recent low-end devices. As these libraries use large buffers, such as intermediate feature map buffers [5]–[8] or request message buffers [9], [10], they require a large amount of memory space. Therefore, to use these libraries within the limited memory capacity of the MCUs, the libraries introduce buffer management optimization techniques [7]–[10] or DNN model compression techniques [5]–[8]. For example,

The associate editor coordinating the review of this manuscript and approving it for publication was Jun Wu.

TABLE 1. State-of-the-art low-end devices used for IoT systems.

Name	CPU	Frequency	SRAM
Nucleo F476RG	Cortex-M4	80 MHz	128 KB
i.MX RT1020	Cortex-M7	500 MHz	256 KB
Nucleo F746ZG	Cortex-M7	216 MHz	320 KB
Nucleo F767ZI	Cortex-M7	216 MHz	512 KB
Nucleo H743ZI2	Cortex-M7	480 MHz	1 MB
Nucleo H7A3ZI-Q	Cortex-M7	280 MHz	1.18 MB
Artik 053	Cortex-R4	320 MHz	1.25 MB
i.MX RT600	Cortex-M33	300 MHz	4.5 MB

CMSIS-NN uses a feature map buffer optimization, referred to as partial-im2col, and a model compression technique, referred to as 8-bit quantization, to reduce the peak memory usage to hundreds of KB [7]. In addition, IoTivity Lite, which supports OCF-standard-based connectivity, improves the request message buffer management to reduce the peak memory usage to 68.72 KB [10]. Despite the memory optimization techniques, these libraries still require large buffers. Therefore, recent IoT systems use also low-end devices equipped with large on-chip memories, as demonstrated in Table 1.

To run JavaScript applications with a small memory capacity, lightweight JavaScript engines [1]–[3] use memory optimization techniques, such as *compressed addressing* and *static heap reservation*. Compressed addressing is a technique that expresses a pointer indicating to a JavaScript object with a low-bitwidth address by specifying the constraints of the JavaScript heap. For example, Fig. 1(a) shows an example of IoT.js [2]. In this example, a contiguous memory space of less than 512 KB is allocated to the JavaScript heap. Subsequently, IoT.js manages the heap in an 8B-aligned block unit and allocates blocks directly to a JavaScript object. Further, it can express a pointer that indicates the object with a 16-bit address by defining the address as an offset from the base address of the heap. IoT.js applies the compressed addressing to all the pointers inside the JavaScript object, which reduces the minimum JavaScript object size from 16 B to 8 B.

Static heap reservation is a scheme that reserves memory space for the entire JavaScript heap. Through the static heap reservation, the JavaScript engine can allocate memory space directly to an object. Owing to the direct memory allocation, it can manage metadata of the memory space directly. As the type of a JavaScript object determines its size, the JavaScript engine can store only the object's type instead of its size. Therefore, the static heap reservation can reduce the metadata's size created when the engine allocates a space to an object.

However, a JavaScript engine reserves a large amount of memory space regardless of the heap usage when it uses the static heap reservation. Thus, when the JavaScript engine and external libraries run simultaneously, they are prone to fail memory space allocation, which is called an *over-provisioning problem*. To address this problem, the heap must be resizable in proportion to the number of objects.

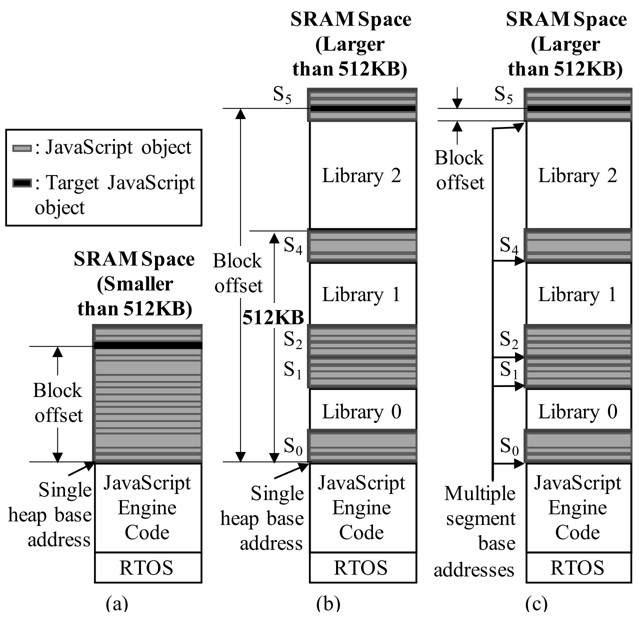


FIGURE 1. Comparison of JavaScript memory optimization techniques; (a) IoT.js [2] (SBCA with static heap reservation), (b) SBCA with dynamic segment allocation, (c) SMBCA (MBCA with dynamic segment allocation).

To implement a resizable heap with minimal metadata, JavaScript engines can use *dynamic segment allocation*, which allocates the heap in fixed-size segment units.

To execute a JavaScript application and external libraries simultaneously on a low-end device, the JavaScript engine must use both dynamic segment allocation and compressed addressing. However, a JavaScript engine merely combining the two techniques cannot support SRAM of more than 512 KB. In this case, the low-bitwidth address cannot indicate all the objects in the memory. If the block size is 8B and a pointer is represented with 16-bit, maximum heap size that can use the 16-bit pointer is 512 KB. Fig. 1(b) shows an example where JavaScript objects are allocated to five segments on an SRAM space larger than 512 KB. As segment S_5 is outside the range of 512 KB from the base address, a 16-bit address cannot point to the object in S_5. This is because the existing compressed addressing maps a low-bitwidth address to a full-bitwidth address using only a single heap base address. We call this compressed addressing as *single-base compressed addressing (SBCA)*.

In this study, we propose *multiple-base compressed addressing (MBCA)* to address the SBCA's problems. MBCA is a compressed addressing scheme that maps a low-bitwidth address to a full-bitwidth address based on multiple segment base addresses. In MBCA, a low-bitwidth address pointing to a JavaScript object is defined as a combination of a segment index and a block offset. The segment index is the identifier of the segment containing the object. The block offset is the difference between the segment base address of the segment and the full-bitwidth address pointing the object.

As the segment size and maximum heap size are configured, the segment index and the block offset can be expressed with small bitwidths. Multiple segment base addresses allow the low-bitwidth address to indicate any JavaScript object allocated to a memory of any size. For example, we assume that a JavaScript engine uses the dynamic segment allocation, the block alignment size of the engine is 8 B, the segment size is 2 KB, and the maximum heap size is 512 KB. In this case, because the segment index and block offset can be expressed with 8 bits respectively, a 16-bit address can indicate any object in the SRAM larger than 512 KB. As shown in Fig. 1(c), the MBCA enables a 16-bit address to indicate an object in S_5.

Compressed addressing requires two types of address translation operations; address decompression and address compression. Address decompression is an operation to convert a low-bitwidth address into a full-bitwidth address. Address compression is an operation to convert a full-bitwidth address into a low-bitwidth address. In MBCA, address translation must determine which segment a low-bitwidth address or a full-bitwidth address indicates. At this time, the JavaScript engine must use the mapping metadata between the segment index and segment base address. It results in more memory accesses of metadata and longer address translation latency than in SBCA. Therefore, the MBCA requires an optimized address translator that reduces the address translation latency.

In this study, we propose *segment-based multiple-base compressed addressing (SMBCA)*, a JavaScript heap management scheme that uses both the dynamic segment allocation and the MBCA. SMBCA enables the JavaScript engine to resize its heap while using the compressed addressing within a memory of any size. The JavaScript engine can run together with external libraries on a low-end device with a minimal memory footprint of the JavaScript objects. SMBCA consists of two sub-modules: dynamic segment allocator (DSA) and multiple-base compressed address translator (MBCAT). DSA is a module responsible for segment allocation and deallocation. MBCAT is a module that performs address translation based on multiple segment base addresses.

In addition, we propose a *reverse map cache (RMC)*, an optimization technique to reduce the address compression latency of MBCAT. The RMC is a software cache of the mapping metadata between a segment base address and a segment index. The RMC reduces memory accesses required for the address compression by using the segment access locality of the JavaScript application.

We implemented *IoT.js-SMBCA*, a variant of IoT.js [2], using SMBCA. In the experiment, we compared *IoT.js-SMBCA* to existing JavaScript engines on a Raspberry Pi 3 and an Artik 053. For the experiment, we executed SunSpider [11] and V8 [12], well-known JavaScript benchmarks, on the two devices. We also ran three real-world applications on Artik 053. The experimental results show that the *IoT.js-SMBCA* used 43.9% less memory than the existing engines on average. The results also show that the RMC reduces the address compression latency by 34.9% on average and reduces SMBCA execution-time overhead from 26.1% to 19.9% in the SunSpider benchmark.

TABLE 2. Comparison of the memory optimization options of lightweight JavaScript engines.

Name	Compressed Addressing	JavaScript Heap Allocation	Pointer Size	Min. Object Size
IoT.js default option (*IoT.js-Def*) [2]	SBCA	Static heap reservation	16-bit	8B
Duktape lowmem option (*Duk-Low*) [3]	SBCA	Static heap reservation	16-bit	20B
Espruino default option (*Esp-Def*) [1]	SBCA	Static heap reservation	8-bit, 10-bit or 16-bit	12B or 16B
Duktape default option (*Duk-Def*) [3]	None	Dynamic object allocation	32-bit	32B
Espruino resizable option (*Esp-Res*) [1]	None	Dynamic segment allocation	32-bit	28B
IoT.js SMBCA option (*IoT.js-SMBCA*)	MBCA	Dynamic segment allocation	16-bit	8B

II. BACKGROUND

A. LIGHTWEIGHT JavaScript ENGINES

The lightweight JavaScript engine is an interpreter engine that runs JavaScript applications on low-end devices with limited computing resources and memory resources. Several lightweight JavaScript engines, such as JerryScript [2], Duktape [3], and Espruino [1], have been proposed previously. IoT.js is a JavaScript engine that adds an event loop and I/O APIs to the JerryScript [2].

The lightweight JavaScript engines introduce as few performance optimization techniques as possible because such techniques create an additional memory footprint. For example, IoT.js introduces a snapshot, an ahead-of-time compile for built-in JavaScript functions to reduce the initialization latency. However, many performance optimization techniques such as inline caching [13], hidden class, and just-in-time compile, which are widely used in high-end JavaScript engines such as V8 [14], are not adopted by the lightweight JavaScript engines.

B. MEMORY OPTIMIZATION OF LIGHTWEIGHT JavaScript ENGINES

Lightweight JavaScript engines adopt many memory optimization techniques to reduce the memory footprint. As presented in Table 2, the JavaScript engines provide several memory optimization options.

The IoT.js's default option (*IoT.js-Def*) [2], Espruino's default option (*Esp-Def*) [1], and Duktape's *lowmem* option (*Duk-Low*) [3] are options for low-end devices, which use static heap reservation and SBCA. JavaScript engines based on static heap reservation prevent the allocation of JavaScript objects from requiring additional metadata. However, because they require to reserve a large amount of SRAM space, an over-provisioning problem occurs.

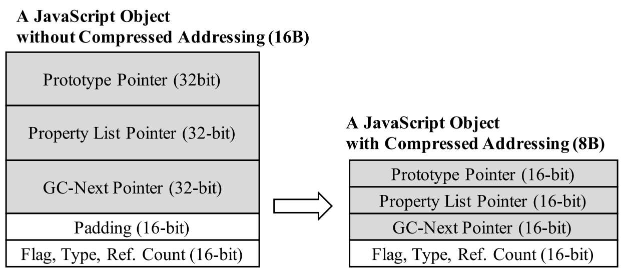

A JavaScript Object
without Compressed Addressing (16B)
Prototype Pointer (32bit)
Property List Pointer (32-bit)
GC-Next Pointer (32-bit)
Padding (16-bit)
Flag, Type, Ref. Count (16-bit)
A JavaScript Object
with Compressed Addressing (8B)
Prototype Pointer (16-bit)
Property List Pointer (16-bit)
GC-Next Pointer (16-bit)
Flag, Type, Ref. Count (16-bit)

FIGURE 2. Structure of a JavaScript object in IoT.js [2].

SBCA reduces the memory footprint of JavaScript applications by reducing the sizes of the pointers that occupy a large portion of JavaScript objects. For example, as shown in Fig. 2, JavaScript objects in *IoT.js-Def* [2] contain at least three pointers. If heap size is 512 KB and block size is 8 B, each pointer can be expressed with 16 bits through compressed addressing. Therefore, the compressed addressing reduces the minimum memory footprint of an object from 16 B to 8 B. In *Duk-Low* [3] and *Esp-Def* [1], the sizes of objects vary because the structure of the JavaScript objects is different. Using the compressed addressing, *Duk-Low* reduces the minimum object size from 32 B to 20 B, whereas *Esp-Def* reduces the size from 28 B to either 12 B or 16 B depending on the heap size.

Meanwhile, Duktape and Espruino provide options for a resizable heap. Duktape's default option (*Duk-Def*) uses dynamic object allocation, which allocates heap in the object unit. Espruino's *resizable* option (*Esp-Res*) uses dynamic segment allocation that allocates heap in segment units. However, because no address translator exists that allows JavaScript engines to use both dynamic segment allocation and compressed addressing, *Esp-Res* does not reduce the memory footprint of JavaScript objects.

If a JavaScript engine merely uses the combination of dynamic segment allocation and SBCA, as shown in Fig. 1(b), a resizable heap can be implemented while reducing the memory footprint of JavaScript objects. However, in this case, the JavaScript engine cannot indicate all the objects outside the range of 512 KB from the base address. Therefore, in this study, we propose SMBCA, a heap management scheme that manages the heap in segment units and compresses the pointers using multiple segment base addresses within a memory of any size. SMBCA enables the JavaScript heap resizable through dynamic segment allocation and reduces the size of JavaScript objects through compressed addressing.

III. RELATED WORK

Several lightweight JavaScript engines, such as Espruino [1], IoT.js [2], and Duktape [3], have been proposed, which run JavaScript applications within the small memory constraints of MCUs. They can run in MCU as they minimize the size of JavaScript objects through memory optimization such as compressed addressing and static heap reservation.

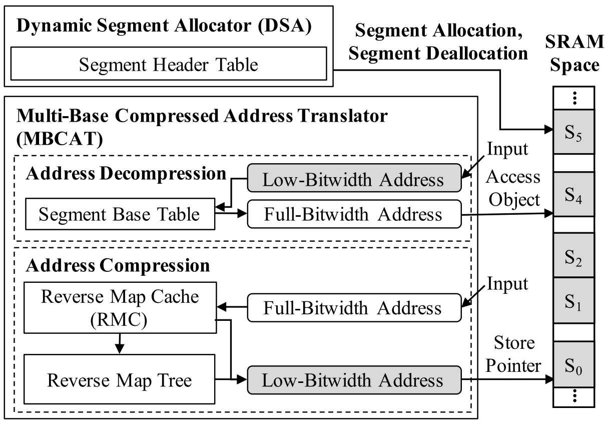


FIGURE 3. Architecture of SMBCA.

Several studies on the optimization of lightweight JavaScript engines have been proposed. Duktape+ is an optimized variant of Duktape which uses several performance optimizations of built-in functions and reduces the binary size of built-in objects through a lazy construction technique [15]. MoMIT is a multi-objective optimization technique that searches the most appropriate Duktape option that satisfies both the target device's constraints and the target application's functional requirements [16]. WebletScript is a Duktape-based automatic code slicing engine that distributes and executes a JavaScript-based IoT application across multiple IoT devices [17]. The existing optimization techniques are orthogonal to our proposed scheme since they focused on optimizing the engine's performance or binary size. Since they assumed that only the lightweight JavaScript engine runs on the MCU, there are few studies to improve the heap allocation of JavaScript engines.

IV. DESIGN

A. OVERALL ARCHITECTURE

SMBCA is a memory optimization technique that enables low-end devices with memories of any size to ensure that both resizable heap and compressed addressing are used. Fig. 3 shows the overall architecture of SMBCA. It also comprises a DSA and an MBCAT. DSA enables the JavaScript engine to support a resizable heap, whereas MBCAT enables the engine to use compressed addressing in a memory of any size.

DSA is a module that allocates and manages the JavaScript heap in segment units. It allocates a new segment when an application attempts to allocate a new object and cannot find free space in the currently allocated heap segments. Further, DSA deallocates a segment when all objects in the segment are deallocated during garbage collection. This module manages a segment header table containing segment allocation metadata. More details of DSA are provided in Section IV-B.

MBCAT is a module in charge of address translation based on multiple segment base addresses. It provides two address translation operations, referred to as address decompression and address compression. After the JavaScript engine converts a low-bitwidth address into a full-bitwidth address through address decompression, it accesses the object with the full-bitwidth address. After the JavaScript engine converts a full-bitwidth address into a low-bitwidth address through address compression, it expresses a pointer with the low-bitwidth address and stores it in the memory space. This module manages a segment base table and a reverse map tree, which are the metadata for address translation. In addition, MBCAT reduces the number of memory accesses during address compression through an RMC. More details about MBCAT are provided in Section IV-C.

B. DSA

DSA is a module that allocates a part of the heap in a segment unit and enables the JavaScript engine to resize the heap dynamically. It also manages a *segment header table* which is metadata to control segment allocation information. An entry of the segment header table comprises a segment index, used size, and segment group ID. The segment index is the identifier of each segment, and the used size is the sum of the sizes of the blocks allocated inside the segment. A segment group is a group of segments physically allocated to consecutive memory spaces, and a segment group ID is an identifier for each segment group. The segment group ID allows DSA to allocate an object larger than a segment.

DSA creates a new segment header table entry during segment allocation and removes an entry from the segment header table during segment deallocation. Whenever the JavaScript engine allocates or deallocates blocks, it updates the used size in the segment header table entry.

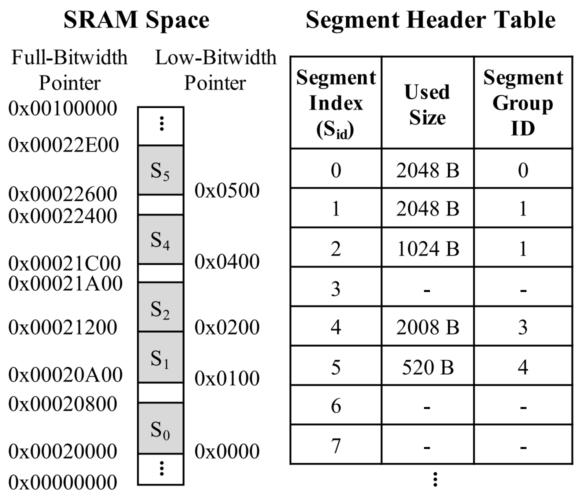
When the JavaScript engine attempts to allocate an object larger than the size of a segment, DSA allocates several segments to a physically contiguous memory space. Subsequently, this module creates segment header table entries for the segments and sets the same segment group ID to the entries. When the used size of all the segments in the segment group becomes zero through garbage collection, all segments in the segment group are removed.

For example, Fig. 4 shows an example of the JavaScript heap segments and segment header table when a JavaScript engine based on SMBCA runs on a device with an SRAM of 1 MB, such as a Nucleo H743ZI2. In this example, we assume that the segment size is 2 KB, and the block alignment size is 8 B. Here, blocks of 7,648 B are allocated to five segments, and an object of 3 KB is allocated across segments S_1 and S_2. In this case, DSA assigns the same segment group ID to S_1 and S_2 to ensure that the two segments are deallocated simultaneously.

C. MBCAT

MBCAT is a module for translating a low-bitwidth address to a full-bitwidth address or vice versa based on multiple segment base addresses. It enables the JavaScript engines to use both compressed addressing and dynamic segment allocation in devices with memories of any size.

MBCAT provides two address translation operations: *address decompression* and *address compression*. Because the address translation operations are hot functions that are called every time an object is accessed, they account for a large portion of an application's execution time.



Segment Index (S_{id})	Used Size	Segment Group ID
0	2048 B	0
1	2048 B	1
2	1024 B	1
3	-	-
4	2008 B	3
5	520 B	4
6	-	-
7	-	-
⋮		

FIGURE 4. Metadata of DSA.

1) ADDRESS DECOMPRESSION

Fig. 5 shows how an address decompression operation works in MBCAT. MBCAT manages metadata called a *segment base table* for address decompression. An entry in the segment base table comprises a segment index and a segment base address, and each entry is identified by the segment index. A segment base table entry is created during segment allocation, deleted during segment deallocation, and accessed during address decompression.

When MBCAT receives a low-bitwidth address, it extracts the segment index and block offset from the low-bitwidth address. When the low-bitwidth address size is l_L, the segment size is l_S, and the block alignment size is l_B, the most significant $l_L - (log_2(l_S) - log_2(l_B))$ bits in the low-bitwidth address is the segment index, and the least significant $log_2(l_S) - log_2(l_B)$ bits is the block offset. In the example of Fig. 5, l_L is 16, l_S is 2048 B, and l_B is 8 B. In this case, the most significant 8 bits of the low-bitwidth address is the segment index, and the least significant 8 bits is the block offset. MBCAT accesses the segment base table with using the segment index as a key and obtains the segment base address. Subsequently, this module calculates the full-bitwidth address using the segment base address and block offset.

2) ADDRESS COMPRESSION

Fig. 6 shows how an address compression operation works in MBCAT. First, MBCAT identifies the segment indicated by the full-bitwidth address and obtains the segment index and segment base address. Subsequently, it also obtains the block offset by calculating the difference between the full-bitwidth address and the segment base address. Finally, MBCAT calculates a low-bitwidth address by combining the segment index and the block offset.

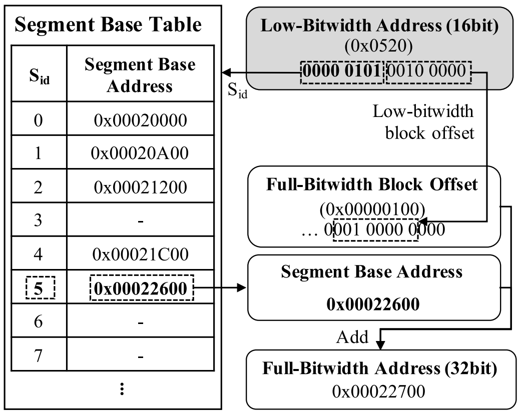


FIGURE 5. Address decompression in MBCAT.

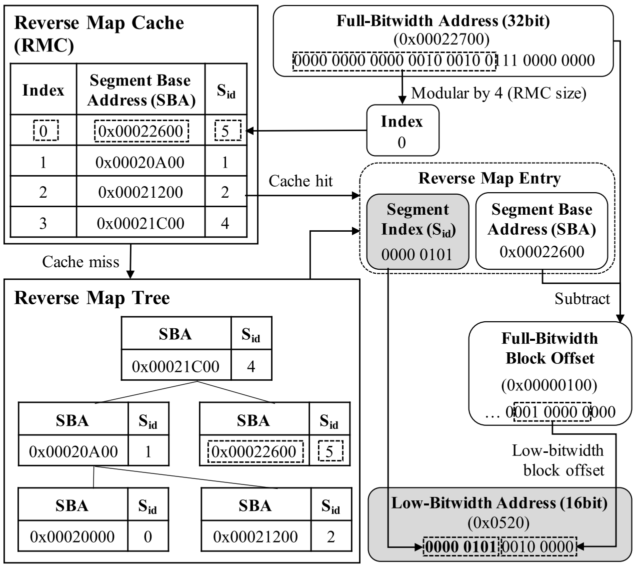


FIGURE 6. Address compression in MBCAT with an RMC.

As a full-bitwidth address does not contain a segment index, identifying the segment indicated by the full-bitwidth address requires a search in the segment base table. However, to prevent a full search on the segment base table, MBCAT uses additional metadata sorted by segment base addresses, called a *reverse map tree*. The reverse map tree is a red-black tree that sorts reverse map entries based on the segment base address. Reverse map entry is a pair of segment indices of the segment base address. MBCAT identifies the segment indicated by the full-bitwidth address through a binary search on the reverse map tree.

In the worst case, address compression requires memory accesses on $log_2(N)$ reverse map entries when the number of segments is N. Meanwhile, address decompression requires a memory access to only one entry in the segment base table. Accordingly, many memory accesses to the metadata make address compression significantly slower than the existing JavaScript engine, resulting in a deteriorating application execution time.

3) RMC

In this study, we propose an optimization technique, based on the segment access locality of a JavaScript application, to reduce the address compression latency of MBCAT. Address compression occurs when a pointer represented with compressed addressing is updated. In JavaScript engines, the main causes of pointer updates are the object allocation and object deallocation. In lightweight JavaScript engines such as IoT.js [2], blocks are allocated in *a first-fit manner*. Therefore, object allocation or deallocation occurring at similar times increase the probability of accessing the same segments. In this study, we propose an RMC, a software cache for reverse map entries based on the segment access locality.

In general, if a software cache is designed as a fully associative cache or a set-associative cache, the cache creates multiple memory accesses. To avoid the multiple memory accesses, the RMC is designed as a direct-mapped cache. An entry of RMC is a reverse map entry, and the index of the RMC is the most significant $32-log_2(l_S)$ bits of full-bitwidth addresses. The size of the RMC can be configured at compile time.

During address compression, MBCAT accesses the RMC and obtains a reverse map entry corresponding to the index. Then, MBCAT compares the given full-bitwidth address with the segment base address in the entry. If the full-bitwidth address indicates the segment indicated by the reverse map entry, a cache hit occurs. Otherwise, a cache miss occurs. In the case of a cache hit, MBCAT performs address compression using the reverse map entry obtained from the RMC. In the case of cache miss, MBCAT uses a reverse map entry obtained through a binary search.

For example, when compressing a full-bitwidth address of 0x00022700, as shown in Fig. 6, only one reverse map entry in RMC is accessed in the case of a cache hit. Meanwhile, in the case of a cache miss, one reverse map entry in RMC and two reverse map entries in the reverse map tree are accessed.

V. EVALUATION

A. EXPERIMENTAL SETTINGS

In this study, we implemented *IoT.js-SMBCA*, a variant of IoT.js [2], applying SMBCA. We used the September 2017 version of IoT.js. In the experiments, we compared the *IoT.js-SMBCA* to several memory optimization options provided by the existing JavaScript engines shown in Table 2. We used *IoT.js-Def*, *Esp-Def*, *Duk-Def*, and *Esp-Res* for the experiments. We excluded *Duk-Low* from the experiments because it cannot execute most of the workloads owing to its pool-based allocator. In particular, *Esp-Res* is the memory optimization option most similar to our proposed *IoT.js-SMBCA*. As many differences exist between Espruino and IoT.js in the structure of JavaScript objects and garbage collection, we implemented *IoT.js-DSA* for a fair comparison. Specifically, *IoT.js-DSA* is a variant of IoT.js that does not use the compressed addressing but uses the dynamic segment allocation.

In *IoT.js-Def*, *Esp*, and *IoT.js-SMBCA*, the maximum heap size is set to 128 KB. Unless otherwise specified, *IoT.js-SMBCA* sets the RMC size to 16. Options using dynamic segment allocation, such as *IoT.js-SMBCA* and *Esp-Res*, set the segment size to 2 KB.

We used a Raspberry Pi 3 and a Samsung Artik 053 for the experiment. The Raspberry Pi 3 is equipped with a quad-core 1.2 GHz Cortex-A53 CPU and a DRAM of 512 MB. The Artik 053 is equipped with a single-core 320 MHz Cortex-R4 CPU and an SRAM of 1280 KB.

In the experiment, we executed SunSpider benchmarks [11], V8 benchmarks [12], and real-world applications. We implemented real-world applications, such as *smart-web-cam*, *face-door*, and *lights-after-dark*, at first hand. Real-world applications are implemented under the assumption that the lightweight JavaScript engine and external libraries [5]–[10] run together. *Smart-web-cam* recognizes a keyword by periodically sampling the sound sensor and the sensor's value into a keyword-spotting model [6], and then reports the camera frame to a server when a keyword is recognized. *Face-door* periodically inputs a camera frame into the image classification model [7] to obtain the classification result. It also transmits an actuation command to the door lock device. *Lights-after-dark* periodically reads a light sensor and notifies another light controller device of an actuation command if the sensor value is above a pre-defined threshold.

B. OPTIMAL SEGMENT SIZE

The user memory used by SMBCA consists of three areas: data block area, free block area, SMBCA metadata area. The data block area is the area of blocks to store data, such as JavaScript objects. The free block area is the area of blocks that do not store anything. JavaScript heap comprises a data block area and a free block area. The SMBCA metadata area comprises a segment base table, a reverse map tree, and RMC. Each segment requires 8 B for the segment header table, 4 B for the segment base table, and 20 B for the reverse map tree. The RMC requires 8 B per reverse map entry.

As the lightweight JavaScript engines aim to operate at small memory capacity, the sizes of the free block area and the SMBCA metadata area must be minimized. In SMBCA, when the maximum heap size is fixed, the sizes of the free block area and SMBCA metadata area change depending on the segment size. As the segment size increases, the over-provisioning problem deteriorates, and the free block area becomes larger. If the segment size becomes extremely large and is equal to the maximum heap size, the SMBCA becomes the same technique as static heap reservation. Meanwhile, as the segment size decreases, the number of segments increases, and the size of the SMBCA metadata area increases. When the segment size becomes extremely small, the SMBCA metadata area becomes larger than the free block area. Between the two extreme cases, an optimal segment size that minimizes the memory overhead must be selected.

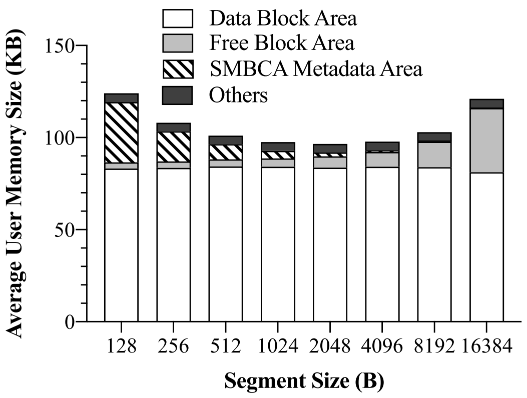


FIGURE 7. Average user memory size of IoT.js-SMBCA corresponding to segment size when executing the smart-web-cam application on Artik 053.

For example, Fig. 7 shows the average user memory size of *IoT.js-SMBCA* depending on the segment size when the *smart-web-cam* application runs. For this experiment, we modified IoT.js to measure the size of each area. We verified the measurement's accuracy by comparing it to the total heap size measured through the Valgrind tool. The others area in Fig. 7 is the difference between the total heap size and the total size of the three areas.

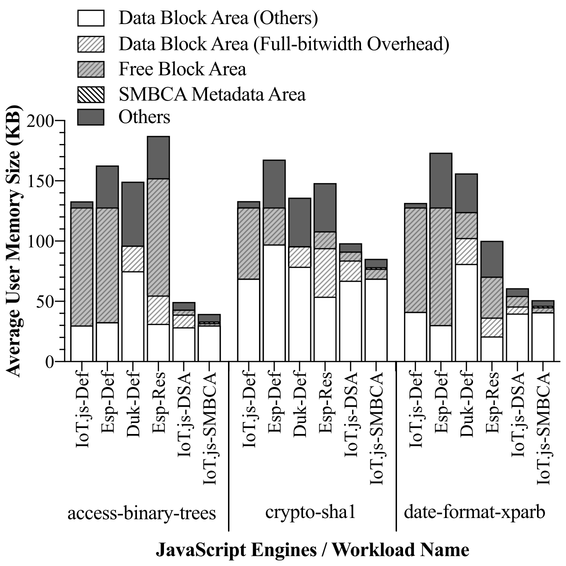
We measured the total user memory size once every 0.1 s and used the average of the total user memory sizes as the average user memory size. The application shows the smallest average user memory size when the segment size is 2 KB. SunSpider benchmarks, V8 benchmarks, and other real-world applications show a tendency similar to Fig. 7. Therefore, in this study, the segment size of SMBCA was set to 2 KB.

C. COMPARISON TO EXISTING LIGHTWEIGHT JavaScript ENGINES

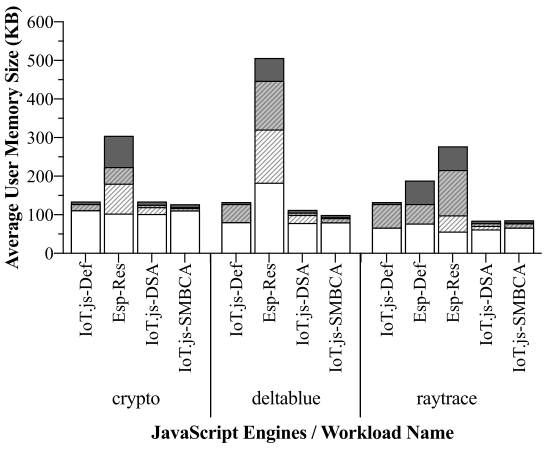
IoT.js-SMBCA shows a smaller user memory size than the existing JavaScript engines including Espruino [1], IoT.js [2], and Duktape [3] because it uses compressed addressing and dynamic segment allocation together. To prove this, we executed SunSpider benchmarks [11], V8 benchmarks [12], and real-world applications on the existing engines and on *IoT.js-SMBCA*. We also modified the Espruino and Duktape to measure each area's size in the JavaScript heap as we did on the IoT.js.

When each JavaScript engine executes the application, we measured the total user memory size once every 0.1 s and used the average of the sizes as the average user memory size. Fig. 8 and Fig. 9 show the experimental results. Full-bitwidth overhead implies the increased amount of the data block area size caused by not using the compressed addressing. *Duk*, *Esp-Res*, and *IoT.js-DSA* do not use the compressed addressing, whereas *IoT.js-Def*, *Esp*, and *IoT.js-SMBCA* do use it.

Some workloads were not used by some JavaScript engines. For example, as *Esp* uses a maximum 6-bit reference counter, it cannot express the numerous references



(a)



(b)

FIGURE 8. Average user memory size of IoT.js-SMBCA and existing lightweight JavaScript engines when executing (a) the SunSpider benchmark or (b) V8 benchmark.

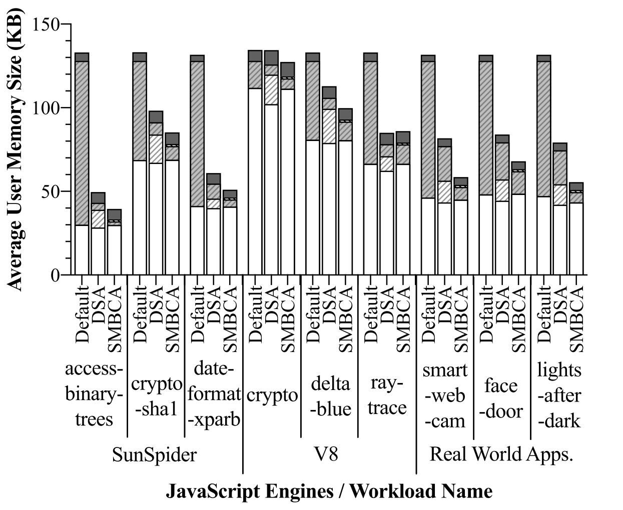


FIGURE 9. Average user memory size of variants of IoT.js when executing a benchmark or a real-world application.

among objects in *crypto* and *deltablue* workloads. As *Duk* rarely calls garbage collection, the V8 benchmark shows an average memory usage of several MBs. Because it makes no sense to compare *Duk* that shows too much memory usage, we excluded *Duk* from the V8 benchmark experiment. Given that both Duktape [3] and Espruino [1] were not ported to Artik 053, we used *IoT.js-DSA* instead.

Given that existing JavaScript engines have differences in their implementations of garbage collection, the engines show different data block area sizes. For example, *IoT.js-Def*, *Esp*, and *Esp-Res* call garbage collection in advance even when there is enough empty block area. Meanwhile, as *Duk* is implemented for a memory-abundant target, it calls garbage collection only in out-of-memory conditions, resulting in a larger data block area size than other engines. In addition, each JavaScript engine has a different structure of JavaScript objects. Even if the engines use similar garbage collection policies, they show different data block area sizes owing to the structures of the objects.

IoT.js-Def [2] and *Esp* [1] use single-base compressed addressing (SBCA) and static heap reservation. Owing to the SBCA, they can minimize the memory space occupied by pointers. Meanwhile, they must use the static heap reservation in order to allocate memory space to a JavaScript object directly, which finally induces over-provisioning problems. For example, owing to the over-provisioning problem, *IoT.js-Def* wastes a free block area of 81.3 KB on average when running the SunSpider benchmarks and 41.6 KB when running V8 benchmarks. Further, *Esp* wastes a free block area of 82.4 KB in the SunSpider benchmarks on average.

Because *Duk* uses dynamic object allocation, the over-provisioning problem does not occur. As *Esp-Res* [1] uses dynamic segment allocation, it shows a smaller free block area size than those of JavaScript engines based on static heap reservation. However, because both *Duk* and *Esp-Res* do not apply compressed addressing, the average full-bitwidth overheads of 19.9 KB and 26.5 KB, respectively, occur in SunSpider. *IoT.js-DSA* induces an average full-bitwidth overhead of 11.1 KB in SunSpider, 15.6 KB in V8, and 12.7 KB in real-world applications.

The full-bitwidth overhead differs depending on the workload in *IoT.js-DSA*. It is because each workload shows various characteristics of the data blocks. As shown in Fig. 10, 53.8% of data blocks, on average, include pointers. Compressed addressing can reduce the size of these blocks. For example, as the *crypto-sha1* workload shows that 53.4% of data blocks have pointers, its full-bitwidth overhead in *IoT.js-DSA* is 16.9 KB, as shown in Fig. 8(a).

Because *IoT.js-SMBCA* uses both dynamic segment allocation and compressed addressing, it shows a smaller user memory size than the existing engines. As *IoT.js-SMBCA* solves the over-provisioning problem using dynamic segment allocation, it reduces memory usage by 43.9% on average in

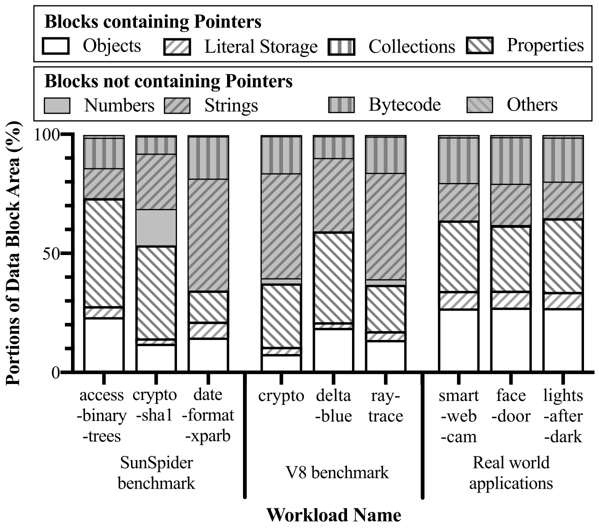


FIGURE 10. Breakdown of the data block area in IoT.js-DSA. Blocks with white backgrounds contain pointers, which can be compressed with compressed addressing. Meanwhile, blocks with grey backgrounds do not contain pointers.

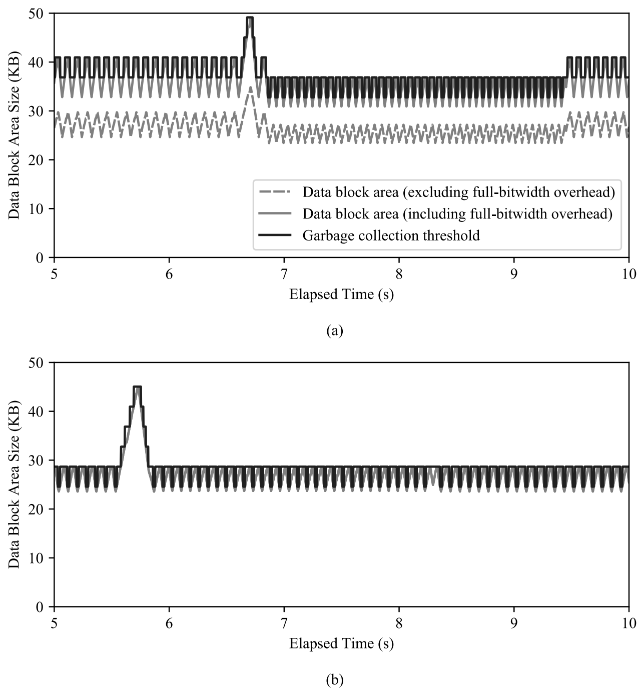


FIGURE 11. Data block area size that changes over time in (a) IoT.js-DSA and (b) IoT.js-SMBCA during the period from 5 s to 10 s when executing SunSpider's access-binary-trees workload.

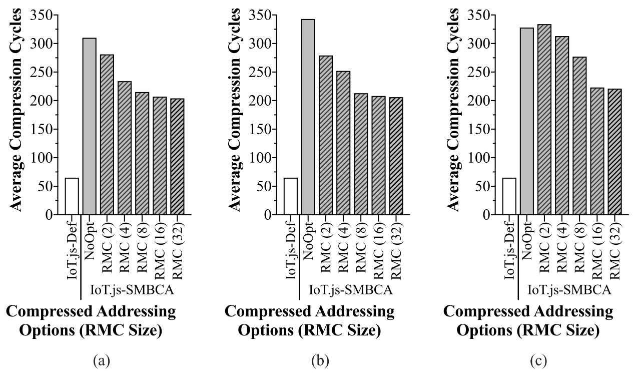


FIGURE 12. Average number of address compression cycles when IoT.js-SMBCA executes SunSpider benchmark workloads on a Raspberry Pi 3; (a) access-binary-trees, (b) crypto-sha1, (c) date-format-xparb.

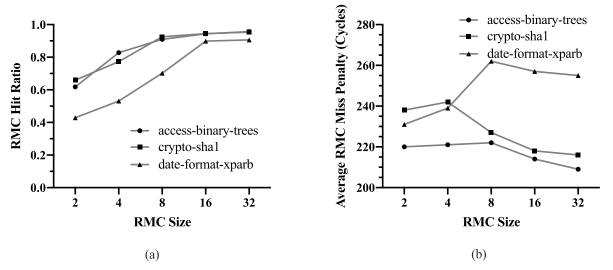


FIGURE 13. (a) Hit ratio and (b) miss penalty of RMC when executing SunSpider benchmark workloads. Miss penalty is the number of cycles consumed for accessing the RMC.

all workloads compared to other engines based on static heap reservation, as shown in Fig. 8. Because *IoT.js-SMBCA* uses compressed addressing to remove full-bitwidth overhead, it reduces memory usage by 14.4% on average compared to *IoT.js-DSA* as shown in Fig. 9.

In particular, as *IoT.js-SMBCA* has a smaller data block size than *IoT.js-DSA*, *IoT.js-SMBCA* calls fewer garbage collections than *IoT.js-DSA* by 1.35-2.71 times. For example, as shown in Fig. 11, the data block size in *IoT.js-SMBCA* increases more slowly than in *IoT.js-DSA* because *IoT.js-SMBCA* does not include the full-bitwidth overhead. As *IoT.js-SMBCA* has a smaller data block size than *IoT.js-DSA*, it is harder for the data block size to reach the garbage collection threshold in *IoT.js-SMBCA* than in *IoT.js-DSA*. In this case, *IoT.js-SMBCA* shows 89 garbage collections, whereas *IoT.js-DSA* shows 144 garbage collections during the period from 5 s to 10 s.

D. ADDRESS TRANSLATION LATENCY

Because SMBCA uses MBCA, the latency of address decompression and address compression is longer than that of SBCA. In particular, in order to reduce the address compression latency, we propose RMC.

We measured the number of clock cycles consumed in address decompression and address compression when *IoT.js-Def* or *IoT.js-SMBCA* executes SunSpider benchmarks on a Raspberry Pi 3. We used a performance monitoring unit (PMU) in Raspberry Pi 3 to measure the clock cycles.

In *IoT.js-Def*, the average number of decompression cycles is 58 cycles, whereas it increases to 100 cycles in *IoT.js-SMBCA*. In the case of address compression, the average number of compression cycles in *IoT.js-Def* is 65 cycles, whereas it increases to 327 cycles in *IoT.js-SMBCA*. If RMC is applied to *IoT.js-SMBCA*, the average number of compression cycles decreases, as shown in Fig. 12. As the RMC size increases, the average number of compression cycles further decreases owing to the increasing RMC hit ratio.

The hit ratio and miss penalty of the RMC differ depending on the characteristics of the workload. For example, as shown in Fig. 13(a) and (b), *date-format-xparb* shows the lower

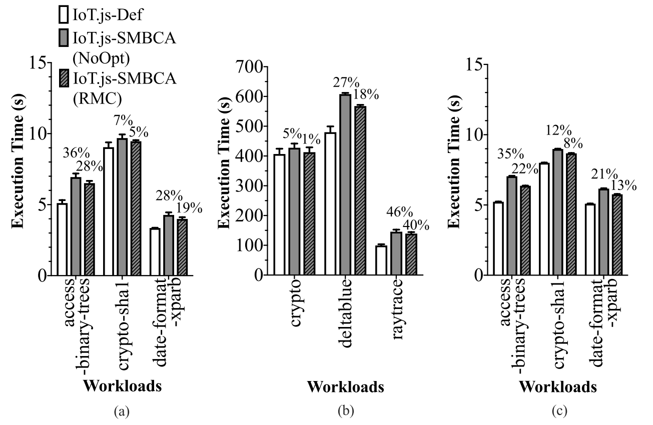


FIGURE 14. Total execution time when IoT.js-SMBCA executes benchmark workloads. A number written above a bar indicates the relative amount of execution-time overhead compared to IoT.js-Def; (a) SunSpider on Raspberry Pi 3, (b) V8 on Raspberry Pi 3, (c) SunSpider on Artik 053.

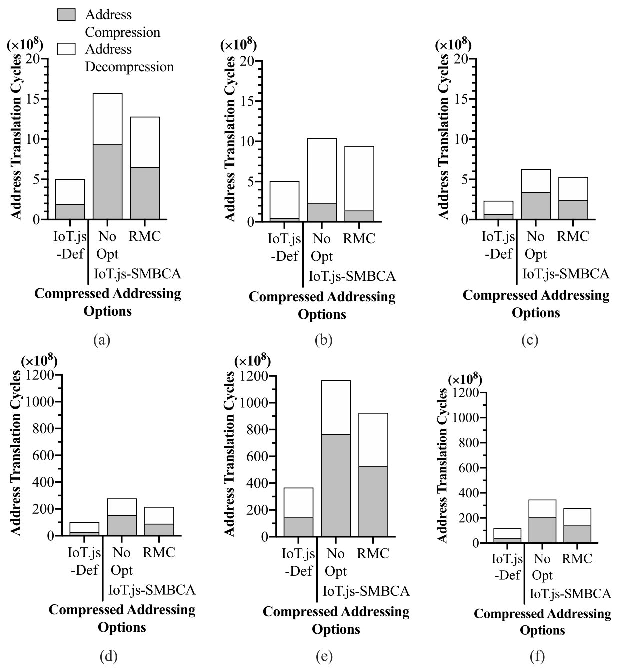


FIGURE 15. Address translation cycles when IoT.js-SMBCA executes the SunSpider or V8 benchmark. A number written above a bar indicates the relative degradation of address translation cycles compared to IoT.js-Def; (a) access-binary-trees (SunSpider), (b) crypto-sha1 (SunSpider), (c) date-format-xparb (SunSpider), (d) crypto (V8), (e) deltablue (V8), (f) raytrace (V8).

hit ratio and the longer miss penalty than *access-binary-trees*. Therefore, when the RMC size is 4, *access-binary-trees* shows average compression cycles of 234 cycles, whereas *date-format-xparb* shows average compression cycles of 313 cycles. When the RMC size is 16, the hit ratio of all workloads exceeds 90%, and the average number of compression cycles converges to 213 cycles, which is 34.9% less than for *IoT.js-SMBCA* with no optimization (*NoOpt*). Therefore, in this study, the RMC size was set to 16.

Fig. 14 shows the total execution time of several benchmarks [11], [14] on a Raspberry Pi 3 and an Artik 053 depending on the compressed addressing options, such as MBCA and RMC. When running SunSpider benchmarks, *IoT.js-SMBCA* with RMC shows an average execution-time overhead of 17.1%, whereas one without the RMC induces an average execution-time overhead of 23.5%. When running V8 benchmarks, *IoT.js-SMBCA* with RMC shows an average overhead of 19.9%, whereas one without the RMC induces an average overhead of 26.1%.

The overhead of *IoT.js-SMBCA* also differs depending on the number of address translation cycles in each workload. For example, as shown in Fig. 15, SunSpider's *access-binary-trees* and V8's *deltablue* show more address compression cycles than other workloads. Therefore, as shown in Fig. 14(a) and (b), these workloads have 36% and 27% execution-time overhead, respectively. By applying the RMC, the overheads of both workloads are reduced to 28% and 18%, respectively. The improvement in the *IoT.js-SMBCA* overhead by the RMC also can be found in Artik 053, as shown in Fig. 14(c).

VI. CONCLUSION AND FUTURE WORK

Existing lightweight JavaScript engines cannot use resizable heap as they use memory optimization techniques targeting low-end devices equipped with small on-chip memories. However, as the functional requirements for low-end devices have increased in recent years, these devices are being equipped with large on-chip memories and use several external libraries for machine learning and connectivity. In this study, we proposed SMBCA, a memory optimization technique of the JavaScript engine suitable for recent low-end devices. SMBCA not only enables a resizable heap through dynamic segment allocation, but also enables compressed addressing to be applied to large memory through MBCA. Experimental results on JavaScript benchmarks show that IoT.js that applies SMBCA had a 43.9% smaller memory footprint than the existing lightweight JavaScript engines, because SMBCA solves the over-provisioning problem while using compressed addressing. To minimize the memory accesses during the address compression operation of SMBCA, we proposed an optimization technique called RMC. We demonstrated that SMBCA with RMC reduced the address compression latency by 34.9% on average when running the SunSpider benchmark on a Raspberry Pi 3 and an Artik 053. However, the RMC cannot reduce the address decompression latency. An inline caching technique or a dedicated hardware for address decompression can be used to reduce the overhead of SMBCA further. We leave the further optimization of address decompression as future work.

REFERENCES

[1] *Espruino*. Accessed: Sep. 4, 2020. [Online]. Available: http://www.espruino.com

[2] E. Gavrin, S. Lee, R. Ayrapetyan, and A. Shitov, "Ultra lightweight JavaScript engine for Internet of Things," in *Proc. ACM SIGPLAN Int. Conf. Syst. Program. Lang. Appl. Softw. Hum.*, 2015, pp. 19–20.

[3] *Duktape*. Accessed: Sep. 4, 2020. [Online]. Available: http://www.duktape.org

[4] *Micropython*. Accessed: Sep. 4, 2020. [Online]. Available: http://www.micropython.org

[5] A. Capotondi, M. Rusci, M. Fariselli, and L. Benini, "CMix-NN: Mixed low-precision CNN library for memory-constrained edge devices," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 5, pp. 871–875, May 2020.

[6] Y. Zhang, N. Suda, L. Lai, and V. Chandra, "Hello edge: Keyword spotting on microcontrollers," 2017, *arXiv:1711.07128*. [Online]. Available: http://arxiv.org/abs/1711.07128

[7] L. Lai, N. Suda, and V. Chandra, "CMSIS-NN: Efficient neural network kernels for arm Cortex-M CPUs," 2018, *arXiv:1801.06601*. [Online]. Available: http://arxiv.org/abs/1801.06601

[8] J. Yu, A. Lukefahr, R. Das, and S. Mahlke, "TF-net: Deploying sub-byte deep neural networks on microcontrollers," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–21, Oct. 2019.

[9] C. Lee, J. Jo, J. Lee, D. An, J. Cho, and R. Jung, "RT-OCF: A lightweight device-to-device framework for the Internet of Things," in *Proc. 3rd Int. Conf. Internet Things (ICIOT)*, vol. 2018, pp. 179–187.

[10] J. Jo, J. Cho, R. Jung, and H. Cha, "IoTivity-lite: Comprehensive IoT solution in a constrained memory device," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, Oct. 2018, pp. 1367–1369.

[11] *SunSpider JavaScript Benchmark*. Accessed: Sep. 4, 2020. [Online]. Available: https://webkit.org/perf/sunspider/sunspider.html

[12] *V8 Benchmark*. Accessed: Sep. 4, 2020. [Online]. Available: https://v8.dev/docs/benchmarks

[13] L. P. Deutsch and A. M. Schiffman, "Efficient implementation of the Smalltalk-80 system," in *Proc. 11th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1984, pp. 297–302.

[14] *V8 JavaScript Engine*. Accessed: Sep. 4, 2020. [Online]. Available: http://v8.dev

[15] M. Kim, H.-J. Jeong, and S.-M. Moon, "Small footprint JavaScript engine," in *Components and Services for IoT Platforms*. Cham, Switzerland: Springer, 2017, pp. 103–116.

[16] R. Morales, R. Saborido, and Y.-G. Guéhéneuc, "Momit: Porting a javascript interpreter on a quarter coin," *IEEE Trans. Softw. Eng.*, early access, Jan. 22, 2020, doi: 10.1109/TSE.2020.2968061.

[17] D. Li, B. Huang, L. Cui, and Z. Xu, "WebletScript: A lightweight distributed JavaScript engine for Internet of Things," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2018, pp. 1–6.

GYEONGHWAN HONG received the B.E. degree in computer engineering from Sungkyunkwan University, Suwon, South Korea, in 2013, where he is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering. His research interests include embedded software, mobile system software, and the Internet of Things.

DONGKUN SHIN (Member, IEEE) received the Ph.D. degree in computer science and engineering from Seoul National University, Seoul, South Korea, in 2004. He is currently a Professor with the Department of Electrical and Computer Engineering, Sungkyunkwan University, Suwon, South Korea. From 2004 to 2007, he was a Senior Engineer with Samsung Electronics, South Korea. His research interests include embedded software, low-power systems, computer architecture, and real-time systems.