Architectural Support for Server-Side PHP Processing

Dibakar Gope David J. Schlais Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin - Madison
Madison, WI, USA
gope@wisc.edu,schlais2@wisc.edu,mikko@engr.wisc.edu

ABSTRACT

PHP is the dominant server-side scripting language used to implement dynamic web content. Just-in-time compilation, as implemented in Facebook's state-of-the-art HipHopVM, helps mitigate the poor performance of PHP, but substantial overheads remain, especially for realistic, large-scale PHP applications. This paper analyzes such applications and shows that there is little opportunity for conventional microarchitectural enhancements. Furthermore, prior approaches for function-level hardware acceleration present many challenges due to the extremely flat distribution of execution time across a large number of functions in these complex applications. In-depth analysis reveals a more promising alternative: targeted acceleration of four fine-grained PHP activities: hash table accesses, heap management, string manipulation, and regular expression handling. We highlight a set of guiding principles and then propose and evaluate inexpensive hardware accelerators for these activities that accrue substantial performance and energy gains across dozens of functions. Our results reflect an average 17.93% improvement in performance and 21.01% reduction in energy while executing these complex PHP workloads on a state-of-the-art software and hardware platform.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; **Special purpose systems**;

KEYWORDS

PHP, dynamic languages, domain-specific accelerators

ACM Reference format:
Dibakar Gope David J. Schlais Mikko H. Lipasti Department of Electrical and Computer Engineering University of Wisconsin - Madison Madison, WI, USA . 2017. Architectural Support for Server-Side PHP Processing. In *Proceedings of ISCA '17, Toronto, ON, Canada, June 24-28, 2017,* 14 pages.
https://doi.org/10.1145/3079856.3080234

1 INTRODUCTION

In recent years, the importance and quantity of code written in dynamic scripting languages such as PHP, Python, Ruby and Javascript has grown considerably. Among all scripting languages used for server-side web development, PHP is the most commonly used [70, 72], representing about 82.3% [13] of all web applications. In particular, server-side PHP web applications have created an ecosystem of their own in the world of web development. They are used to build and maintain websites and web applications of all sizes and complexity, ranging from small websites to complex large scale enterprise management systems. PHP powers many of the most popular web applications, such as Facebook and Wikipedia. Despite their considerable increase in popularity, their performance is still the main impediment for deploying large applications. Since these PHP applications run on live datacenters hosting millions of such web applications, even small improvements in performance or utilization will translate into immense cost savings.

The desire to reduce datacenter load inspired the design of the HipHop JIT compiler (HHVM) [19, 72], which translates PHP to native code. HHVM demonstrates a significant performance increase for a set of micro-benchmark suites. However, runtime characteristics of popular real-world PHP web applications (e-commerce platforms, social networking sites, online news forums, banking servers, etc.) are found to be dramatically different than the de-facto benchmark suites SPECWeb2005 [10], bench.php [18], and the computer language benchmarks [1] used so far for evaluating the performance of web servers. These micro-benchmark suites have been the primary focus for most architectural optimizations of web servers [20]. Furthermore, these micro-benchmarks spend most of their time in JIT-generated compiled code, contrary to the real-world PHP applications that tend to spend most of their time in various library routines of the VM.

Figure 1 depicts the distribution of CPU cycles spent in the hottest leaf functions of a few real-world PHP applications compared to SPECWeb2005's banking and e-commerce workloads. Clearly, the SPECWeb2005 workloads contain significant hotspots – with very few functions responsible for about 90% of their execution time. However, the PHP web applications exhibit significant diversity, having very flat execution profiles – the hottest single function (JIT compiled code) is responsible for only $10-12\%$ of cycles, and they take about 100 functions to account for about 65% of cycles. This tail-heavy behavior presents few obvious or compelling opportunities for microarchitectural optimizations.

In order to understand the microarchitectural implications (performance and energy-efficiency bottlenecks) of these workloads with hundreds of leaf functions, we undertook a detailed architectural characterization of these applications. Despite our best effort, we could not find any obvious target or easy opportunity for architectural optimization.

ISCA '17, June 24-28, 2017, Toronto, ON, Canada

ACM ISBN 978-1-4503-4892-8/17/06…$15.00
https://doi.org/10.1145/3079856.3080234

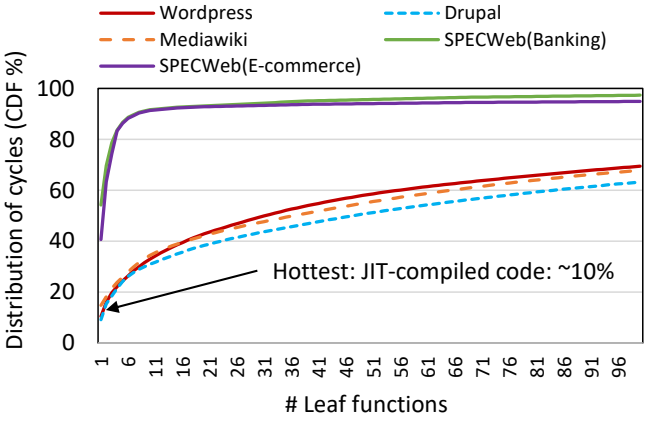


Figure 1: Distribution of CPU cycles of SPECWeb2005 workloads and few real-world PHP web applications over leaf functions (details in evaluation section).

As the processor industry continues to lean towards customization and heterogeneity [41, 54, 65, 67] to improve performance and energy efficiency, we seek to embrace domain-specific specializations for these real-world PHP applications. We note function level specialization is not a viable solution for these applications given their very flat execution profiles. However, a closer look into the leaf functions' overall distribution reveals that many leaf functions suffer from either the abstraction overheads of scripting languages (such as type checking [22], hash table accesses for user-defined types [31, 32], etc.) or the associated overhead of garbage collected languages [25]. These observations guide us to apply several hardware and software optimization techniques from prior works [22, 31, 32, 40] *together* to these PHP applications in order to minimize those overheads. After applying these optimizations, a considerable fraction of their execution time falls into four major categories of activities – hash map access, heap management, string manipulation, and regular expression processing. These four categories show the potential to improve the performance and energy efficiency of many leaf functions in their overall distribution. This motivates us to develop specialized hardware to accelerate these four major activities.

Hash Table Access. Unlike micro-benchmarks that mostly accesses hash maps with static literal names, these real-world applications often tend to exercise hash maps in their execution environment with dynamic key names. These accesses cannot be converted to regular offset accesses by software methods [31, 32, 40]. Programmability and flexibility in writing code are two of several reasons to use dynamic key names. In order to reduce the overhead and inherent sources of energy inefficiency from hash map accesses in these PHP applications, we propose to deploy a hash table in hardware. This hash table processes both GET and SET requests entirely in hardware to satisfy the unique access pattern of these PHP applications, contrary to prior works deploying a hash table that supports only GET requests in a memcached environment [55]. Furthermore, supporting such a hash table in the PHP environment presents a new set of challenges in order to support a rich set of PHP features communicating with hash maps that these real-world PHP applications tend to exercise often.

Heap Management. A significant fraction of execution time in these applications come from memory allocation and deallocation, despite significant efforts to optimize them in software [37, 51]. Current software implementations mostly rely on recursive data structures and interact with the operating system, which makes them non-trivial to implement in hardware. However, these applications exhibit an unique memory usage pattern that allows us to develop a heap manager with its most frequently accessed components in hardware. It can respond to most of the memory allocation and deallocation requests from this small and simple hardware structure.

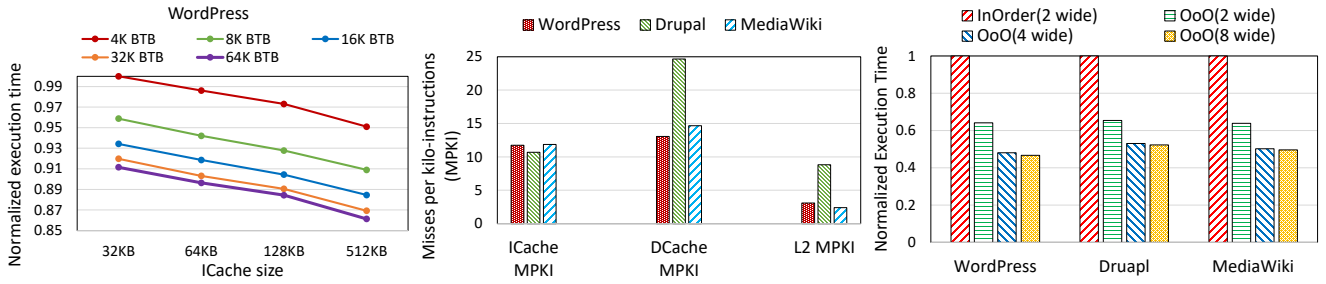
String Functions. These PHP applications exercise a variety of string copying, matching, and modifying functions to turn large volumes of unstructured textual data (such as social media updates, web documents, blog posts, news articles, and system logs) into appropriate HTML format. Prior works [68] have realized the potential of hardware specialization for string matching, but do not support all the necessary string functions frequently used in these PHP applications. Nevertheless, designing separate accelerators to support all the necessary string functions will deter their commercial deployment. Surprisingly, all the necessary string functions can be supported with a few common sub-operations. We propose a string accelerator that supports these string functions by accelerating the common sub-operations rather than accelerating each function. It processes multiple bytes per cycle for concurrency to outperform the currently optimal software with SSE extensions.

Regular Expressions. These PHP applications also use regular expressions (regexps) to dynamically generate HTML content from the unstructured textual data described above. Software-based regexp engines [8, 57] or recent hardware regexp accelerators [34, 39, 58, 66] are overly generic in nature for these PHP applications as they do not take into consideration the inherent characteristics of the regular expressions in them. We introduce two novel techniques – *Content Sifting* and *Content Reuse* – to accelerate the execution of regexp processing in these PHP applications and achieve high performance and energy efficiency. These two techniques significantly reduce the *repetitive* processing of textual data during regular expression matching. They essentially exploit the content locality across a particular or a series of consecutive regular expressions in these PHP applications.

The remainder of this paper is organized as follows. Section 2 performs an in-depth microarchitectural analysis of the PHP applications. Failing to find any clear microarchitectural opportunities shifts our focus towards designing specialized hardware. We discover the potential candidates for specialization in Section 3. In Section 4 we design specialized hardware accelerators for them. Section 5 presents results. Section 6 discusses related work and Section 7 concludes the paper.

2 MICROARCHITECTURAL ANALYSIS

We begin by performing an in-depth architectural characterization of the PHP applications to identify performance and energy-efficiency bottlenecks (if any) in them. We use gem5 [4] for our architectural simulation. Surprisingly, the most significant bottlenecks lie in the processor front-end, with poor branch predictor and branch target buffer performance. Neither increased memory bandwidth, nor larger



(a) Sensitivity to BTB and ICache sizes, normalized to 4K BTB 32KB ICache. Other PHP applications show similar behavior.

(b) Cache analysis

(c) In-order vs. out-of-order behavior

Figure 2: Microarchitectural characterization of the PHP applications.

instruction or data caches show significant opportunity for improving performance.

Branch predictor bottlenecks. We experimented with the state-of-the-art TAGE branch predictor [63][1] with 32KB storage budget. The branch mispredictions per kilo-instructions (MPKI) for the three PHP applications considered in this work are 17.26, 14.48, and 15.14. Compared to the 2.9 MPKI average for the SPEC CPU2006 benchmarks, these applications clearly suffer from high misprediction rates. The poor predictor performance is primarily due to the presence of large number of data-dependent branches in the PHP applications. The outcomes of data-dependent branches depend solely on unpredictable data that the PHP applications require to process during most of their execution time. Prior work on predicting data-dependent branches [35] may improve the MPKI of the PHP applications.

Branch target buffer bottlenecks. Real-world PHP applications suffer significantly from the poor performance of branch target buffers (BTBs) in server cores. We simulate a BTB that resembles the BTB found in modern Intel server cores with 4K entries and 2-way set associativity. Such behavior stems from the large number of branches in the PHP applications. Around 12% of all dynamic instructions are branches in the SPEC CPU2006 workloads [73], whereas in the PHP applications about 22% of all instructions are branches, thus adding more pressure on BTB. Figure 2(a) shows the execution time of one of the PHP applications, as the BTB size is progressively increased from 4K entries to 64K entries for different sizes of I-cache. However even with 64K entries, the PHP application obtains a modest BTB hit rate of 95.85%. Deploying such large BTBs is not currently feasible in server cores due to power constraints.

Cache analysis. Figure 2(b) presents the cache performance. L1 instruction and data cache behavior are more typical of SPEC CPU-like workloads contrary to the instruction cache behavior observed in prior works with other server-side applications (servers-side Javascript applications [73] or memcached workloads [55]). Note that we simulate an aggressive memory system with prefetchers at every cache level. Although there are hundreds of leaf functions in the execution time distribution of these PHP applications, they are compact enough that can be effectively cached in the L1. Besides, the numerous data structures in these PHP applications do not appear to stress the data cache heavily. The L2 cache has very low MPKI, as the L1 filters out most of the cache references. Figure 2(a) shows the potential of minor performance gain with very large instruction caches.

In-order vs. out-of-order. Figure 2c shows the impact four different architectures (2-wide in-order, 2-wide out-of-order (OoO), 4-wide OoO, and 8-wide OoO) had on workload execution time. Changing from in-order to OoO cores shows a significant increase in performance. We also note that the 4-wide OoO shows fairly significant performance gains over the 2-wide OoO architecture, hinting that some ILP exists in these workloads. However, increasing to an 8-wide OoO machine shows very little (< 3%) performance increase, hinting that ILP cannot be exploited for large performance benefits beyond 4-wide OoO cores.

Overall, our analysis suggests that PHP applications require far more BTB capacity and much larger caches than server cores currently provide to obtain even minor performance benefit. In short, our analysis does not present any obvious potential target for microarchitectural optimizations.

3 MITIGATE ABSTRACTION OVERHEAD

As microarchitectural analysis fails to reveal any clear opportunities for improvement, we shift our focus towards augmenting the base processor with specialized hardware accelerators. However, function-level acceleration is not an appealing solution for these applications given their very flat execution profiles. Nevertheless, it is commonly known that PHP-like scripting languages suffer from the high abstraction overheads of managed languages. Overhead examples include dynamic type checks for primitive types, and hash table accesses for user-defined types. Furthermore PHP, like all garbage collected languages, suffers from the overhead of reference counting.

While there are many research proposals [22, 31, 32, 40] from the academic community in mitigating each of these abstraction overheads *separately*, most of them have not been adopted so far by the industry into commercial server processors. Considering the fact that these abstraction overheads constitute a significant source of performance and energy overhead in these PHP applications (as our results indicate), we believe the industry is more likely to embrace these

[1]The branch prediction accuracy observed on Intel server processors and TAGE are in the same range [60]

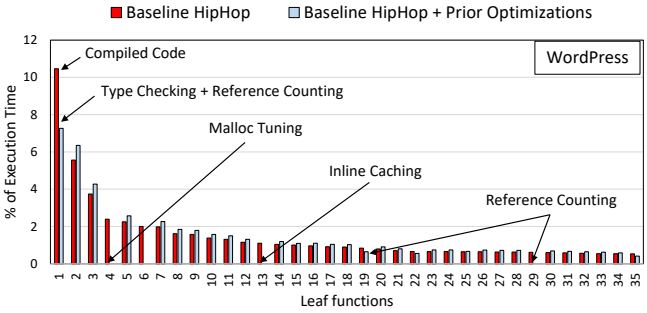


Figure 3: Contribution of leaf functions to the execution time of WordPress before and after applying all optimizations.

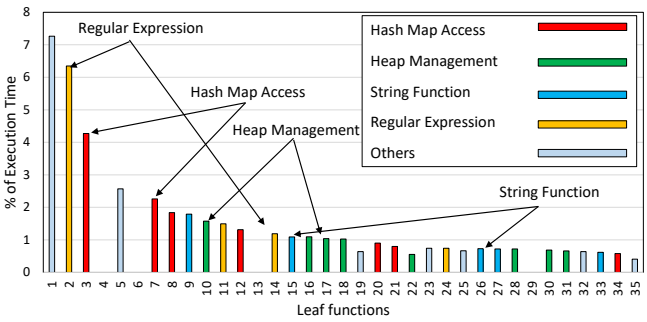


Figure 4: Categorization of leaf functions of WordPress into major categories.

proposals sooner than later. So in order to mitigate the abstraction overheads and get a clear view of what other *fundamental* activities are going to dominate the execution time of these PHP applications in near future, we apply several hardware and software optimizations from prior research *together* to these applications in our simulated environment. Note that the objective of this exercise is not only to move the abstraction overheads towards the tail of the distribution of these applications, but also to determine the fundamental dominant activities in many of the leaf functions which were obscured by these overheads that have known solutions. Next, we describe briefly those optimizations from prior research proposals.

Inline Caching [31, 32] and Hash Map Inlining [40]. Modern JIT compilers [14, 19] use *inline caching* (IC) to specialize code that accesses members in dynamically-typed objects. With IC, access to a dynamically-typed object is specialized to a simple offset access from the start of the object. A type check around that offset access ensures that the assumptions used in the generation of specialized code hold at runtime. We extend the IC technique to specialize these PHP applications' accesses to hash maps as done in [40]. Further, we adopt the recent proposal on *hash map inlining* [40] (HMI) to specialize hash map accesses with variable though predictable key names.

Type Checking. As discussed above, specialized code for accessing variables of primitive or user-defined types now requires run-time type checks. We adopted a technique from prior work [22] to mitigate this overhead in hardware. With this technique, the cache subsystem performs the required type check for a variable before returning its value.

Reference Counting. Reference counting constitutes a major source of overhead in these PHP applications as it is spread across compiled code and many library functions. We adopted a hardware proposal from prior work that introduces minimal changes to the cache subsystem to mitigate this overhead [46].

In addition to applying the above four optimizations, we tuned the applications to reduce their overhead from expensive memory allocation and deallocation calls to the kernel.

Figure 3 demonstrates the effect of applying these above optimizations to the leaf functions of one of the PHP applications, WordPress [15]. The left bar shows the contribution of the leaf functions to the overall execution time before applying the optimizations, whereas the right bar shows their corresponding contribution after applying the optimizations. Clearly, the contribution of many leaf functions diminishes with these optimizations (indicated by arrows), and as a result, the contributions of the remaining functions in the overall distribution have gone up. More interestingly, many of the leaf functions in the overall distribution now fall into four major categories – hash map access, heap management, string manipulation, and regular expression processing, as shown by the different color coding in Figure 4. This consequently presents opportunities to accelerate them in hardware to obtain performance and energy efficiency. Figure 5 shows the execution time breakdown of a few real-world PHP applications after applying the above optimizations. If a function contained aspects of one of the four categories, we grouped it into execution time for that category.[2] Since these four categories show the potential to improve the execution efficiency of many leaf functions, we propose specialized hardware to accelerate these activities.

4 SPECIALIZING THE GENERAL-PURPOSE CORE

In this section, we propose accelerators for the four major hot spots observed in the PHP applications. We first describe the design principles that we followed while developing these accelerators and then describe their hardware design.

4.1 Accelerator Design Principles

Recent explorations in cache-friendly accelerator design demonstrate the criticality and feasibility of balancing the efficiency of application specific specializations with general-purpose programmability using tightly-coupled accelerators [64]. We espouse a similar accelerator design philosophy, and propose accelerators that adhere to the following design principles so that they fit naturally into multi-core server SoCs.

a) They are VM and OS agnostic. The VM still observes the same view of software data structures in memory.

b) They have cache interfaces and participate in the cache coherence mechanism.

c) They only accelerate the frequently-executed common path through each function. Any unusual or unexpected condition is relegated to a software handler.

[2]For the few functions containing aspects of multiple categories, it was placed in the category it spent more of its execution time.

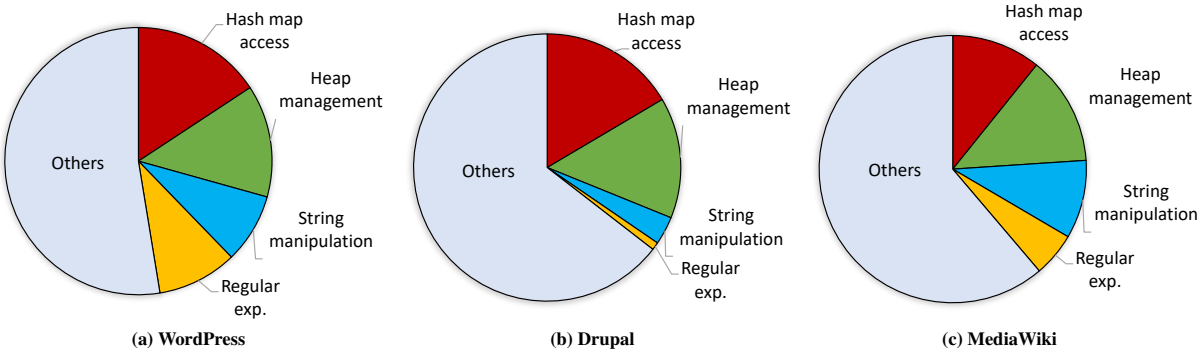


Figure 5: Execution time breakdown after mitigating the abstraction overheads.

d) They are tightly coupled and are invoked via a small set of single-cycle instruction extensions to the general-purpose ISA.

e) They rely on a shared virtual address space and maintain a coherent view of memory to avoid the need for explicitly managed scratchpad memories. Our evaluation shows that most memory references from these accelerators are small and fall within the boundaries of a single 4KB page, so a single TLB lookup performed by the invoking instruction is sufficient. The hardware for managing coherence is necessarily somewhat complex, but, in our view, well worth the effort since it dramatically simplifies deployment of these accelerators in a realistic multicore system.

4.2 Hash Table

In order to reduce the overhead from hash map accesses, we deploy a hash table in hardware.

Overview Real-world PHP applications frequently access hash maps with dynamic key names; such accesses cannot be converted to efficient offset references by software methods [31, 32, 40]. Typically, these applications use many PHP commands that access short-lived hash maps using dynamic key names. For example, the PHP *extract* command is commonly used to import key-value pairs from a hash map into a local symbol table[3] in order to communicate their values later to an appropriate application template that is responsible for generating some dynamic content. Populating such a symbol table always occurs using dynamic key names. Furthermore, these PHP applications often store key-value pairs in a global or local symbol table to communicate their values to other functions in the appropriate scope. For example, the regular expression manager shares a search pattern (key) and its FSM table (value) with other appropriate functions through a hash map. Accesses to all such hash maps occur using dynamic key names. More importantly, such accesses to hash maps ease programming while developing large applications.

Proposed design Figure 6 shows the hash table accelerator design. A critical requirement with hardware-traversable hash tables is to bound the number of hash table entries accessed per lookup. Thus, when a key is looked up in the hash table in our design, several consecutive entries are accessed in parallel, starting from the first indexed entry, to find a match. A hash table lookup in hardware thus

[3] A symbol table is implemented using a hash map.

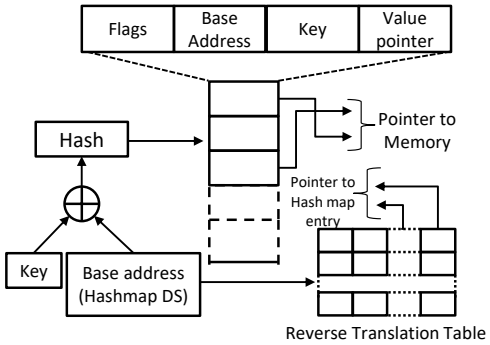


Figure 6: Hardware hash table.

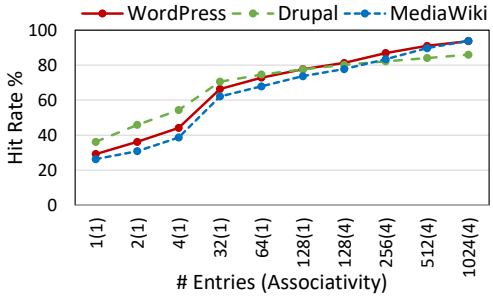


Figure 7: Hash table hit rate.

reduces control-flow divergence and exploits parallelism with hash map accesses. Note that it is not easy to extract parallelism from a serial hash map walk in software because of the complex control flow, memory aliasing, and numerous loop-carried dependencies.

Each hash table entry contains a string field to store a key, an 8-byte address (base address of a hash map structure in memory), a pointer to the memory location of the value, a dirty bit to indicate if the hash map structure in memory is required to be updated with the corresponding key, and a valid bit. The valid and dirty bits assist in replacing entries and making space for new incoming keys. The 8-byte address field contains the base address of a hash map data structure, accesses to which for a key-value pair the hash table

attempts to provide a fast lookup. Thus, in response to a request, the hash table performs a hash on the combined value of the key and the base address of the requested hash map to index into an entry and begin the lookup process. Starting with the entry, when a hash table lookup is performed, each entry has its key and 8-byte base address compared with the key and the base address of the request. Upon a match, the hash table updates the entry's last-access time stamp (for LRU replacement) and sends the response to the request. If no match is found, control falls back to the software to perform the regular hash map access in memory.

Design considerations The HHVM JIT compiler [19] already uses an efficient hash computation function that can operate on variable length strings (in our case the combination of a hash map base address and a key in it). However, we used a simplified hash function for the hash table without compromising its hit rate. This is because the HHVM hash function is overly complex to map into an efficient hardware module and it requires many processor cycles to compute a hash. Our design leverages several inherent characteristics of these PHP applications.

First, in contrast to the most large-scale memcached deployments [23, 55] where the GET requests vastly outnumber the SET and other request types, these PHP applications observe relatively higher percentage of SET requests (ranging from $15-25\%$) when generating dynamic contents for web pages. As a result, a hash table deployed for such applications should respond to both GET and SET requests in order to take full advantage of the hardware hash table and offload most operations associated with hash map accesses from the core.

Second, the majority (about 95%) of the hash map keys accessed in these PHP applications are at most 24 bytes in length. As a result, we store the keys in the hash table itself, unlike the hash table designed for memcached deployments [55]. Storing the keys directly in the hash table eases the traversal of the hash table in hardware.

We next discuss typical operations offloaded from a traditional software hash map to our hardware hash table. Note that the allocation or the deallocation of a hash map structure in memory is still handled by software.

GET: A GET request attempts to retrieve a key-value pair of the requested hash map from the hash table without any software interaction. Upon a match, the hash table updates the entry's last-access time stamp. If the key is not found for the requested hash map, control transfers to the software to retrieve the key-value pair from memory and places it into the hash table. In order to make space for the retrieved key, if an invalid entry is not found, then a clean entry (dirty bit not set) is given more priority for replacement. This avoids any costly software involvement associated with replacing a dirty entry from the hardware hash table. The hash map of a dirty entry is not up-to-date in memory with the key-value pair and therefore requires software intervention in our design to be updated when the entry is evicted. If none is found, the LRU dirty entry is replaced with incurring the associated software cost.

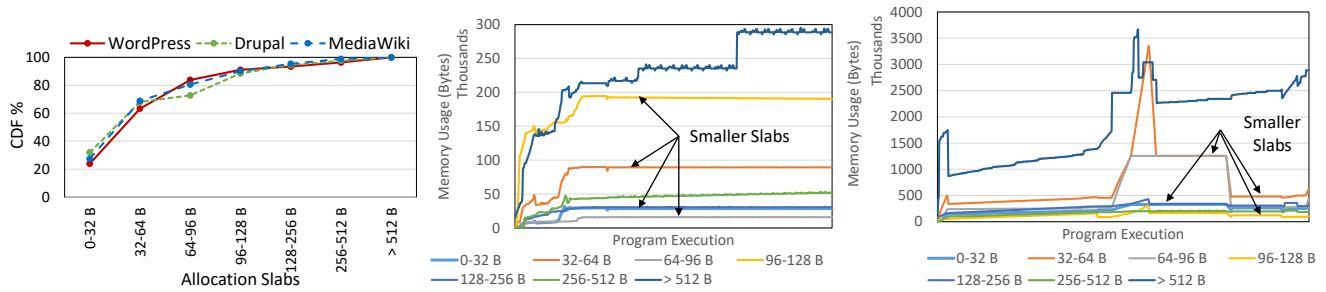
SET: A SET request attempts to insert a key-value pair of the requested hash map into the hash table without any software interaction. Upon finding a match, SET simply updates the value pointer and the entry's last-access time stamp. If the key is not found, then the key-value is inserted into the hash table and the entry's dirty bit is set. If an eviction is necessary (due to hash table overflow), the same replacement policy as described for GET is followed. Note that a SET operation silently updates the hash table in our design without updating the memory. Figure 7 demonstrates the hit rate of such a hash table. Even a hash table with only 256 entries observes a high hit rate of about 80%. Since SET operations never miss in our design, a hash table with very few entries (1, 2 or 4) shows such a decent hit rate. Having support of a SET operation in hardware helps in serving a major fraction of the short-lived hash map accesses from the hash table.

Free: In response to deallocating a hash map from the memory, the hardware hash table would (in a naive implementation) need to scan the entire table to determine the set of entries belonging to the requested hash map. The reverse translation table (RTT) in Figure 6 assists the hash table during this seemingly expensive operation. The RTT is indexed by the base address of a requested hash map. Each RTT entry stores back pointers to the set of hash table entries containing key-value pairs of a hash map. Each RTT entry also has a write pointer. The write pointer assists in adding these back-pointers associated with different key-value pairs of a hash map into the RTT entry in the same order as they are inserted into the hash table. As a result, a SET operation adds a back pointer in a RTT entry using the associated write pointer and increments it to point to the next available position in the RTT entry. Consequently, each entry in the RTT is implemented using a circular buffer. When an entry is evicted from the hash table, its back pointer in the RTT is invalidated. Hence in response to a Free request, the RTT invalidates the hash table entries of the requested hash map. This way, short-lived hash maps mostly stay in the hash table throughout their lifetime without ever being written back to the memory.

Replacement: Replacement is handled by software as discussed above.

foreach: The *foreach* array iterator in PHP iterates over the key-value pairs of a hash map in their order of insertion. The RTT assists in performing this operation. It captures the order of insertion and later updates the memory of the hash map using it in response to a foreach command. Even if an inserted key-value pair is evicted from the hash table and re-inserted later, the RTT can still guarantee the required insertion order invariant.

Ensure coherence Each hash map in the hardware hash table has an equivalent software hash map laid out in the conventional address space. The two are kept coherent by enforcing a writeback policy from the hardware hash map, and by requiring inclusion of the software equivalent at the L2 level. When a hash map is first inserted into the hardware hash table, the accelerator acquires exclusive coherence permission to the entire address range (depending on the size of the hash map this could be several cache lines). If remote coherence requests arrive for any hardware hash map entries, they are forwarded via the RTT to the accelerator, which flushes out any entries corresponding to the address range of the hash map (the same thing happens for L2 evictions to enforce inclusion). In practice, the hash maps we target are small (requiring a handful of cache lines), process private, and exhibit a lot of temporal locality (small hash maps are freed and reallocated from the process-local heap), so there is virtually no coherence activity due to the hash map accelerator. The software hash map stores each key/value pair in a table ordered



(a) Cumulative memory usage with different allocation slabs.

(b) Memory usage of WordPress.

(c) Memory usage of MediaWiki.

Figure 8: Memory usage pattern of the PHP applications.

based on insertion, and also stores a pointer to that table in a hash table for fast lookup. The hardware hash table only writes back to the former table while flushing out entries, and marks a flag in the software hash map to indicate that the hash table of the software hash map is now stale. Subsequent software accesses to the software hash map check this flag and reconstruct the hash table if the flag is set. This is exceedingly rare in practice (triggered only by process migration), so the reconstruction mechanism is necessary only for correctness.

4.3 Heap Manager

Memory allocations and deallocations are ubiquitous in real-world PHP applications. They consume a substantial fraction of the execution time and are spread across many leaf functions. To handle dynamic memory management, the VM typically uses the well-known *slab allocation* technique. In slab allocation, the VM allocates a large chunk of memory and breaks it up into smaller segments of a fixed size according to the slab class's size and stores the pointer to those segments in the associated free list.

Overview We propose a heap manager with its most frequently accessed components in hardware to improve its performance and energy-efficiency. Our hardware heap manager is motivated by the unique memory usage pattern of these PHP applications.

First, a majority of the allocation and deallocation requests retrieve at most 128 bytes, which reflects heavy usage of small memory objects. Figure 8a shows the cumulative distribution of memory usage with different memory allocation slabs.

Second, these applications exhibit strong memory reuse. Such strong memory reuse is due to the following reasons: (a) these applications insert many HTML tags while generating dynamic contents for web pages. HTML tags are the keywords within a web page that define how the browser must format and display the content. Generating and formatting these HTML tags often require retrieving many attribute values from the database, storing them in string objects and later concatenating those values to form the overall formatted tag. Once a HTML tag is produced with all its required attributes, the memory associated with these strings are recycled. (b) These applications typically process large volumes of textual data, URL etc. Such processing commonly parses the content through many string functions and regular expressions with frequent memory allocations and deallocations to hold the string contents. To illustrate this, Figure 8b and Figure 8c show memory usage pattern of two workloads during the course of their execution. We see that memory usage stays flat for the four smallest slabs of $0-32$, $32-64$, $64-96$, and $96-128$ bytes, demonstrating their strong memory reuse for the slabs that dominate the total memory usage.

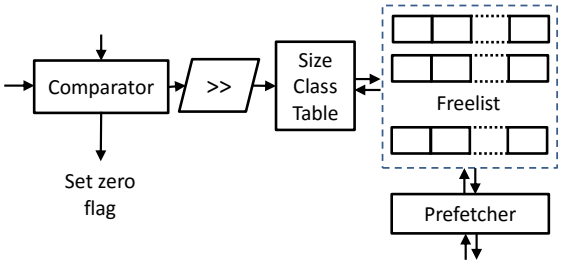


Figure 9: Hardware heap manager.

Proposed design We deploy a heap manager with only its size class table and a few free lists in hardware to capture the heavy memory usage of small objects and their strong memory reuse. Figure 9 shows the high-level block diagram. The comparator limits the maximum size of a memory allocation request that the hardware heap manager can satisfy. The size class table chooses an appropriate free list for an incoming request depending on its request size. Typically, a memory allocation request accesses the size class table and retrieves an available memory block from the chosen hardware free list without any software involvement. The free list for each size class has head and tail pointers to orchestrate allocation and deallocation of memory blocks. The core uses the head pointer for push and pop requests, and the prefetcher pushes to the location of the tail pointer. Upon finding a miss in the hardware free list, control transfers to software to satisfy the memory allocation request. At the start of a program, upon finding a miss for an available memory block in the hardware free list, the runtime retrieves memory blocks from the software heap manager and places it into the appropriate hardware free list. A prefetcher ensures that the free lists stay populated with available memory blocks so that a request for memory allocation can hide the latency of software involvement whenever possible. We use a pointer-based prefetcher to prefetch the next available memory blocks from the software heap manager structure.

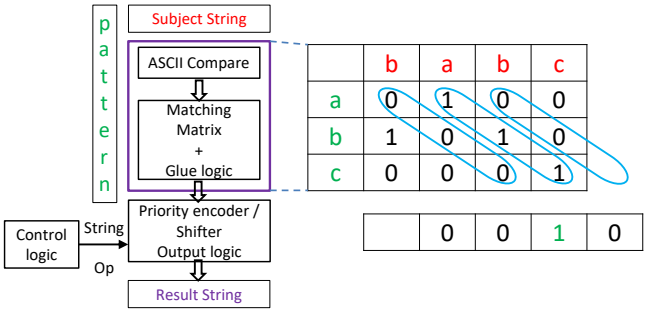


Figure 10: String accelerator.

Design considerations Since a major fraction of the requests attempt to retrieve at most 128 bytes, the hardware heap manager is restricted to serve requests that are at most 128 bytes in size. It uses only 8 memory allocation slabs to perform this, resulting in a very small, power-efficient hardware heap manager. Furthermore, the strong memory reuse means that in the common case it satisfies the requests from the hardware free list, resulting in very infrequent fall-backs to the software routine that handles any complex cases. Concurrent research work on accelerating memory allocation [48] eagerly updates the memory's head pointer and linked list on all malloc and free requests to keep coherency with the memory's data structure. However, due to the high memory reuse of these workloads, we instead lazily update the memory's heap manager data structure only on overflow or during context switches. This avoids the overheads of constantly updating memory to be coherent with the hardware table during periods that the heap manager accelerator is servicing the common case requests, while not causing any correctness errors or memory leaks.

4.4 String Accelerator

In order to reduce the cost of searching, modifying, or otherwise processing text in PHP applications, we propose a generalized string accelerator. Although a significant portion of execution time comes from this category of functions, the execution time is spread out through numerous different string operations. These tasks include string finding, matching, replacing, trimming, comparing, etc. Previous work, such as [68], propose methods for string matching in hardware. However, the hardware proposed processes a single character every cycle, leaving large opportunities for parallelism and higher throughput. Prior designs also do not support the large variety of string operations we wish to accelerate.

Proposed design We note that although string processing is an aggregation of many functions, their overall operation can be broken down into common hardware sub-blocks. By sharing hardware resources, we propose a single string accelerator that can perform several of these operations, as opposed to creating a separate accelerator for every string function.

In addition to matching, our accelerator can substitute characters, perform priority encoding of matches, and determine ranges of character types (useful for detecting lower case, upper case, alphanumeric, etc. characters). We also design our accelerator to process multiple bytes per cycle to exploit concurrency that is not utilized in sequential (single-byte) string accelerators. Figure 10 shows a block diagram of some of the sub-blocks used within our accelerator design.

ASCII compare uses combinational logic to find the presence of pattern characters within the subject string to populate a matching matrix. This operation is done in parallel, and can process as many characters per cycle as is supported by the table width and the width of subject string reads. Operations that require matching of multiple characters use AND gates of diagonal entries within the matching matrix to find the position of consecutive character matches. For example, Figure 10 shows the subject string 'babc' doing a *string_find* for 'abc.' String operations that require index calculation of pattern/character matches use a priority encoder to find the first instance of a valid match. *Output logic* forwards the updated ASCII value to the final output for string functions writing new characters to the result string. *Shifting logic* aligns the subject string to the correct address offset for string manipulation that requires (re-)writing a resulting string to memory. The datapath is organized in such a way that control logic determines the correct combination of sub-blocks enabled based on the incoming string operation. Multi-byte character sets (Unicode) can be handled by grouping the single-byte characters comparisons in the simplified ASCII example shown.

Our accelerator design extracts concurrency by processing many bytes of the subject string in parallel. Our design is not limited to sequential text processing, and therefore can significantly increase string processing throughput. This is because each element in the matching matrix does not require any other element's output for correct calculation. Additionally, due to the commonalities between the string manipulation functions, our generalized accelerator supports many different operations with low overheads.

Design considerations There are several important implementation details that are required for correct operation. First, it is important to support wrap-around in string matching operation, since a match is possible between read text-block boundaries. We support this by buffering previous matching matrix values, and feeding them into the glue-logic sub-block. Second, since a few string functions such as *string_to_upper* and *string_to_lower* are dependent on a range of many ASCII characters, we allow 6 of our matching matrix rows to also support inequality comparisons, as opposed to exclusively equal comparisons. We also allow our pattern length (rows in the matching matrix) and size of subject string processed per cycle (columns in the matching matrix) to be configurable. Entries within the ASCII compare matrix that are unused during a given operation can be clock-gated to further reduce energy consumption. Coherence for writes to destination strings are handled by standard coherence mechanisms, while ordering of memory writes with respect to trailing loads is handled with a hardware interlock similar to a store queue in an out-of-order processor. Since PHP strings are of known length (rather than null-terminated), this coherence and consistency logic is straightforward to implement.

4.5 Regular Expression Accelerator

Traditional regular expression (regexp) processing engines [8] are built around a character-at-a-time sequential processing model that

```
// Replace plain text characters into formatted HTML entities
(?<![\r\n\t ]|\xC2\xA0| )'(...)          sieve regexp

(?<=\A|[([{\-]|&lt;|[\r\n\t ]|\xC2\xA0| )''(...)
                                              shadow regexp
...
// Find double \n\n
\n\n                                          shadow regexp
// Find HTML block-level opening tags
<table|caption|..[\s/>]                       shadow regexp
```

Figure 11: Code snippet from WordPress. All four regexps look for special characters – apostrophe, double quote, newline character and opening angle bracket (highlighted in red).

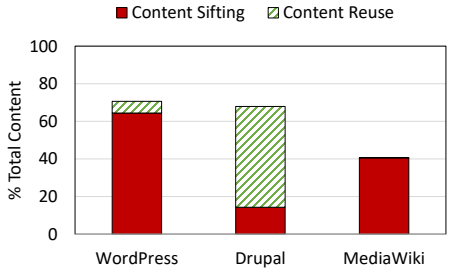


Figure 12: Opportunity with content sifting and content reuse. y-axis shows the percentage of total textual content in the entire application regexps can skip processing using content sifting or content reuse.

introduces high microarchitectural costs. Straightforward parallelization to multi-character inputs leads to exponential growth in the state space [44]. Recent software-based solutions [26, 57, 61] use SIMD parallelism to mitigate that and accelerate regexp processing, but they fail to exploit the regexp-intrinsic characteristics in real-world PHP applications. On the other hand, the high hardware cost associated with parallel regexp accelerators [39, 58, 66] may deter their commercial deployment in the near future. Instead, we exploit the inherent characteristics of the regexps in the PHP applications to skip regexp processing of large volumes of textual data and thus improve the overall execution efficiency. We exploit the following two key insights to achieve that. We believe the two key observations exploited here are generic enough to apply across a wide range of real-world PHP applications.

Content Sifting. The PHP applications process the same unstructured textual content through a series of several regexps during their execution. Furthermore, it is common for most of the regexps among them to seek the presence of some special characters in the source content to convert them into appropriate HTML format or tags. In this work, we classify the following characters {A-Za-z0-9_.,- } as *regular* characters and the remaining ASCII characters as *special* characters. Figure 11 illustrates a set of consecutive regexps from an example function of one of the PHP applications. First, all four regexps in the example function process the same content one after another and second, they all look for special characters – apostrophe, double quote, newline character and opening angle bracket character in the source content[4]. For the sake of simplicity, we assume now that these four regexps do not change the content, and relax this restriction later. We name the first regexp in the set as the *sieve* regexp and the following ones as *shadow* regexps.

Now if the sieve regexp can confirm the presence of no special character in the incoming content, the following shadow regexps can effectively skip scanning the content regardless of the different special characters they look for. Although the sieve regexp must sift (hence the name sieve) the incoming content to observe such a case, this approach can dramatically improve the overall execution efficiency by preventing the shadow regexps from processing the entire source content. In order to exploit this key observation, we use our proposed string accelerator to sift the incoming source content in search of special characters during the course of executing the sieve regexp. This outputs a bit vector indicating segments (of some *granularity*) in the incoming content that *may* have some special characters. We name these bit vectors as *hint* vectors, or HVs. Later, the shadow regexps can solely examine the HV to orchestrate skipping the content. The X86 ISA's *count leading zeros* instruction is used to find the next segment in the HV that requires regexp processing. This way the shadow regexps can skip repeated processing of the *similar* content and thus can avoid character-at-a-time sequential processing. The VM performs function level data flow analysis to determine the dependency relationships between a set of consecutive regexps inside a function and reveals this opportunity for content sifting. However, if the shadow regexps update the initial content, it can change the segment boundaries in the content for which the sieve regexp has generated the HV. This poses a problem because it thwarts content sifting's strategy of processing the source content *once* during the sieve regexp and leverage the generated HV for all of the following regexps to improve execution efficiency.

Whitespace padding in HTML content Fortunately, we can exploit the HTML specification, which allows an arbitrary number of linear white spaces in the response body, to embed the appropriate number of whitespace characters in the updated content to realign the segment boundaries to the existing HV. This ensures the seamless use of the once generated HVs without reprocessing the updated content. For example, when an HTML tag or new characters are inserted into a given text segment, $(SEGMENT\_SIZE - inserted\_text.length())$ whitespace characters will be added to the text segment so that subsequent hint vectors remain aligned within the segment boundaries.

Content Reuse. There exist regexps in the PHP applications that often process *almost* similar content over and over during their execution. For example, they sometimes scan URLs (https://locahost/?author=abc) of two author names with only the name field (last field) in them changing from 'abc' to 'xyz'. Furthermore, in these PHP applications, HTML tags often observe similar attribute values, leading to generating almost same content for the regexps that process them.

As a result, during the course of scanning the second URL, if the regexp can remember observing the almost same content before, it can effectively skip parsing the content up to which it has not changed from last time. Note that with content reuse the regexps can skip processing content even in the presence of special characters

[4]First two regexps look for an apostrophe or double quote before they *look behind* [12] to find a match for the first segment inside the parenthesis.

PC	ASID	Content	Size	Next Valid FSM State
...		...	...	...
PC X	ASID X	https://localhost/?author=	26	State Y
...		...	...	...
...		...	...	...

Figure 13: Hardware content reuse table

which content sifting technique can not. We use a reuse table to capture this opportunity. The reuse table is indexed by a regexp PC value, and address space identifier (ASID). Each entry in the table has three fields – the first stores the matching content seen last time when the regexp was executed, the second captures the content size, and the third captures the state in the FSM table that the regexp can advance to if the incoming content finds a match with the first field. When accessing the content reuse table, there are three possible scenarios. First, there could be a PC, ASID, and content match. In this case, the software can automatically `jumpto` the FSM state located in the hardware table. Second, if there is a PC miss, ASID miss, or the first byte of content doesn't match, we consider that an invalid-miss. In this case, the new content is placed in the table (or overwriting the previous entry if a PC + ASID hit), and the size and FSM fields in the table are cleared. The software handler then traverses the FSM normally. In the last case, if there is a PC + ASID hit, and a non-zero matching size differs from the size listed in the table (or size is currently cleared), then the content and size fields are updated, and the software handler will traverse the FSM to determine the new FSM state to `jumpto` in future hits of that size. Once this state is determined, the software handler writes the FSM state in the table for future accesses. Figure 13 shows the appearance of the content reuse table from the example listed above. The 'Content' field in the reuse table is limited to a maximum of 32 bytes for efficiency reasons. Figure 12 shows the percentage of textual content that regexps can avoid processing using either content sifting or content reuse.

4.6 ISA Extensions

In order to invoke our tightly-coupled accelerators, we add ISA extensions. We added `hashtableget` and `hashtableset` instructions to invoke GET and SET requests to the hash table. The zero flag is raised upon a miss of a GET, or hash table overflow of a SET, in which case the code branches to the software handler fallback. The state of the hash table is hardware coherent, so no cleanup operations are required during context switches.

We also added `hmmalloc` and `hmfree` instructions in order to invoke malloc and free requests through our hardware. Similar to the hash table, the zero flag is used to conditionally branch to the software handler fallback. For `hmmalloc`, the flag is set if the hardware's requested size class is empty and requires involvement of the software handler to gain the next free block. For `hmfree`, the flag is set if adding the new block overflows the maximum number of entries for the given size class. In this case, the software handler updates the content of the second-to-last block (which becomes the last block for the given size class post-eviction) in memory to point to the evicted block (which can be done using a single `str` instruction). At context switches, the hardware heap manager must flush its entries to the memory's heap manager data structure. We do this by adding the instruction `hmflush`. `hmflush` is resumable in order to guarantee forward progress in the case that multiple page faults occur during the flush.

We create the `stringop[op]` instruction to invoke our string accelerator. Six bits are used as an extra opcode to specify which function (eg. trim, find, translate, etc. – denoted `[op]`) the accelerator should perform, in addition to the source and destination registers. For most of the string functions, the accelerator has a straightforward invocation based on the source register passed. For complex string functions[5] we create the `strreadconfig` instruction, which populates the string accelerator's matching matrix rows if it is not already configured. Additionally, the string accelerator should return to its previous configuration after a context switch. For this reason, we create a `strwriteconfig` instruction to store the string accelerator's current configuration. `strreadconfig` is used to reinitialize the string accelerator after a context switch.

To perform any regular expression matching, we replace the Perl Compatible Regular Expression (PCRE) library calls with our own APIs – We separate regular expression matching into *regexp_sieve* and *regexp_shadow*. *regexp_sieve* is called on the first regular expression for a set of data. It does the traditional regexp matching, in addition to populating the HV (stored in memory) by invoking our string accelerator. Future calls are made with *regexp_shadow* in order to optimize regexp searching in light of the populated HV. In order to make use of the content reuse table, we create a `regexlookup` instruction. It searches the table for a PC, ASID, and content match. In order to update the FSM state value, we create a `regexset` instruction, which the software handler invokes after a duplicate substring is found. The details of how and when the hardware is updated are explained in Section 4.5.

5 EXPERIMENTAL FRAMEWORK AND RESULTS

Our evaluation is divided into two subsections. Section 5.2 presents the performance and energy benefits obtained with deploying our specialized hardware. Section 5.3 illustrates the detailed breakdown of performance benefits.

5.1 Methodology

We study three popular real-world PHP web applications – WordPress [15], Drupal [2], and MediaWiki [7] from the *oss-performance* suite [5]. WordPress is reportedly the easiest and most popular blogging platform in use today supporting more than 60 million websites [16]. Drupal powers at least 2.2% of all web sites worldwide, including some of the busiest sites on the web, ranging from discussion forums and personal blogs to corporate sites [3]. MediaWiki

[5]We denote complex string functions to require multiple row initializations of the string accelerator's matching matrix that are *not* determined by source operands (eg. *string_to_upper* and creating the HVs for the regexp accelerator). These configurations can be large and may not be practical or feasible to pass as a source operand, and therefore require the matching matrix to be loaded from memory using a separate instruction, `strreadconfig`. `strreadconfig` is involved at the start of the program and after context switches.

serves as the platform for Wikipedia and many other wikis, including some of the largest and most popular ones.

In order to understand the characteristics of the PHP applications and guide the subsequent architectural simulations, we conduct a system-level performance analysis using the linux *perf* command on an Intel Xeon processor running 64-bit Ubuntu 12.04. We used the load generator available with the oss-performance suite to generate client requests. The load generator emulates load from a large pool of client clusters, closely imitating the environment of commercial servers. It generates 300 warmup requests, then as many requests as possible in next one minute. These experiments use the nginx web server, configured to use the HHVM (with all its optimizations turned on) via FastCGI.

We use gem5 [4] for microarchitectural characterization of the PHP applications. We evaluate our proposal using an in-house trace-driven simulator. Our simulated server processor is configured similarly to the Intel Xeon-based (4 wide out-of-order) server. Among the proposed accelerators, we implement the string accelerator in Verilog and synthesize using TMSC 45nm standard cell library operating at 2GHz. At 2GHz, the string accelerator requires a maximum of 3 cycles to process up to 64 character blocks. We use CACTI 6.5+ [52] to estimate the access latency, energy and area of the remaining proposed accelerators. The combined area overhead of the specialized hardware accelerators is 0.22 mm^2. An Intel Nehalem core (precursor to the Xeon core with same fetch and issue width) measures 24.7 mm^2 including private L1 and L2 caches. If integrated into a Nehalem or Xeon-based core, our proposed specialized hardware is merely 0.89% of the core area. The hardware hash table has 512 entries in our design. In response to a hash table access request, only 4 consecutive entries are accessed (and in parallel) with the computed hash. This restricts the hash table access latency to a constant 1 cycle after performing the initial hash computation. If no match is found, control falls back to the software to perform the regular hash map access in memory. The hardware heap manager has 8 size classes, each having 32 entries in its corresponding free list. 32 entries provides enough flexibility to the prefetcher in hiding the prefetch latency. The heap manager requires 1 cycle to satisfy a request from a hardware free list. The content reuse table has 32 entries. We use McPAT [52] to collect core power and energy.

5.2 Performance and Energy Improvement

Figure 14 shows the improvement in execution from our specialized architecture. Applying the prior research proposals as discussed in Section 3 brings down the average execution time to about 88.15% of the time obtained with unmodified HHVM, whereas our specialized core brings down the execution time further to 70.22%. Behind the 11.85% improvement in execution time from mitigating abstraction overheads, reducing the overheads of the frequent reference counting mechanism contributes the most (on average 4.42%). Among the three applications, Drupal shows the least opportunity (Figure 5), and naturally benefits less from the our proposed accelerators. Note that the performance benefit from our accelerators will be even more prominent (19.79% on average) as future server processors incorporate the prior optimizations.

We consider the reduction of dynamic CPU instructions (after using our accelerators) as a simple proxy for estimating the CPU

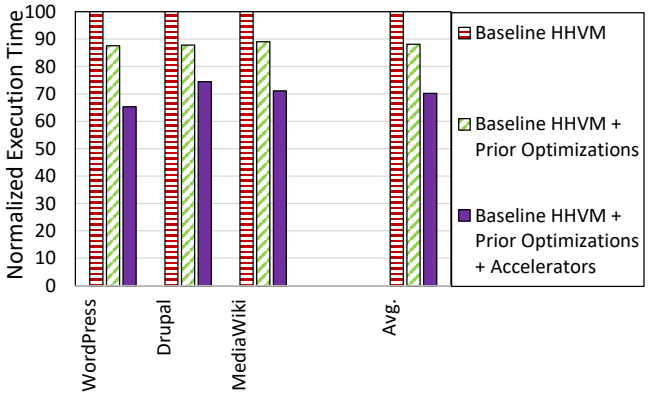


Figure 14: Improvement in execution time with applying prior optimizations and our specialized hardware. Execution time is normalized to unmodified HHVM.

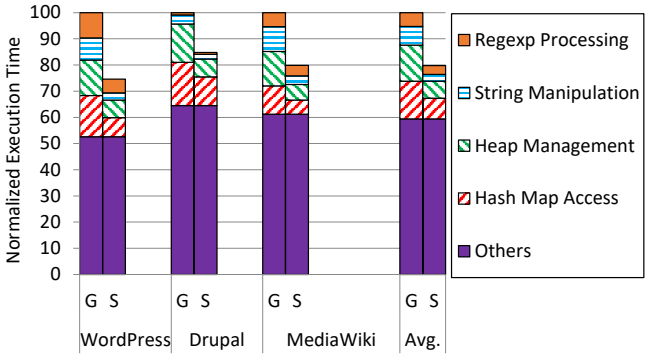


Figure 15: Breakdown of execution time. *G* refers to the execution time with applying optimizations from prior works as discussed in Section 3, *S* refers to the execution time with all our proposed accelerators. Execution time is normalized to *G*.

energy savings. We calculate total energy consumption of our accelerators by using simulation counters of the cycles offloaded to each accelerator, in combination with the accelerator energy numbers provided by CACTI and Verilog synthesis. On average, our specialized hardware delivers about 21.01% energy savings on top of the energy savings obtained from deploying the prior proposals. Among the three PHP applications, the energy consumption for WordPress, Drupal, and MediaWiki drops by 26.06%, 16.75%, and 19.81% respectively. Most of the energy savings are attributed to the reduction in the execution of instructions. Note that the specialized architecture may also gain additional energy benefit from fewer data cache accesses, since it now traverses hash table and heap manager data structures in hardware.

Memory allocation requests (*malloc* and *free*) require on average 69 and 37 x86 micro-ops, respectively, in software to execute (assuming cache hits). Hash map walks in software require on average 90.66 x86 micro-ops. Our specialized architecture saves energy from accelerating the frequently executed paths of these operations in hardware. The string accelerator reduces instruction count by exploiting concurrency, whereas our regular expression accelerator

processes less content and thus performs less work by leveraging the two content filtering techniques.

5.3 Breakdown of Execution Time

Figure 15 shows the breakdown of execution time for the PHP applications. The different entries in the legend denote the time consumed in corresponding activities. We observe that in general, the hardware heap manager delivers the biggest benefit (7.29% on average) among all four accelerators. First, it is spread across many leaf functions, and second, the hardware traversal of the size class table and the free lists not only reduces contention in memory system, but also sometimes helps in avoiding cache misses otherwise occurring with accessing those structures in memory. The hardware hash table also delivers significant benefit of 6.45% on average across applications. The string accelerator and the regular expression accelerator deliver 4.51% and 1.96% performance benefit respectively. WordPress observes considerable benefit from the regexp accelerator, whereas MediaWiki obtains modest benefit. Although Drupal demonstrates significant opportunity in Figure 12 from the two regexp acceleration techniques, it does not translate into performance gain, as it does not spend much time either in regexp processing or in string functions.

6 RELATED WORK

Server core design. In recent years, numerous research efforts have been devoted to optimizing warehouse-scale(WSC) and big data workloads [24, 36, 47, 53, 55, 56, 69] developed in C++-like compiled languages. [47] has demonstrated in-depth microarchitectural characterization of WSC workloads and provided several possible directions for architects to accelerate them. Further, there have been multiple recent efforts to address instruction cache bottlenecks [45, 50] in datacenter workloads.

However, the software community is increasingly leaning towards scripting languages due to their high programmer productivity [27]. As these live datacenters host millions of web applications developed primarily in scripting languages (typically PHP and Javascript), even small improvements in performance or utilization will translate into immense cost savings. Prior work [73] concentrated on server-side Javascript applications. However, PHP is most commonly used [70, 72], representing 82.3% [13] of all web applications. In this context, our work is the first to present a comprehensive analysis of the microarchitectural bottlenecks of real-world server-side PHP applications.

Scripting language optimizations. There is a large body of research in optimizing the performance of scripting languages from the software side. Prior works [21, 31–33, 38, 40, 42, 43, 49, 62, 71] attempt to mitigate abstraction overheads (see Section 3) associated with these languages.

In recent years, research interest in developing new architectural support for server-side and client-side scripting applications has gone up significantly. In the client side, Javascript is used predominantly whereas PHP is the language of choice for server-side web development [13]. [22, 59] propose microarchitectural changes to avoid runtime checks in jitted execution of Javascript programs. [27] demonstrates the potential of asymmetric multiprocessors in mitigating the abstraction overheads even further. Front-end bottlenecks, more specifically instruction cache misses are observed to be a major source of performance bottlenecks in real-world Javascript applications. Prior works propose instruction prefetching [28], pre-execution techniques [29] and modifying cache insertion policy [73] to mitigate this. However, surprisingly the real-world PHP applications in our work do not observe instruction cache misses causing a performance bottleneck despite having the presence of hundreds of leaf functions in their distribution. There are very few works in exploring architectural support for PHP applications and they mainly experiment with micro-benchmarks[20]. [20] improves the energy efficiency of SPECWeb2005 workloads by aligning the execution of similar requests together. Instead our work focuses on real-world PHP web applications and studies its implications on general-purpose server cores that host those.

Specialization alternatives. A hash table that supports only GET operation has been deployed in hardware before for memcached workloads [55]. Furthermore, [30] deployed the entire memcached algorithm (supporting both GET and SET) in an FPGA platform. However in addition to the two operations, the PHP applications require support for other important PHP operations (for example, foreach) on hash maps. Without these features, such a hash table in PHP environment will be highly inefficient and not safe to operate on. Our hash table design supports these features, ensures CMP coherence of such table in a multicore server platform and obtains efficiency by exploiting the inherent characteristics (for example, short-lived hash maps) of PHP applications.

Dynamic heap management in its entirety is non-trivial to implement in hardware [47]. Our heap manager on the other hand relies on hardware only for the common case, allowing software to provide full-featured dynamic heap management. It achieves that goal by capturing the application-intrinsic characteristics: the strong memory reuse of the principal data structures observed in the PHP applications.

There is a large body of research in accelerating regular expression (regexp) processing [26, 39, 57, 58, 61, 66]. In recent years, several works [39, 58, 66] attempt to parallelize processing by running the regexp separately on separate substrings of the input or by splitting the regexp into multiple sub-regexps and later combining the results appropriately. Our proposed regexp acceleration technique that exploits regexp-intrinsic characteristics of the PHP applications can benefit further by adopting these prior proposals.

7 CONCLUSION

Server-side PHP applications occupy a huge footprint in the world of web development. By performing an in-depth analysis, we found potential for specialized hardware to accelerate these PHP applications in a future server SoC. We believe the behavioral characteristics that we found for three popular PHP applications in this work exist across a wide-range of other PHP applications such as Laravel [6], Symfony [11], Yii [17], Phalcon [9] etc. and hence will all gain execution efficiency when using our proposed accelerators.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their insightful comments and feedback on the paper. This work was supported in part by NSF grants CCF-1318298 and CCF-1615014. Mikko Lipasti has a financial interest in Thalchemy Corp.

REFERENCES

[1] The Computer Language Benchmarks Game, http://shootout.alioth.debian.org/.

[2] Drupal. https://www.drupal.org/.

[3] Drupal wikipedia. https://en.wikipedia.org/wiki/Drupal.

[4] The Gem5 simulator: http://gem5.org/.

[5] https://github.com/hhvm/oss-performance.

[6] Laravel. https://laravel.com/.

[7] MediaWiki. https://www.mediawiki.org/wiki/MediaWiki.

[8] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/.

[9] Phalcon. https://phalconphp.com/en/.

[10] Standard Performance Evaluation Corporation. SPECweb2005. https://www.spec.org/web2005/.

[11] Symfony. https://symfony.com/.

[12] Tutorial on Regular Expressions. http://www.regular-expressions.info/lookaround.html.

[13] Usage of server-side programming languages for websites. https://w3techs.com/technologies/overview/programming_language/all.

[14] V8 JavaScript Engine. https://developers.google.com/v8/.

[15] WordPress. https://wordpress.com/.

[16] WordPress wikipedia. https://en.wikipedia.org/wiki/WordPress.

[17] Yii. http://www.yiiframework.com/.

[18] Zend PHP, http://php.net/.

[19] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The Hiphop Virtual Machine. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '14)*. New York, NY, USA, 777–790. https://doi.org/10.1145/2660193.2660199

[20] Sandeep R. Agrawal, Valentin Pistol, Jun Pang, John Tran, David Tarjan, and Alvin R. Lebeck. 2014. Rhythm: Harnessing Data Parallel Hardware for Server Workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. New York, NY, USA, 19–34. https://doi.org/10.1145/2541940.2541956

[21] Wonsun Ahn, Jiho Choi, Thomas Shull, María J. Garzarán, and Josep Torrellas. 2014. Improving JavaScript Performance by Deconstructing the Type System. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. New York, NY, USA, 496–507. https://doi.org/10.1145/2594291.2594332

[22] Owen Anderson, Emily Fortuna, Luis Ceze, and Susan Eggers. 2011. Checked Load: Architectural Support for JavaScript Type-checking on Mobile Processors. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*. Washington, DC, USA, 419–430.

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[24] Islam Atta, Pinar Tozun, Anastasia Ailamaki, and Andreas Moshovos. 2012. SLICC: Self-Assembly of Instruction Cache Collectives for OLTP Workloads. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 188–198. https://doi.org/10.1109/MICRO.2012.26

[25] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and Realities: The Performance Impact of Garbage Collection. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '04/Performance '04)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1005686.1005693

[26] Robert D. Cameron and Dan Lin. 2009. Architectural Support for SWAR Text Processing with Parallel Bit Streams: The Inductive Doubling Principle. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 337–348. https://doi.org/10.1145/1508244.1508283

[27] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. 2012. The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE Computer Society, Washington, DC, USA, 225–236. http://dl.acm.org/citation.cfm?id=2337159.2337185

[28] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2014. EFetch: Optimizing Instruction Fetch for Event-driven Webapplications. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 75–86. https://doi.org/10.1145/2628071.2628103

[29] Gaurav Chadha, Scott Mahlke, and Satish Narayanasamy. 2015. Accelerating Asynchronous Programs Through Event Sneak Peek. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 642–654. https://doi.org/10.1145/2749469.2750373

[30] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. 2013. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/2435264.2435306

[31] Craig Chambers and David Ungar. 1989. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation (PLDI '89)*. ACM, New York, NY, USA, 146–160. https://doi.org/10.1145/73141.74831

[32] Craig Chambers, David Ungar, and Elgin Lee. 1989. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications (OOPSLA '89)*. New York, NY, USA, 49–70. https://doi.org/10.1145/74877.74884

[33] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '84)*. ACM, New York, NY, USA, 297–302. https://doi.org/10.1145/800017.800542

[34] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. 2015. Fast Support for Unstructured Data Processing: The Unified Automata Processor. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 533–545. https://doi.org/10.1145/2830772.2830809

[35] M. Umar Farooq, Khubaib, and Lizy K. John. 2013. Store-Load-Branch (SLB) Predictor: A Compiler Assisted Branch Prediction for Data Dependent Branches. In *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA) (HPCA '13)*. IEEE Computer Society, Washington, DC, USA, 59–70. https://doi.org/10.1109/HPCA.2013.6522307

[36] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[37] Tais B Ferreira, Rivalino Matias, Autran Macedo, and Lucio B Araujo. 2011. An experimental study on memory allocators in multicore and multithreaded applications. In *2011 12th international conference on parallel and distributed computing, applications and technologies (pdcat)*. IEEE, 92–98.

[38] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based Just-in-time Type Specialization for Dynamic Languages. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. New York, NY, USA, 465–478. https://doi.org/10.1145/1542476.1542528

[39] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. 2016. HARE: Hardware Accelerator for Regular Expressions. In *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE Computer Society, Washington, DC, USA.

[40] Dibakar Gope and Mikko H. Lipasti. 2016. Hash Map Inlining. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT '16)*. ACM, New York, NY, USA, 235–246. https://doi.org/10.1145/2967938.2967949

[41] Nathan Goulding-Hotta, Jack Sampson, Ganesh Venkatesh, Saturnino Garcia, Joe Auricchio, Po-Chao Huang, Manish Arora, Siddhartha Nath, Vikram Bhatt, Jonathan Babb, Steven Swanson, and Michael Taylor. 2011. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future. *IEEE Micro* 31, 2 (March 2011), 86–95. https://doi.org/10.1109/MM.2011.18

[42] Brian Hackett and Shu-yu Guo. 2012. Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, New York, NY, USA, 239–250. https://doi.org/10.1145/2254064.2254094

[43] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '91)*. Springer-Verlag, London, UK, UK, 21–38. http://dl.acm.org/citation.cfm?id=646149.679193

[44] Nan Hua, Haoyu Song, and TV Lakshman. 2009. Variable-stride multi-pattern matching for scalable deep packet inspection. In *INFOCOM 2009, IEEE*. IEEE, 415–423.

[45] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely, and Joel Emer. 2015. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 343–353.

[46] José A. Joao, Onur Mutlu, and Yale N. Patt. 2009. Flexible Reference-counting-based Hardware Acceleration for Garbage Collection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA '09)*. ACM,

New York, NY, USA, 418–428. https://doi.org/10.1145/1555754.1555806

[47] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 158–169. https://doi.org/10.1145/2749469.2750392

[48] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 33–45. https://doi.org/10.1145/3037697.3037736

[49] Madhukar N. Kedlaya, Jared Roesch, Behnam Robatmili, Mehrdad Reshadi, and Ben Hardekopf. 2013. Improved Type Specialization for Dynamic Scripting Languages. In *Proceedings of the 9th Symposium on Dynamic Languages (DLS '13)*. New York, NY, USA, 37–48. https://doi.org/10.1145/2508168.2508177

[50] Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. 2013. RDIP: Return-address-stack Directed Instruction Prefetching. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 260–271. https://doi.org/10.1145/2540708.2540731

[51] Sangho Lee, Teresa Johnson, and Easwaran Raman. 2014. Feedback Directed Optimization of TCMalloc. In *Proceedings of the Workshop on Memory Systems Performance and Correctness (MSPC '14)*. ACM, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2618128.2618131

[52] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 42)*. ACM, New York, NY, USA, 469–480. https://doi.org/10.1145/1669112.1669172

[53] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 476–488. https://doi.org/10.1145/2749469.2750416

[54] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. 2010. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–12.

[55] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 36–47. https://doi.org/10.1145/2485922.2485926

[56] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. 2008. Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA '08)*. IEEE Computer Society, Washington, DC, USA, 315–326. https://doi.org/10.1109/ISCA.2008.37

[57] Dan Lin, Nigel Medforth, Kenneth S. Herdy, Arrvindh Shriraman, and Robert D. Cameron. 2012. Parabix: Boosting the efficiency of text processing on commodity processors. In *18th IEEE International Symposium on High Performance Computer Architecture, HPCA 2012, New Orleans, LA, USA, 25-29 February, 2012*. 373–384. https://doi.org/10.1109/HPCA.2012.6169041

[58] Jan Van Lunteren, Christoph Hagleitner, Timothy Heil, Giora Biran, Uzi Shvadron, and Kubilay Atasu. 2012. Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 461–472. https://doi.org/10.1109/MICRO.2012.49

[59] Mojtaba Mehrara and Scott Mahlke. 2011. Dynamically Accelerating Client-side Web Applications Through Decoupled Execution. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. Washington, DC, USA, 74–84.

[60] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. 2015. Branch Prediction and the Performance of Interpreters: Don'T Trust Folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*. IEEE Computer Society, Washington, DC, USA, 103–114. http://dl.acm.org/citation.cfm?id=2738600.2738614

[61] Valentina Salapura, Tejas Karkhanis, Priya Nagpurkar, and Jose Moreira. 2012. Accelerating Business Analytics Applications. In *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/HPCA.2012.6169044

[62] Henrique Nazare Santos, Pericles Alves, Igor Costa, and Fernando Magno Quintao Pereira. 2013. Just-in-time Value Specialization. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13)*. IEEE Computer Society, Washington, DC, USA, 1–11. https://doi.org/10.1109/CGO.2013.6495006

[63] André Seznec. 2011. A New Case for the TAGE Branch Predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*. ACM, New York, NY, USA, 117–127. https://doi.org/10.1145/2155620.2155635

[64] Y Sophia Shao, Sam Xi, Viji Srinivasan, Gu-Yeon Wei, and David Brooks. Toward cache-friendly hardware accelerators.

[65] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. 2009. Accelerating Critical Section Execution with Asymmetric Multi-core Architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*. ACM, New York, NY, USA, 253–264. https://doi.org/10.1145/1508244.1508274

[66] Prateek Tandon, Faissal M Sleiman, Michael J Cafarella, and Thomas F Wenisch. 2016. Hawk: Hardware support for unstructured log processing. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 469–480.

[67] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. 2010. Conservation Cores: Reducing the Energy of Mature Computations. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, USA, 205–218. https://doi.org/10.1145/1736020.1736044

[68] Shin'ichi Wakabayashi, Shinobu Nagayama, Yosuke Kawanaka, and Sadatoshi Mikami. 2009. Hardware Accelerators for Regular Expression Matching and Approximate String Matching. In *Proceedings: APSIPA ASC 2009: Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference*. Asia-Pacific Signal and Information Processing Association, 2009 Annual Summit and Conference, International Organizing Committee.

[69] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, and others. 2014. Bigdatabench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 488–499.

[70] Samuel R. Warner and James Steven Worley. 2008. SPECweb2005 in the real world: Using IIS and PHP. In *Proceedings of SPEC Benchmark Workshop*.

[71] Kevin Williams, Jason McCandless, and David Gregg. 2010. Dynamic Interpretation for Dynamic Scripting Languages. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. New York, NY, USA, 278–287. https://doi.org/10.1145/1772954.1772993

[72] Haiping Zhao, Iain Proctor, Minghui Yang, Xin Qi, Mark Williams, Qi Gao, Guilherme Ottoni, Andrew Paroski, Scott MacVicar, Jason Evans, and Stephen Tu. 2012. The HipHop Compiler for PHP. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. New York, NY, USA, 575–586. https://doi.org/10.1145/2384616.2384658

[73] Yuhao Zhu, Daniel Richins, Matthew Halpern, and Vijay Janapa Reddi. 2015. Microarchitectural Implications of Event-driven Server-side Web Applications. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 762–774. https://doi.org/10.1145/2830772.2830792