Architectural Support for Scripting Languages

By

Dibakar Gope

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2017

Date of final oral examination: 6/7/2017

The dissertation is approved by the following members of the Final Oral Committee:

Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Gurindar S. Sohi, Professor, Computer Sciences
Parameswaran Ramanathan, Professor, Electrical and Computer Engineering
Jing Li, Assistant Professor, Electrical and Computer Engineering
Aws Albarghouthi, Assistant Professor, Computer Sciences

This thesis is dedicated to my parents, Monoranjan Gope and Sati Gope.

ACKNOWLEDGMENTS

First and foremost, I would like to thank my parents, Sri Monoranjan Gope, and Smt. Sati Gope for their unwavering support and encouragement throughout my doctoral studies which I believe to be the single most important contribution towards achieving my goal of receiving a Ph.D.

Second, I would like to express my deepest gratitude to my advisor Prof. Mikko Lipasti for his mentorship and continuous support throughout the course of my graduate studies. I am extremely grateful to him for guiding me with such dedication and consideration and never failing to pay attention to any details of my work. His insights, encouragement, and overall optimism have been instrumental in organizing my otherwise vague ideas into some meaningful contributions in this thesis. This thesis would never have been accomplished without his technical and editorial advice. I find myself fortunate to have met and had the opportunity to work with such an all-around nice person in addition to being a great professor. I do not know what the future awaits, but thanks to Prof. Mikko, I feel more prepared than ever before to pursue a career in areas where I may not have a strong expertise to begin with.

I would like to extend my appreciation to the members of my thesis committee, including Prof. Guri Sohi, Prof. Parameswaran Ramanathan, Prof. Jing Li, and Prof. Aws Albarghouthi, whose wealth of knowledge and insights have immensely guided me during the transition from preliminary Ph.D. proposal to completed Ph.D. thesis. I benefited greatly from the other excellent faculty at University of Wisconsin-Madison as well. I am particularly grateful to Prof. Mark Hill, Prof. David Wood, and Prof. Karu Sankaralingam

for their excellent computer architecture courses that benefited me greatly to understand and appreciate the intricacies involved in designing complex systems such as processors in the early phases of my graduate studies. I also thank the professors from compilers, systems, and machine learning groups for all the things that I have learned from their respective courses.

Finally, I would like to thank my graduate school friends, including Mitch Hayenga, Arslan Zulfiqar, Vignyan Reddy, David Palframan, Sean Franey, Andrew Nere, Zhong Zheng, Rohit Shukla, Gokul Ravi, Sooraj Puthoor, David Schlais, Joel Hestness, Tony Nowatzki, Newsha Ardalani, Hongil Yoon, Nilay Vaish, Muhammad Shoaib bin Altaf, Swapnil Haria, Jason Lowe-Power, Marc Orr, Somayeh Sardashti, Vinay Gangadhar, Amin Farmahini, Hamid Reza Ghasemi, Hao Wang, Mohammad Alian, Felix Loh, Kyle Daruwalla, and Carly Schulz for their support and camaraderie. I benefited greatly from numerous discussions with them in various occasions. Special thanks go to David Schlais for his help with hardware prototyping one of the hardware accelerators proposed in this thesis.

I also thank the staff in the Electrical and Computer Engineering department for making my academic life at University of Wisconsin-Madison a great experience.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Scripting languages like PHP and Javascript are widely used to implement application logic for dynamically-generated web pages. Their popularity is due in large part to their flexible syntax and dynamic type system, which enable rapid turnaround time for prototyping, releasing, and updating web site features and capabilities. Among all scripting languages used for server-side web development, PHP is the most commonly used. The most common complex data structure in these languages is the hash map, which is used to store key-value pairs. In many cases, hash maps with a fixed set of keys are used in lieu of explicitly defined classes or structures, as would be common in compiled languages like Java or C++. Unfortunately, the runtime overhead of key lookup and value retrieval is quite high, especially relative to the direct offsets that compiled languages can use to access class members. Furthermore, key lookup and value retrieval incur high microarchitectural costs as well, since the paths they execute contain unpredictable branches and many cache accesses, leading to substantially higher numbers of branch mispredicts and cache misses per access to the hashmap. This thesis quantifies these overheads, describes a compiler algorithm that discovers common use cases for hash maps and inlines them so that keys are accessed with direct offsets, and reports measured performance benefits on real hardware. A prototype implementation in the HipHop VM infrastructure shows promising performance benefits for a broad array of hash map-intensive server-side PHP applications, up to 37.6% and averaging 18.81%, while improving SPECWeb throughput by 7.71% (banking) and 11.71% (e-commerce).

Just-in-time compilation, as implemented in Facebook's state-of-the-art HipHopVM,

helps mitigate the poor performance of PHP with the help of the Hash map Inlining algorithm (introduced above) and other state-of-the-art code optimization techniques, but substantial overheads remain, especially for realistic, large-scale, content-rich PHP applications that spend most of their time in rendering web pages. This dissertation 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, content-rich PHP workloads on a state-of-the-art software and hardware platform.

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 as they are used for an increasing share of application software. Among all scripting languages used for server-side web development to access databases and other middleware, PHP is the most commonly used [115, 105], representing over 80% [99] 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. Because of their dynamic features, PHP like scripting languages are typically interpreted by virtual machine runtimes. Usually these interpreted implementations are one or two orders of magnitude slower compared to their corresponding implementations in compiled languages [93]. This has spurred a number of research to improve the performance of PHP scripts through just-in-time(JIT) compilation [7, 30, 38, 77, 115, 1]. 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.

To this end, this dissertation first proposes a compiler optimization technique called Hash Map Inlining to eliminate the overheads associated with populating and accessing

key-value pairs stored in hash maps, the most commonly occurring data structure in server-side PHP applications. JIT compilation with the help of Hash map Inlining and other state-of-the-art code optimization techniques helps to mitigate the poor performance of PHP, but substantial overheads remain, especially for realistic, large-scale, content-rich PHP applications that spend most of their time in rendering web pages. This dissertation then analyzes such content-rich applications in detail, identifies four fine-grained PHP activities - hash table accesses, heap management, string manipulation, and regular expression handling in them as common building blocks across their many leaf functions that constitute a significant fraction of total server cycles, and proposes novel, inexpensive hardware accelerators for these activities. The contributions made in this dissertation show the potential to improve the efficiency of web servers and thus in turn to directly influence the throughput of data centers.

1.1 Hash Map Inlining

The most common complex data structure in PHP like scripting languages is the hash map, which is used to store key-value pairs. Typically hash maps with a fixed set of keys are used instead of explicitly defined classes or structures, as would be common in compiled languages like Java or C++. Unfortunately, the runtime overhead of key lookup and value retrieval is quite high, especially relative to the direct offsets that compiled languages can use to access class members. Figure 1.1 demonstrates the microarchitectural behavior of a microbenchmark that repeatedly updates and accesses a configurable number of key-value pairs stored in a hash map. The bottom three lines illustrate the behavior with accessing

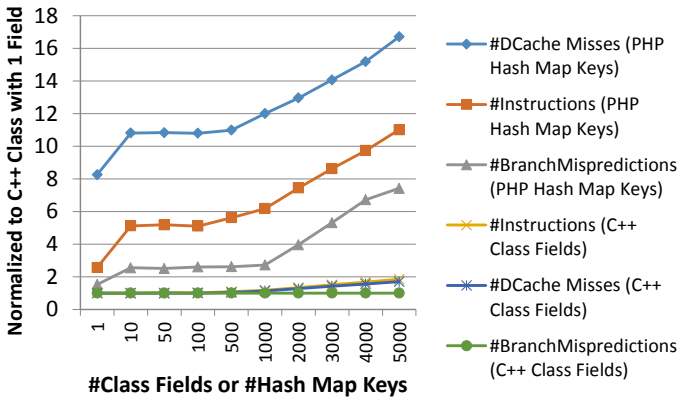


Figure 1.1: Comparison of branch mispredictions, data cache misses and instructions count between accessing a hash map and a class object (details in evaluation chapter).

class objects with equivalent number of fields. Clearly key lookup and value retrieval from a hash map incur significantly higher number branch mispredictions, cache misses and instructions.

Recently the HipHop Virtual Machine(HHVM) [1] PHP JIT compiler from Facebook has shown tremendous gains to close the performance gap with compiled (statically-typed) languages. However, as observed in this dissertation, in a set of real-world, server-side PHP applications that power many e-commerce platforms, online news forums, banking servers, etc., hash map processing constitutes a significant fraction of the overall execution time and failure to optimize accesses to those hash maps by HHVM causes a substantial performance bottleneck.

In this dissertation, we propose *Hash Map Inlining (HMI)* to minimize the overheads associated with populating and accessing key-value pairs stored in hash maps. HMI dynamically converts a hash map into a vector-like data structure that is accessed with fixed, linear offsets for each key value, and specializes the code at each access site to use fixed offsets from the HMI base address to update and/or retrieve values corresponding to

each key. Our implementation of HMI is inspired by *inline caching*, an existing compiler optimization technique for streamlining access to dynamically-typed objects. With our prototype HMI implementation, the vast majority of the overhead of hash map accesses in the microbenchmark can be elided, leading to gains of up to $40-45\%$ (with a hash map of 10 or 50 key-value pairs).

However, we observe that our initial HMI implementation delivers only marginal gains when applied to real-world, server-side PHP applications that utilize hash maps to retrieve information from a back-end database management system (DBMS). The effectiveness of HMI in these applications is limited for two reasons. First, the hash maps are populated inside SQL runtime libraries written in C code, which are not visible to the HHVM optimizer, effectively preventing HMI from triggering inlining and code specialization. Second, our initial version of HMI can only specialize code for accesses where the hash map keys are specified as literal values at the access site (e.g. myhashmap[“literalkey”]), whereas these applications commonly specify the keys as variables (e.g. myhashmap[$myvariablekey]). In theory, flow analysis and constant propagation may reveal that some of the latter cases are in fact constants (literals), but this is not the case for the applications we examine. Instead, we find that the variables at each access site sequences through a number of different, though predictable, key names at run time.

In order to address these shortcomings, we extend our initial HMI implementation to inline accesses to keys with variable names at an access site, whenever we can guarantee that the variable at the access site will sequence through a number of different, though predictable, fixed key names at run time. This condition is trivially satisfied for the SQL runtime library functions we targeted, since the ordered set of keys is determined by the

database schema, which is fixed at the time a SQL query is evaluated. This results in automatic conversion of hash maps into inlined form for these realistic, server-side PHP applications, such that subsequent accesses within the PHP code can be efficiently specialized to take full advantage of the inlined hash map structure. The prototype implementation of HMI in HHVM shows performance benefits for a broad array of hash map-intensive PHP scripts, up to 37.6% and averaging 18.81%, and improves SPECWeb throughput by 7.71% (banking) and 11.71% (e-commerce).

1.2 Architectural Support for Server-Side PHP Processing

PHP is the dominant server-side scripting language used to implement dynamic web content. JIT compilation, as implemented in Facebook's state-of-the-art HipHopVM [115, 1], helps to improve the performance of PHP with the help of HMI and other state-of-the-art code optimization techniques and demonstrates a significant performance increase for a set of server-side PHP applications. However, we observe that our HMI technique is not very effective for an important class of large-scale, server-side PHP applications that spend most of their time in rendering web pages or in other words, in HTML generation. Runtime characteristics of popular, content-rich PHP web applications are found to be dramatically different than the de-facto benchmark suites SPECWeb2005 [87], bench.php [111], and the computer language benchmarks [93] used so far for evaluating the performance of web servers.

Figure 1.2 depicts the distribution of CPU cycles spent in the hottest leaf functions of a few large-scale, content-rich PHP applications compared to SPECWeb2005's banking and

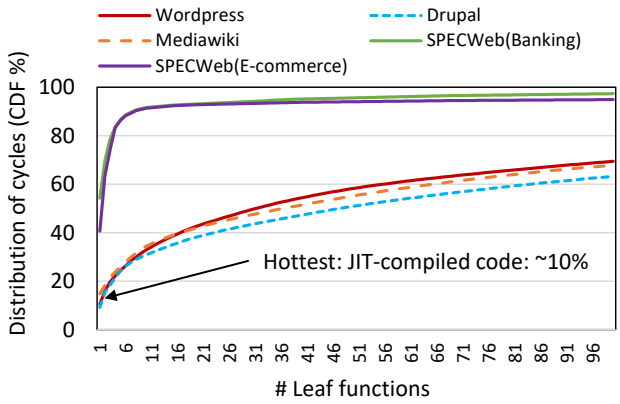


Figure 1.2: Distribution of CPU cycles of SPECWeb2005 workloads and few content-rich PHP web applications over leaf functions (details in evaluation chapter).

e-commerce workloads. Clearly, the SPECWeb2005 workloads contain significant hotspots – with very few functions responsible for about 90% of their execution time. However, the content-rich 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 in this dissertation. Despite our best effort, we could not find any obvious target or easy opportunity for architectural optimization.

Failing to find any clear microarchitectural opportunities shifts our focus towards designing domain-specific specialized hardware for these large-scale, content-rich PHP applications. 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 [4], hash table accesses for user-defined types [15, 14], etc.) or the associated overhead of garbage collected languages [8]. These observations guide us to apply several hardware and software optimization techniques from prior works [4, 15, 14, 33] *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. Our results reflect an average 17.93% improvement in performance and 21.01% reduction in energy while executing these complex, content-rich PHP workloads on a state-of-the-art software and hardware platform.

1.3 Thesis Contributions

The research presented in this dissertation makes the following contributions:

- **Proposes Hash Map Inlining to specialize accesses to hash maps in server-side PHP applications:** The overheads associated with accessing hash maps in server-side PHP applications accessing databases and other middleware are quantified. The root cause hampering inlining of such hash maps is identified and compiler enhancements to inline them so that keys can be accessed with direct offsets are implemented and

evaluated in the HipHop VM infrastructure.

- **Performs microarchitectural analysis of content-rich PHP applications:** An in-depth architectural characterization of realistic, large-scale, content-rich PHP applications is performed to identify performance and energy-efficiency bottlenecks (if any) in them. Microarchitectural analysis suggests that these PHP applications require far more BTB capacity and much larger caches than server cores currently provide to obtain even minor performance benefit.

- **Proposes domain-specific accelerators for content-rich PHP applications:** Four fine-grained PHP activities - hash table accesses, heap management, string manipulation, and regular expression handling are identified as common building blocks across leaf functions that constitute a significant fraction of total server cycles in large-scale, content-rich PHP applications. Novel, inexpensive hardware accelerators are proposed for these activities and substantial performance and energy gains are accrued across dozens of functions.

- **Evaluation of performance benefits with Hash Map Inlining:** A prototype implementation in the HipHop VM infrastructure shows promising performance benefits on real hardware for a broad array of hash map-intensive server-side PHP applications.

- **Evaluation of Domain-Specific PHP Accelerators:** Simulation results reflect that our specialized hardware offers a significant improvement in performance and a considerable reduction in energy while executing the complex, large-scale, content-

rich PHP applications on a state-of-the-art software and hardware platform.

1.4 Related Published Work

This dissertation encompasses these previously published works.

- **Hash Map Inlining (PACT - 2016).** This paper identifies the root cause hampering specialization of hash map accesses in real-world, server-side PHP applications and provides compiler enhancements to mitigate those in a state-of-the-art JIT compiler [33]. This paper was coauthored by Mikko Lipasti.

- **Architectural Support for Server-Side PHP Processing (ISCA - 2017).** This paper performs an in-depth microarchitectural analysis of large-scale, content-rich PHP applications and proposes few domain-specific hardware accelerators to improve their execution efficiency [34]. This paper was coauthored by David Schlais, and Mikko Lipasti.

1.5 Thesis Organization

This dissertation is organized as follows: Chapter 2 provides context and background for the reader including a discussion of scripting languages, PHP in particular, followed by their unique features, associated overheads and existing software and hardware solutions to mitigate them. Chapter 3 presents the Hash Map Inlining technique for streamlining access to dynamically-shaped hash maps, the most commonly occurring data structure in server-side PHP applications, and Chapter 4 evaluates its efficacy. Chapter 5 introduces domain-

specific accelerators to improve further the execution efficiency of large-scale, content-rich PHP applications. Chapter 6 discusses the methodology to simulate those domain-specific accelerators and presents the simulation results. Finally, Chapter 7 concludes the dissertation and discusses avenues for future work.

2 BACKGROUND

This chapter provides background and context for the work presented in this dissertation. It begins with a general overview of scripting languages and discusses their prevalence in today's programming world. This is followed by a brief introduction to the PHP scripting language in Section 2.2. This section presents PHP's key language features that are more general than C++-like compiled languages but are responsible for causing significant obstacles to good performance. This section also provides an outline of automatic memory management, a key feature of PHP like scripting languages. The following sections discuss PHP's standard implementation and few existing software and hardware solutions to mitigate PHP's performance bottlenecks. We adopt them in our simulation environment. Finally, this chapter ends with outlining a hardware solution in mitigating the overhead associated with its automatic memory management, which we adopt in our simulation environment as well.

2.1 Scripting Languages

Scripting languages are becoming more and more important in today's programming world as they are used for an increasing share of application software. Their popularity is due in large part to their flexible syntax and dynamic type system, rich built-in libraries, and ease of use, which enable rapid turnaround time for prototyping, and developing production-grade applications. In general, scripting languages are easier to learn, thus in turn enable high programmer's productivity than more structured and compiled languages such as C and C++. Besides, in order to ensure high-productivity programming environments, popular

languages for large-scale scripting often provide some very appealing features, including sophisticated pattern matching and string manipulation, automatic memory management, object-oriented programming paradigm, and high-level data structures. More importantly, scripting languages are typically interpreted (rather than compiled), thus providing a fast cycle to modify, test and release application code updates in large-scale systems. Being interpreted, scripting languages do not require an explicit compilation step. In today's world, people use scripting languages for variety of purposes, including Perl, Ruby, or Python for general-purpose programming, Lua for game programming and writing plug-ins, PHP or Javascript for web programming, and R or Matlab for mathematical or statistical computing.

Despite their considerable increase in popularity, their performance is still the main impediment for developing large applications. In other words, the appealing features of scripting languages do not come for free. In order to support their dynamic features and ensure their ease of use, as these languages are commonly interpreted at runtime, applications implemented in scripting languages have significant runtime overheads in comparison to their corresponding implementations using traditional compiled languages such as C and C++.

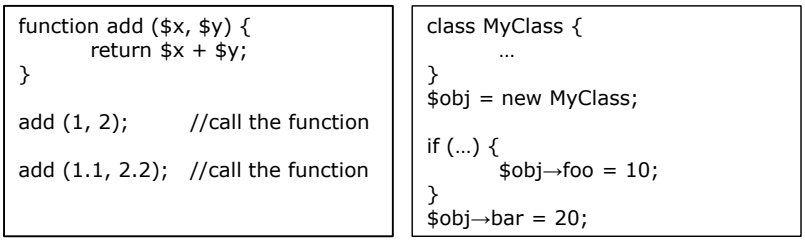
This has triggered a number of efforts to mitigate the performance overheads of scripting languages through just-in-time (JIT) compilation also commonly known as dynamic translation [7, 30, 38, 77, 92, 106, 115, 1], where the compilation is done during execution of a program – at runtime – rather than prior to execution. Statically typed languages such as C or C++ can leverage sophisticated analyses to perform classic and aggressive compiler optimizations. However this is not entirely feasible for JIT-compiled scripting languages

where such complex analyses must happen at runtime and the overhead of applying any such analyses may potentially outweigh the benefit gained from compilation. As a result a JIT compiler typically continuously analyses the code being executed and compiles frequently executed regions of the code where the speedup gained from compilation would outweigh the overhead of compiling that code.

2.2 PHP Background

PHP is a server-side scripting language used primarily to implement application logic for dynamically created web pages. PHP code may be embedded into HTML or HTML5 markup, or it can be used in combination with various web template systems, web content management systems and web frameworks. Among general-purpose scripting languages used for server-side web development to access databases and other middleware, PHP is the most commonly used [115, 105], representing over 80% [99] of all web applications. Considering the explosive growth in the scale of users and digital data volumes in the last decade, improving PHP's performance will play a significant role in drastically improving the server efficiency of Internet companies, leading to significant cost reductions for both provisioning and operating large data centers.

Although PHP syntactically resembles C++, PHP is inherently much more dynamic in nature. The next section briefly discusses the key language features that are more general in PHP than in C++. These PHP features typically cause significant obstacles to good performance and their more restricted C++ implementations allow better execution efficiency.



(a) Dynamic type of variables. (b) Dynamic addition of properties to class objects.

Figure 2.1: PHP code examples.

2.2.1 Dynamic Features

Dynamic Typing. In PHP-like scripting languages, variables can hold values from different types during an execution. Figure 2.1a illustrates a simple PHP example where a function `add` accepts two input arguments, performs an addition operation and returns the result. The function `add` in the example is invoked with two input arguments `$x` and `$y` holding either integer or floating-point values. When `add` is called with `$x` and `$y` as integers, it performs an integer addition operation, whereas when the same function is called with `$x` and `$y` as floating-point values, it performs a floating-point addition operation and prints a floating-point value to the standard output.

Dynamic Properties. Another prominent feature of PHP like scripting languages is their ability to add new properties to class objects on the fly without having to change the type declaration of the native object. As shown in Figure 2.1b, the two properties *foo* and *bar* can be added to the *MyClass* object at different memory offsets depending on the intervening branch.

Another prominent features of PHP is dynamic name binding, where the mapping of names to function declarations and classes is decided dynamically. Besides, the names of

variables, properties, functions, and classes may exist in variables.

Note that this is fundamentally different from C++-like statically-typed languages, where programmers must declare the exact types of variables, and a class object can not be extended to integrate new fields and methods or override existing ones during execution.

2.2.2 Automatic Memory Management

Automatic memory management (garbage collection) is a key feature of scripting languages because it relieves the programmer from the error-prone task of freeing dynamically allocated memory when memory blocks are not going to be used anymore. Without this automatic garbage collection (GC) feature, large software systems will be susceptible to memory leaks and dangling-pointer bugs [42]. As a result, many recent high-level programming languages (including PHP-like scripting languages) use GC as a feature.

GC basically distinguishes memory objects reachable from a valid reference or pointer variable ("live objects") from unreachable objects ("dead objects"). Algorithms that determine object reachability fall primarily into either of these two categories: reference counting [16] or pointer tracing [66]. Reference counting keeps track of the number of references (pointers) to every object. When this count goes down to zero, the object becomes dead. On the other hand, pointer tracing recursively follows every pointer starting with global, stack and register variables, scanning every reachable object for pointers and following them. Any object not reached by this exhaustive tracing then becomes considered as dead. Once the GC identifies dead objects, it makes their memory blocks available to the memory allocation engine for future use.

```
// load pointers of RB and RC to rb and rc
Value * rb = RB ( bc ); Value * rc = RC ( bc );
Number nb , nc ;
// check if rb and rc is integer
// if so , calculate sum and set type
if ( isInt ( rb ) && isInt ( rc )) {
        type ( ra ) = INTEGER ;
        ival ( ra ) = ival ( rb ) + ival ( rc );
// convert rb and rc to float
// if successful , caculate sum and set type
} else if ( toNumber (rb ,& nb ) && toNumber (rc ,& nc ) ){
        type ( ra ) = FLOAT ;
        fval ( ra ) = nb + nc ;
} else {
        /* handle exception cases */
}
```

Figure 2.2: Bytecode ADD in C.

2.3 Standard PHP Implementation

2.3.1 Interpreter

In order to support various dynamic features of PHP as described in the previous section, a PHP code is typically interpreted at runtime. PHP's standard interpreter is the Zend engine [111]. It converts PHP code into lower-level bytecode (an intermediate language) instructions which are then interpreted and executed one at a time.

Since PHP allows a variable to hold any type of data without having to explicitly declare its type, each Zend bytecode instruction essentially requires to first check the types of the two operands before performing any operation on them. Figure 2.2 shows the underlying implementation of the `ADD` bytecode in a Zend execution environment, where the bytecode essentially examines the types of the two input arguments before performing any operation on them [49].

Furthermore, as the types of objects depend on the underlying execution of a program in PHP, accessing a given property can not be accomplished using a simple offset access

from the start of the object. Thus, in PHP, each property access requires a dictionary lookup to resolve its location in memory.

Another major source of inefficiency in the Zend engine comes from PHP's support of dynamic name binding. The mapping of names to variables and function declarations may change in PHP during execution. As a result, each time a PHP code executes a function or accesses the value of a variable, the Zend engine needs to consult a lookup table to disambiguate the declaration of the function or the dynamic name of the variable at runtime. The dynamic name binding feature becomes more expensive for classes, as it requires composing class methods, properties, and constants along with those from parent classes and traits besides performing a number of semantic checks [1].

While interpreters have the benefits of simplicity and portability, they incur high CPU overheads [1]. Usually these interpreted implementations are one or two orders of magnitude slower compared to their corresponding implementations in compiled languages [93]. Modern JIT compilers use specialization to mitigate these overheads from supporting the various dynamic features.

2.3.2 Just-in-time Compiler

The HipHop Virtual Machine (HHVM) [1] is the current state-of-the-art JIT compiler and runtime for PHP. HHVM converts PHP code into an intermediate bytecode, which is then translated into machine code dynamically at runtime by a JIT compiler, resulting in significant performance improvements.

When a PHP script is invoked, HHVM attempts to discover the types of the variables

by symbolically executing the bytecode instructions. Starting from the input variables, symbolic execution propagates their known types (if any) through the PHP script in an attempt to derive the types of the remaining variables in the script. Furthermore, there exists static type checkers such as Flow [28] that can infer the static types (static in the developer's mind) of the variables in a dynamically-typed code automatically wherever possible. This is based on the empirical evidence that most dynamically-typed code is implicitly statically typed; even though types may not appear anywhere in the code, they are in the developer's mind as a way to reason about the correctness of the code. By adding static typing to dynamically-typed code wherever possible, Flow-like static type checkers can then improve developer productivity and code quality and can find type errors without requiring any changes to the code. For any variable types that are not discovered by the symbolic execution or by such static type checking, HHVM then observes their dynamic types during the execution of the script. HHVM essentially depends on the empirical evidence that, at run time, the dynamic type of a variable at a given access site tends to stay consistent, and the type of an object tends not to change. HHVM records the most frequently observed dynamic type and specializes accesses for that type. A runtime check ensures that the assumptions used in the generation of specialized code hold at run time. If the check passes, HHVM can optimize a basic block[1] with all the type information collected during symbolic execution and can perform classic compiler optimizations such as constant propagation, dead code elimination, etc. If the check fails, the HHVM runtime re-specializes the access for the new type observed. HHVM-like modern JIT compilers use

[1] A basic block is a single-entry, multiple-exit region of the source program, annotated with the types of all input values that flow into it.

shadow classes to capture different types of a dynamic object. The basic idea is similar to the notion of dynamic types in Self [15, 14]. Objects that are created in the same way are grouped in the same shadow class. Each time a new property is added or a previously-seen property is introduced in a different order, JIT creates a new shadow class to capture that as a new type of the object.

In short, while emitting machine code for an access site, a modern JIT compiler guards *only* on the observed types of the variables or the shadow classes of the objects before performing any associated operations on them, instead of checking their types for every possible permutation and combination. As a variable or an object must carry a type tag, and a type guard must be executed before any operation is performed on them, the mere type check amounts to a very significant overhead even in a JIT-compiled code [68, 4, 19, 48, 49]. There are many recent proposals [42, 68, 4, 48, 49] for mitigating these abstraction overheads implicitly in hardware. The next section discusses a few of those hardware proposals.

2.4 Hardware Support for Mitigating Overheads

2.4.1 Dynamic Type Checking

While hardware support for dynamic type checking dates back to 1970s-80s (e.g., LISP machines), there has been extensive research in the last decade to reduce the impact of type checking in hardware [68, 4, 48, 49].

One of the research proposals is Checked Load [4] that proposes an ISA extension

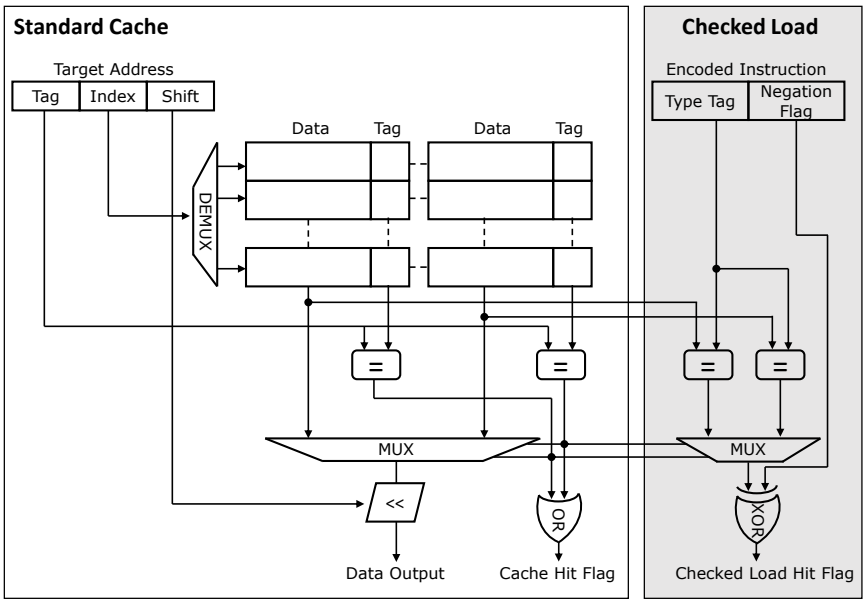


Figure 2.3: Implementation of type tag checking in parallel with cache tag checking.

(checked load instruction) to reduce the performance overhead of dynamic type checking. The key idea of Checked Load is to retain the high-level type information for each variable in the cache subsystem. A checked load instruction takes as operands a memory location of the data to be loaded, a word-sized register for the destination, and a byte-sized immediate for the type tag. When a checked load instruction is executed, the type tag field of a cache line is checked against the tag immediate in addition to accessing the value from the cache, thus significantly reducing the dynamic instruction count responsible for performing the type checking in software.

Figure 2.3 shows the implementation of the Checked Load proposal in the cache subsystem. On a cache hit, in parallel with the cache tag comparison, the type tag is compared against the relevant portion of the cache line in the set before returning the cache value to the application. Just as the cache tag comparison selects which way contains the target address, it also selects which result of the type tag comparisons to use. If the stored type tag

does not match with the intended type tag of a checked load instruction, control transfers to an error handler.

2.4.2 Reference Counting

The HHVM JIT compiler uses the reference counting mechanism to perform garbage collection. As reference counting can find dead objects as soon as they become unreachable, it eliminates the need for explicit garbage collections, which determines reachability by periodically tracing all pointers. However, the overhead of updating the reference counts on every pointer creation and destruction imposes significant overhead in software. The overhead is even higher in multi-threaded systems, which require synchronization for all reference count updates.

The Hardware-Assisted Automatic Memory Management (HAMM) proposal [42] introduces minimal changes to the cache subsystem to reduce this overhead from updating reference counts of memory objects. It essentially introduces new ISA instructions to process the resulting reference count updates implicitly in hardware while simultaneously accessing such memory objects from caches. These reference count updates are consolidated in Reference Count Coalescing Buffers (RCCB) (which are tightly coupled to caches) to reduce the frequency of these updates propagated to memory-resident objects. The zero flag is raised upon reaching a value of zero of a reference count update, in which case the code branches to a software handler to release the memory object to heap manager. The RCCBs thus can filter most of the reference count updates, and can significantly reduce the instructions responsible for performing this operation in software.

2.5 Summary

The chapter provides a brief introduction to PHP, its key dynamic features and their associated overheads. PHP's standard interpreter implementation and JIT compilation that support its various dynamic features are outlined next. Though effective, JIT compilation can not mitigate the overheads from supporting PHP's dynamic features completely in software. As a result we adopt an existing hardware solution – Checked Load in our baseline simulation environment to mitigate PHP's type checking overhead in hardware. Additionally the chapter provides an overview of automatic memory management. As PHP, like all garbage collected languages, suffers from the overhead of reference counting, we present a hardware proposal from prior work and adopt it in our baseline simulation environment.

3 HASH MAP INLINING

This chapter proposes Hash Map Inlining, a JIT compiler optimization technique to minimize the overheads associated with accessing key-value pairs stored in hash maps, the most commonly occurring data structure in server-side PHP applications. Section 3.1 provides an overview of Hash Map Inlining. The remainder of this chapter is organized as follows. Section 3.2 describes how we adapt inline caching, a technique for streamlining access to dynamically-typed objects, to similarly improve the performance of hash maps. Section 3.3 describes how PHP uses hash maps to interface with SQL databases. Section 3.4 explains why our initial HMI algorithm fails to work with the SQL interface, and describes how we extend it to capture this opportunity. Section 3.5 presents details of our modifications to HHVM. Section 3.6 discusses related work and Section 3.7 concludes the chapter.

3.1 Overview

As previously discussed, the most common complex data structure in PHP like scripting languages is the hash map, which is used to store key-value pairs. Unfortunately, the runtime overhead of key lookup and value retrieval is quite high, especially relative to the direct offsets that compiled languages can use to access class members.

The HipHop Virtual Machine(HHVM) [1] PHP JIT compiler from Facebook shows tremendous gains to close the performance gap with statically-typed languages such as C++. Its basic design of a stack-based bytecode compiled into type-specialized machine code provides large speedups for diverse, real-world PHP applications when compared to an

interpreted environment. However, as observed in this dissertation, in a set of popular real-world PHP scripts that power many e-commerce platforms, online news forums, banking servers, etc., hash map processing constitutes a large fraction of the overall execution time and failure to optimize accesses to those hash maps by HHVM causes a substantial performance bottleneck. More specifically, server-side PHP scripts will commonly retrieve information from a back-end database management system (DBMS) engine by issuing a SQL query [70], the results of which are communicated to the PHP script as key-value pairs stored in a hash map. The key-value pairs are subsequently processed by application logic in the PHP code to generate dynamic HTML content. Considering the fact that a considerable fraction of the execution time of these scripts are spent on processing such hash maps (as our results indicate), the cost of populating and accessing these hash maps should be reduced in order to reduce script execution time.

In this dissertation, we propose *Hash Map Inlining (HMI)* to minimize the overheads associated with populating and accessing key-value pairs stored in hash maps. HMI is a dynamic optimization technique that is triggered whenever runtime profiling indicates that hash maps are being populated and accessed in a *hot*[1] region of the PHP program. HMI dynamically converts a hot hash map into a vector-like data structure that is accessed with fixed, linear offsets for each key value, and specializes the code at each hot access site to use fixed offsets from the HMI base address to update and/or retrieve values corresponding to each key.

Our implementation of HMI is inspired by *inline caching*, an existing approach for specializing code that accesses members (or fields) in dynamically-typed objects. JIT

[1] A hot region is a region of a program where most time is spent during the program's execution.

compilers for scripting languages that support dynamic type systems (e.g. Chrome V8 for Javascript [100] and HHVM for PHP [1]) rely on a *shadow class* system to map object field names to offsets for each instance of a dynamic object. With inline caching, code that accesses these fields is specialized to short-circuit expensive offset lookups by including an efficient shadow class type check in the specialized code, followed by a direct offset-based access to the field as long as the type matches the common case. By analogy, HMI treats hash map keys as field names, and specializes direct-offset accesses to the corresponding hash map values while protecting them with a type check similar to the one used for inline caching.

We demonstrate the performance benefits of our implementation of HMI by way of the microbenchmark (previously discussed in Figure 1.1 of Chapter 1) that repeatedly updates and accesses a configurable number of key-value pairs stored in a hash map. We show that the vast majority of the overhead of hash map accesses can be elided, leading to gains of up to $40-45\%$ (with a hash map of 10 or 50 key-value pairs) with our prototype HMI implementation, running on real hardware. The performance gain goes up with bigger hash map sizes.

However, we find that our initial HMI implementation delivers only marginal gains when applied to real-world PHP applications that utilize hash maps to retrieve information from a back-end DBMS. The effectiveness of HMI in these applications is limited for two reasons. First, the hash maps are populated inside SQL runtime libraries written in C code, which are not visible to the HHVM optimizer, effectively preventing HMI from triggering inlining and code specialization. Second, our initial version of HMI can only specialize code for accesses where the hash map keys are specified as literal values at the access site

(e.g. myhashmap["literalkey"]), whereas these applications commonly specify the keys as variables (e.g. myhashmap[$myvariablekey]). In theory, flow analysis and constant propagation would reveal that some of the latter cases are in fact constants (literals), but this was not the case for the applications we examined. Instead, we found that the variables at each access site would sequence through a number of different, though predictable, key names at run time.

In order to address these shortcomings, we extended our HMI implementation in two ways. First, we wrote new versions of the SQL runtime library functions used to access DBMS contents: ones that directly utilize inlined hash maps for communicating query results to the PHP scripts. Second, we augmented the HHVM JIT to first check for the necessary set of conditions that trigger correct use of these HMI-friendly functions, and then to specialize any qualifying call sites to call them instead of the original functions. We elaborate on the necessary set of conditions in Section 3.4: we can invoke HMI whenever we can guarantee that there is a finite and ordered set of keys that are used to populate the hash map. This condition is trivially satisfied for the SQL runtime library functions we targeted, since the ordered set of keys is determined by the database schema, which is fixed at the time the SQL query is evaluated. In other cases, this condition could also be satisfied based on PHP language semantics. For example, the *foreach* array iterator in PHP iterates over the key-value pairs in the hash map in a fixed order, providing the same guarantee of a finite and ordered set of keys. These two situations allow automatic conversion of hot hash maps into inlined form for these PHP applications, such that subsequent accesses within the PHP code can be efficiently specialized to take full advantage of the inlined hash map structure. Our prototype implementation in HHVM shows performance benefits for a

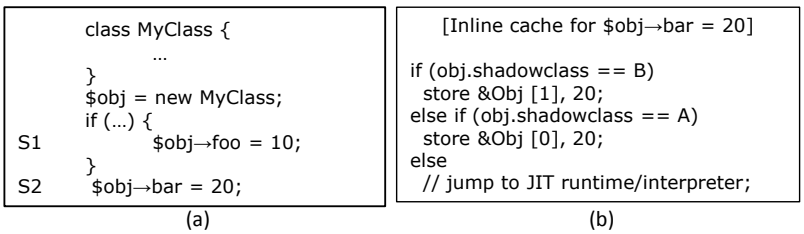


Figure 3.1: Example PHP code (a) and inline cache to access property *bar* (b).

broad array of hash map-intensive PHP scripts, up to 37.6% and averaging 18.81%, and improves SPECWeb throughput by 7.71% (banking) and 11.71% (e-commerce).

3.2 Inline Caching for Hash Maps

3.2.1 PHP Scripting Language

In PHP variables can hold values from different types during an execution. One prominent feature of PHP is its ability to add new properties to class objects on the fly without having to change the type declaration of the native object. However, the absence of declared types makes it very challenging for the compiler to generate code, as the types of objects depend on the underlying execution of the program. As a result, accessing a given property can not be accomplished using a simple offset access from the start of the object. As shown in Figure 3.1(a), the two properties *foo* and *bar* can be added to the *MyClass* object at different memory offsets depending on the intervening branch. Thus, in PHP, each property access requires a dictionary lookup to resolve its location in memory.

3.2.2 Inline Caching for Dynamic Classes

Modern JIT compilers use specialization to mitigate this problem. They essentially depend on the empirical evidence that, at run time, the dynamic type of an object at a given access site tends to stay consistent. The JIT compiler records the most frequently observed dynamic type and specializes accesses for that type. A runtime check ensures that the assumptions used in the generation of specialized code hold at run time. If the check fails, the JIT runtime re-specializes the access for the new type observed. Modern JIT compilers use *shadow classes* to capture different types of a dynamic object. The basic idea is similar to the notion of dynamic types in Self [15, 14]. Objects that are created in the same way are grouped in the same shadow class. Each time a new property is added or a previously seen property is introduced in a different order, JIT creates a new shadow class to capture that as a new type of the object.

Figure 3.2 shows how a JIT compiler creates shadow classes for the code shown in Figure 3.1(a). When the runtime enters this code section for the first time, JIT creates an empty *MyClass* (not shown in Figure 3.2) pointing to an empty shadow class type A. Now if a *Not Taken(NT)* branch direction is observed, the runtime adds properties *foo* and *bar* to *Obj1* and creates two more shadow classes B and C of *Obj1* at access site *S1* and *S2* respectively. Shadow classes record the added properties and their offsets. Any future invocation of this code with the branch observing again a *NT* direction will cause the runtime to follow the same shadow class transitions at *S1* and *S2*. Subsequently when a *Taken(T)* branch direction is observed, the runtime will create a new shadow class type D from initially empty type A. So as observed here, access site *S2*'s addition of property *bar*

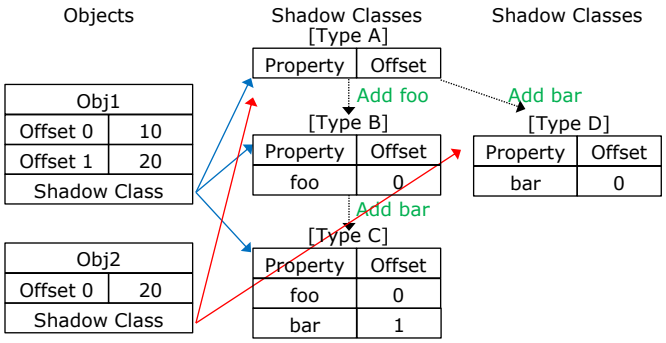


Figure 3.2: Example of shadow classes.

results in an object of either shadow class C or D. This supports the common assumption that at a given site the JIT runtime accesses objects of very few types. Modern JIT compilers use a technique called *inline caching* to exploit this assumption to optimize accesses to properties of objects at a given site. The inline caching mechanism essentially caches the offsets of the property *bar* for the two object types C and D seen before at access site *S2* and specializes the site *S2* as shown in Figure 3.1(b). Any future invocations of the code, regardless of the intervening branch's behavior, will be able to exploit the cached offsets to map *bar* either to type C or D.

3.2.3 Adapted to Hash Maps

In this work, we extend this inline caching approach for coping with dynamically-typed objects to also enable specialization of accesses to hash maps. By analogy, our HMI implementation treats hash map keys as property names. Algorithm 1 and 2 describe our initial HMI implementation inspired by inline caching. Note that similar to the inline caching approach, Algorithm 1 and 2 can only specialize code for accesses where the hash map key is specified as a literal value at the call site. With HMI in Algorithm 1, a call site that has already generated specialized code can perform a direct-offset access to the *Key* as

Algorithm 1 HMI Populate based on inline caching

Input: Hash Map *h*, *Key*, *Value*, *CallSite PC*

```
1: if Key.IsStaticLiteral is True then
2:     if Inline cache found for CallSite PC then
3:         Perform inlined populate (CallSite PC, Key, Value)
4:     else if h.IsInlined is True then
5:         h.SymbolTable.NextKeyOffset++
6:         Offset ← h.SymbolTable.NextKeyOffset
7:         h.SymbolTable[Offset] ← (Key,Offset)
8:         h.Data[Offset] ← Value
9:         /*Generate inlined populate for CallSite PC*/
10:        JIT.generateInlineCache(CallSite PC,Key,Offset)
11:    else if Is CallSite PC Hot then
12:        h.IsInlined ← True
13:        h.SymbolTable.NextKeyOffset ← 0
14:        Offset ← h.SymbolTable.NextKeyOffset
15:        h.SymbolTable[Offset] ← (Key,Offset)
16:        h.Data[Offset] ← Value
17:        /*Generate inlined populate for CallSite PC*/
18:        JIT.generateInlineCache(CallSite PC,Key,Offset)
19:    else
20:        Profile (CallSite PC, Key)
21:        Regular hash map populate (h, Key, Value)
22:    end if
23: else
24:    /* Key is not a static literal */
25:    Regular hash map populate (h, Key, Value)
26: end if
```

Algorithm 2 HMI Access based on inline caching

Input: Hash Map *h*, *Key*, *CallSite PC*

```
1: if h.IsInlined is True then
2:     if Key.IsStaticLiteral is True then
3:         if Inline cache found for CallSite PC then
4:             Perform inlined access(CallSite PC, Key)
5:         else
6:             Offset ← h.symbolTableLookup(Key)
7:             /*Generate inlined access for CallSite PC*/
8:             JIT.generateInlineCache(CallSite PC,Key,Offset)
9:             return h.Data[Offset]
10:        end if
11:    else
12:        /* Key is not a static literal */
13:        Regular hash map access (h, Key)
14:    end if
15: else
16:    Regular hash map access (h, Key)
17: end if
```

long as the type of the hash map *h* matches with any previously observed types at the site. When it is the first time that the hash map *h* is accessed in a hot region of the PHP program, HMI converts it into a vector-like data structure, adds the *Value* into the first location and records the location information or offset in a table structure, which we call the *Symbol Table*. The symbol table essentially captures the type of a hash map by recording the offsets

of inserted keys. Before performing a direct access to a inlined hash map at a call site, the type checking step in HMI finds the appropriate symbol table for the hash map from the set of symbol tables cached previously at the call site. In order to retrieve values from a inlined hash map, Algorithm 2 can look for a key in the generated symbol table of the hash map and specialize the code at that access site to use direct fixed offsets.

In order to investigate the performance impact of our initial HMI implementation, we apply this to a microbenchmark that repeatedly updates and accesses a configurable number of key-value pairs stored in a hash map and observe that the vast majority of the overhead of hash map accesses can be elided. This results in significant performance gains of up to $40-45\%$ with a hash map of 10 and 50 key-value pairs. This benefit primarily comes from the substantial reduction in branch mispredictions, caches misses and overall instructions enabled by direct-offset access from HMI.

However, we observe that our initial HMI implementation delivers marginal or no gains when applied to real-world PHP applications (Table 4.1) that utilize hash maps to retrieve information from a back-end DBMS. SPECWeb(E-commerce) suite shows marginal benefit of about 1.14% when compared against the unmodified HHVM, whereas the remaining benchmark suites do not deliver any visible performance improvement. Before we investigate the reasons behind its poor performance with real-world applications, we study the SQL interface in the next section that communicates with back-end DBMS and executes such PHP applications.

```
$q_result = mysql_query("SELECT id, name, initial_price,
max_bid, nb_of_bids, end_date FROM items WHERE
category=$categoryId
AND end_date >= NOW()...");

while ($q_row = mysql_fetch_array($q_result)) {
  $maxBid = $q_row["max_bid"];
  if ($maxBid == 0) $maxBid = $q_row["initial_price"];
  print("<TR><TD><a href=\"/PHP/...itemId=".$q_row["id"].
  "\">".$q_row["name"]."<TD>$maxBid".
  "<TD>".$q_row["nb_of_bids"].
  "<TD>".$q_row["end_date"].
  "<TD><a ...PutBidAuth.php?itemId=".$q_row["id"]...");
}
```

Figure 3.3: Code snippet of a server-side PHP script.

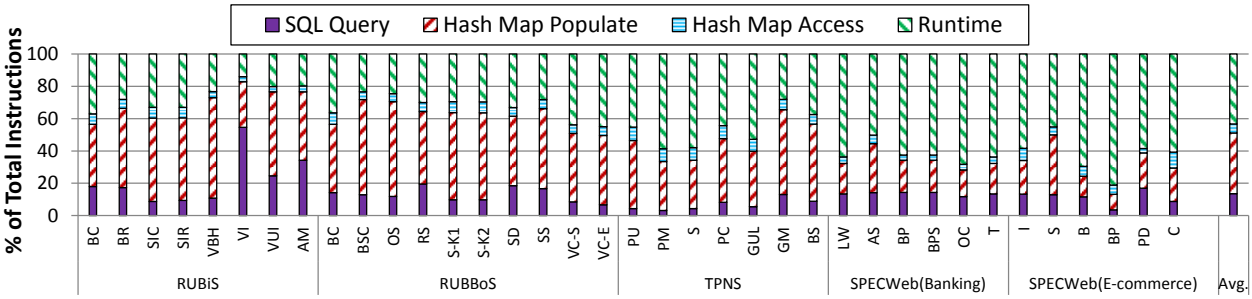


Figure 3.4: Breakdown of dynamic instructions, measured using Pin [63]. Averaged across scripts in Table 4.1. Runtime = instructions that execute string operations, regular expressions, and miscellaneous operations.

3.3 Hash Map Interface to SQL DBMS

When a client makes a HTTP request to a web server, the server usually invokes PHP scripts to serve the request. The PHP scripts in turn formulate the necessary query plans and dispatch those queries to the backend DBMS engine (e.g., SQL or memcached). Once the DBMS engine produces the query result tables, the PHP scripts utilize standard SQL library functions to iterate over the rows in the tables before sending a response with dynamic HTML contents back to the client. Figure 3.3 illustrates a example PHP script from a real-word benchmark suite RUBiS [80]; it follows the structure of a typical server-side PHP script as described above.

Note that retrieving the rows from the query result table *q_result* and mapping their

```
Variant mysql_fetch_array (const Resource& mysql_result, ...) {
    .....
1)  Array ret; /*Allocate an empty array, resize as per requirement*/
2)  MYSQL_ROW mysql_row = mysql_fetch_row(mysql_result);
3)  long *mysql_row_lengths=mysql_fetch_lengths(mysql_result);
4)
5)  for (mysql_field = mysql_fetch_field(mysql_result), i = 0;
6)        mysql_field;
7)        mysql_field = mysql_fetch_field(mysql_result), i++) {
8)      if (mysql_row[i]) {
9)              data = mysql_makevalue(String(mysql_row[i],
10)             mysql_row_lengths[i],CopyString), mysql_field);
11)     }
12)     ret.set(String(mysql_field->name,CopyString),data);
13)}
14)return ret;
}
```

Figure 3.5: Implementation of mysql_fetch_array() function.

various keys into the *q_row* hash map during the loop iterations in Figure 3.3 essentially involve repeated interpretation of hash maps. Figure 3.4 illustrates that a major fraction of the dynamic instructions of server-side PHP scripts (details in Table 4.1) are spent on populating and accessing hash maps.

Figure 3.5 excerpts the underlying implementation of the SQL library function mysql_fetch_array(), used in the PHP script in Figure 3.3 to retrieve rows and populate such hash maps from the query result table. It relies on the loop (starting on line 5) to extract the various keys of a row into the *q_row* hash map. Delving down into the details further, for each and every key, the code sequence (from line 5 to line 13) retrieves a value from the row and adds a new (key, value) pair to the hash map (line 12). Close examination of the usage of the populated hash maps inside the while loop of Figure 3.3 in conjunction with the underlying implementation of mysql_fetch_array() function in populating those hash maps (Figure 3.5) reveals the following main abstraction overheads.

First, as the mysql_fetch_array() function iterates over the rows in the result table, it allocates a hash map to hold the values associated with the different keys of a row. This introduces a significant number of expensive memory allocations and releases on the

critical path. Second, the string key is run through a hashing function to index into an entry of the hash map. The hash computation is followed by a string comparison along a linked list of possible entries to find the appropriate entry for the current key. Finally, the hash maps may need to be resized during runtime in order to accommodate more data. These resizing operations also add a modest overhead, especially when these hash maps are accessed many times during the course of parsing the entire result table. The lookup process also incurs equally high abstraction overheads in hashing and comparison functions when the different keys of the hash maps are retrieved inside the while loop in Figure 3.3. Analysis using Pin [63] shows that populating such a hash map with 4 keys inside this SQL library function requires about 2,400 instructions. Reading the hash map key inside the while loop of Figure 3.3 requires about 90 instructions.

In the next section, we investigate the shortcomings with our initial HMI implementation that could deliver only marginal or no gains when applied to real-world PHP applications that utilize hash maps to retrieve information from a back-end DBMS.

3.4 Why HMI Fails for DBMS Scripts

3.4.1 Hidden Library Functions

As described in Section 3.2, a JIT compiler like HHVM collects profile information at runtime and uses it to specialize a code section written in the scripting language (PHP in this work) of an application. However, SQL library functions, which are implemented in statically-typed languages such as C++, are not visible to the HHVM optimizer. As a

result, when hash maps are populated inside the mysql_fetch_array() SQL library function in Figure 3.5, HMI cannot capture their shape and type. Hence, when these populated hash maps are accessed later in the while loop in Figure 3.3, HMI cannot anticipate their shape despite staying the same across invocations of this library function. Consequently HMI cannot trigger inlining and code specialization to convert accesses to such DBMS hash maps to simple direct-offset accesses.

3.4.2 Variable Key Names

Even if the SQL library function is written in PHP and visible to the JIT compiler, due to the reasons detailed below, hash maps populated inside the function can not be efficiently inlined using the initial HMI implementation.

Note that, regardless of which control path is followed in the example in Figure 3.1(a), the name of the properties at the two access sites *S1* and *S2* are static literals. But in case of populating hash maps inside mysql_fetch_array(), as different keys are populated at the same access site (line 12 in Figure 3.5), the HMI runtime does not observe any static literal keys; instead it ends up observing a variety of keys at that access site. In other words, our initial version of HMI can only specialize code for accesses where the hash map keys are specified as literal values at the access site, such as the key values in our microbenchmark. As a result, when we attempt to apply Algorithm 1 or 2, they fall back on the expensive dictionary lookup.

In order to address this shortcoming, we augment the initial HMI implementation (Algorithm 1 and 2) to handle keys with variable names. However, the HMI runtime

```
                Inline cache for ret.set(…) (line 12 in Figure 6)
if (ret is a DBMS hash map &&
    ret.Type == Type_NO_Key) {    /*Type_NO_Key = empty hash map*/

   if (ret.current_key == "x")      /*String comparison to check current key
      return ret[cached_x_offset];   matching against 1st key of schema */
}
else if (ret. Type == Type_K_x) {  /*Type_K_x = hashmap with key "x"*/

   if (ret.current_key == "y")      /*String comparison to check current key
      return ret[cached_y_offset];   matching against 2nd key of schema */
}
...
else if (ret.Type == Type_K_x_y_...)
{...}
...
else {regular hash map access and inline cache update}
```

Figure 3.6: Required changes to initial HMI implementation.

now no longer observes a single static literal key when accessing the different keys at the same access site. As a result, populating different keys at the same site requires the HMI implementation to emit code that performs a string comparison to ensure the current string key matches against the key seen before with the current hash map type. Hence, the specialized code will not be able to avoid the expensive string comparison while populating hash maps inside mysql_fetch_array() SQL library function. Figure 3.6 shows the changes required to the initial HMI implementation to inline populating such hash maps at the same call site inside the SQL library function with all the associated overheads as discussed above. This example assumes the presence of key "x", followed by "y" and so on in the database schema. Note that now in order to specialize the code that inserts keys at the same call site, the HMI implementation needs to include an additional string comparison (highlighted in brown) in addition to the type check in the inlined code.

3.4.3 High Polymorphism

Figure 3.6 demonstrates the specialized code that the HMI implementation will emit to inline populating hash maps inside SQL library function at the common call site. However,

note that updating a key (for example, key y) at the common call site in SQL library function now requires a traversal through a path containing branches for all the other type checks. Hence, populating a key at this site requires performing a linear search for the correct entry in the specialized code that matches the current input types, leading to inefficient execution. A call site that only observes a single type is called *monomorphic*; if it observes multiple types, it is *polymorphic*. The more polymorphic a site becomes in specialized code, the more overhead it adds to the overall execution. Considering the fact that such hash maps inside the SQL library function in our experimental suite typically populate in the range of 6 to 18 keys, the HMI implementation will in turn make the access site highly polymorphic. Although it will avoid the hashing function to index into the hash map while populating keys, the presence of the highly polymorphic access site along with the necessary string comparison to find a cached offset will still accrue substantial overheads. Furthermore, unlike regular PHP code sections, since the SQL library functions are more likely to be called from many places in the application to satisfy various queries, the access site that populates hash maps may end up being highly polymorphic.

However, note that the HMI runtime can avoid the string comparison imposed by variable key names, and can achieve similar benefit to that of inlining accesses to static literal keys, if either of the following conditions are met:

(1) the application (or programmer) ensures a *statically* ordered set of keys being inserted at a call site or

(2) the runtime guarantees a *dynamically* ordered set of keys being inserted at a call site.

In other words, our extended HMI implementation is invoked whenever we can guarantee that the variable at an access site will sequence through a number of different, though

Algorithm 3 HMI (Handling keys with variable names)

Input: Hash Map *h*, *Key*, *Value*, *CallSite PC*

```
1: if Key.IsStaticLiteral is True then
2:     Same as line 2-22 in Algorithm 1
3: else
4:     /* Key is not a static literal */
5:     if Either application can guarantee statically or runtime can guarantee dynamically that Key belongs to ordered set of keys being
   inserted at CallSite PC then
6:         JIT.invokeExtendedHMI (section 3.5 for SQL)
7:     else
8:         /* Regular hash map populate/access */
9:     end if
10: end if
```

predictable, fixed key names at run time. Algorithm 3 describes the algorithm that triggers extended HMI implementation to inline accesses to keys with variable names at a common call site. Note that this condition is trivially satisfied for the SQL runtime library functions we targeted, since the ordered set of keys is determined by the database schema, which is fixed at the time the SQL query is evaluated.

3.5 Extended HMI for SQL

In order to address the shortcomings, we extended our HMI implementation in two ways. First, we wrote new versions of the SQL runtime library functions used to access DBMS contents: ones that directly utilize inlined hash maps for communicating query results to the PHP scripts. Second, we augmented the HHVM JIT to first check for the necessary set of conditions that trigger correct use of these HMI-friendly functions, and then to specialize any qualifying call sites to call them instead of the original functions. In case of SQL library function, since the application statically guarantees ordered set of keys being inserted at the call site (line 12 of Figure 3.5), HMI can redirect the JIT runtime to use the symbol table-generating version of mysql_query (*step* 1) and vector arrays-generating version of mysql_fetch_array thus inlining populating of hash maps (*step* 2). Once the symbol table is

```
Variant mysql_query (const String& query, ...) {
  .....
  std::vector<StringData *> mysql_key_names;
  for (mysql_field = mysql_fetch_field(mysql_result);
       mysql_field; mysql_field =
       mysql_fetch_field(mysql_result)) {

    /* Collect name of keys */
    mysql_key_names.push_back(makeStaticString
      (String(mysql_field->name, CopyString)));
  }
  /* Generate symbol table at runtime */
  define_MySQL_SymbolTable(mysql_key_names);
}
```

Figure 3.7: Generate symbol table in mysql_query(*) and expose that to the JIT runtime.

generated and exposed to the JIT runtime in *step* 1, HMI can use that to inline all future lookups or accesses to hash maps within the PHP script (*step* 3). The following three steps describe this in detail.

(a) Generate symbol table in SQL query function. When a database query is executed, the runtime accesses the associated meta-data about the relation, such as the name of the relation, the number of keys, their names, types, etc. and builds a symbol table that records different keys of the relation and their corresponding offsets. So as shown in the underlying implementation of the SQL query execution function in Figure 3.7, it is modified to collect information about the keys of the associated database schema at the end of its execution. It thus builds a symbol table, associates that with the current query plan and attaches that to the pool of symbol tables. Note that the keys are mapped to the symbol table in order of their appearance in the query result table. Before defining a new symbol table, the runtime checks if it has already declared a symbol table for the current query plan. If it finds a table with an identical set of keys inserted into it in the same order as that of the current plan in the pool of symbol tables, then it does not create a new symbol table across invocations of the query execution function and returns the old table only. When

the database query function subsequently finishes execution, it exposes this populated symbol table to the JIT runtime and returns a pointer to it along with the regular result table. The generated symbol table later is used to inline any subsequent accesses to hash maps populated from the query result table.

(b) Populate vector-like arrays in SQL fetch_array function. When a DBMS engine produces a query result table, mysql_fetch_array() extracts the rows from the table into hash maps. However, instead of retrieving rows into hash maps, HMI specializes the mysql_fetch_array() function to map the keys into vector-like arrays. Furthermore, each key is made to update the slot in that array in the order of its appearance in the query result table. That way it entirely eliminates the abstraction overheads associated with populating hash maps inside this library function.

Any subsequent accesses to these populated vector-like arrays within the PHP script can then determine the mapping of the keys of the extracted rows into these arrays using the symbol table generated in mysql_query() function above. Thus the extended HMI implementation can inline accesses to such hash maps populated inside the SQL fetch_array() function. This becomes possible since the symbol table maps the keys of the relational schema in the mysql_query() function in the same order as the values of different keys from the extracted rows are inserted to form vector-like arrays in mysql_fetch_array() function. HMI essentially decouples hash map accesses from populating vector-like arrays inside the library function. Considering the fact that these hash maps are populated many times during the course of parsing the entire query result table, this decoupling will eliminate the overheads associated with populating those hash maps entirely. Generating a symbol

```
Variant mysql_fetch_array (const Resource& mysql_result, ...) {
  .....
  /*Allocate vector-like array of size mysql_num_fields()*/
  ret.init_mysql_array (mysql_num_fields(mysql_result));

  for ( mysql_field = mysql_fetch_field(mysql_result), i = 0;
        mysql_field;
        mysql_field = mysql_fetch_field(mysql_result), i++) {
        if (mysql_row[i])
                data = mysql_makevalue(String(mysql_row[i],
                mysql_row_lengths[i],CopyString), mysql_field);
        }
        ret.set(i, data); /*Populate vector-like array locations*/
  }
  return ret;
}
```

Figure 3.8: Populate vector-like arrays of size mysql_num_fields(*).

table should add insignificant overhead to a query's execution time.

So, as shown in the underlying implementation of mysql_fetch_array() function in Figure 3.8, the hash map of Figure 3.5 has been replaced by a vector-like array with size equal to the number of keys in the query relation. As a result, the arrays are no longer required to be resized during runtime in order to accommodate all the keys of the rows and hence they can avoid the resizing and its associated overhead. However, populating any keys not present in the symbol table into vector-like arrays requires falling back to the runtime and using expensive hash map lookups.

(c) Use symbol table to inline accesses to hash maps. As discussed before, a pointer to the symbol table generated during the execution of a SQL query plan is attached to the vector-like arrays populated from the query result table. As a result, HMI can later use that symbol table to inline accesses to hash maps within the PHP script. It is commonly observed that the shape of hash maps generated inside the mysql_fetch_array() library function tends to be always consistent at a given access site. This holds true for our experimental workloads also. However, if the query plan is designed in such a way that it depends on

the schema of the connected database (a database query such as, SELECT * From a Table), then the mysql_query() execution might return symbol tables with different shapes and different sets of keys across invocations. As a result, before performing an inlined access to the hash map, an access site must check the shape of the symbol table to determine if it has changed from the last time the query was executed. Any change in the shape of the symbol table (set of keys associated with the symbol table or their order of insertion into it) from the last time will necessitate HMI to re-specialize for the new shape and find the offset of the key using the new symbol table.

As a result, while specializing an access site that retrieves values from hash maps populated inside a database library function, HMI first checks the shape of the symbol table attached to the vector-like array. HMI checks if the symbol table matches the cached symbol table seen earlier at this site. If so, the key-value in the hash map can be accessed using a simple cached offset at this site. Figure 3.9(a) shows the symbol table generated with executing the query in Figure 3.3 and Figure 3.9(b) illustrates the specialized code to access the "max_bid" key. Reading "max_bid" from the hash map *row* in Figure 3.3 is now guarded by the cached symbol table. If it succeeds, "max_bid" is accessed with simple offset-access. However, if the runtime encounters a symbol table it has not seen before, HMI re-specializes the access on the new symbol table observed.

In summary, in case of accessing hash maps with literal keys, traditional inline caching and HMI inline key accesses in the same way as shown in the example of Figure 3.2. However, for accessing hash maps with variable key names as in Figure 3.3, HHVM will invoke the HMI-friendly version of SQL library functions to populate vector-like arrays instead of hash maps and thus will inline inserting keys to hash maps. The symbol table

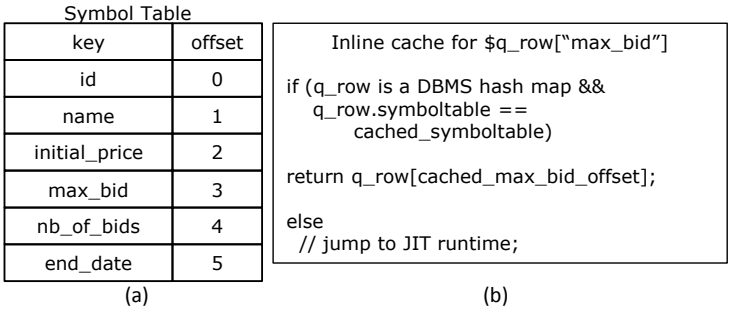


Figure 3.9: Generated symbol table (a) and inline cache to access key *max_bid* (b).

generated (Figure 3.9(a)) while executing the SQL library functions will then be used to specialize accesses to keys such as "max_bid" (Figure 3.9(b)).

3.5.1 Multiple Call Sites

Since the SQL query function can be called from many call sites in a program to satisfy various queries, HMI must generate different symbol tables to capture various database schemas. When updating a symbol table in the SQL query function, the HMI implementation ensures that each query execution points to a distinct symbol table dictated by the program counter of the call site in the bytecode. Note that HMI does not need any extra provisions to handle multiple call sites for the mysql_fetch_array() function. Each of the instantiations of this function receives a distinct symbol table from their corresponding SQL query function, which they attach to the vector-like arrays populated inside them.

3.5.2 Other DBMS Engines and Languages

There exist popular DBMS engines other than MySQL such as MongoDB [69], Oracle DB [71] etc. They have similar methods for accessing database results. For example, MongoDB uses a collection method (*find()*) in conjunction with cursor methods (*forEach()* or *hasNext()*)

(analogous to *mysql_query()* and *mysql_fetch_array()* methods from MySQL) to collect hash maps from database results. Hence these PHP applications accessing MongoDB servers instead will face similar hash map overheads and can benefit from HMI.

In this work we primarily target server-side PHP applications. However, server-side workloads developed in any other scripting languages may have similar hash maps with variable key names and thus can benefit from HMI. Creating HMI-friendly versions of DBMS library functions in Figure 3.7 and 3.8 introduce minor changes to the HHVM's C++ library. Furthermore, we added about 150 lines of code in HHVM to capture the opportunity for HMI and generate hash map-inlined code. Hence our proposed changes can be easily applied to other DBMS engines and JIT compilers.

3.5.3 Applying HMI Outside DBMS Queries

HMI is a general technique that can provide significant performance benefit whenever hash maps are accessed repeatedly with a fixed set of key values. To obtain this benefit, the PHP VM must first identify hash map access sites that are suitable, either via straightforward profiling of their key set behavior, or by taking advantage of API semantics (as we describe above). Second, the VM must guarantee that all subsequent access at each access site conform to the same shape, or set of key values. In many cases, this check is rare and/or inexpensive, as outlined above, leading to significant benefit. However, in the most general case, when the set of key values is determined outside the scope of the VM or DBMS interface library (e.g. by reading the key values from an external file, or prompting the user to type them in), the cost of checking the shape of the hash map could equal or even

exceed the cost of relying on a standard hash map. Here, the VM should profile the relative frequency of execution of the sites where the hash map shape is set (and the check must be performed) vs. the sites where the hash maps are accessed (where performance benefit is obtained), and should only enable HMI if the former is less frequent than the latter. Such an evaluation is beyond the scope of this thesis, as we could not find any workloads where this tradeoff has to be made.

3.6 Related Work

This Hash Map Inlining technique touches on topics across a broad spectrum of computer systems related topics, including, but not limited to: database query optimization, effective data structure selection, and type specialization in dynamic languages. We briefly discuss these areas below.

DBMS-specific intra-operator optimizations. There has recently been extensive research on how to specialize the code of a query execution by using an approach called Micro-Specialization [112, 114, 113]. In this line of work, the authors propose a framework to encode DBMS-specific intra-operator optimizations, like unrolling loops and removing unnecessary branching conditions by exploiting invariants present during a query execution. Another recent work is the query execution engine of LegoBase [50] that develops a compilation framework to achieve the same. All these works aim to improve database systems by removing unnecessary abstraction overheads during a query execution. Furthermore, there are many works [65, 72, 76, 86] that aim to speed-up query execution by focusing mostly on improving the way data are processed, rather than individual database

operators. In contrast, our work eliminates the abstraction overheads associated with post-processing the result table of a database query in context of its usage in real-world PHP scripts in a jitted environment. All these software query optimizers and recent works on database hardware accelerators [51, 109] will bring the spotlight on the performance of the post-processing phase and the PHP scripts even more in the future than what our work focuses on.

Effective selection of data structures. There has been plenty of research [43, 44, 84] in optimizing the usage of data structures in applications written in statically-typed languages. Recent work [43, 44] by Jung et al. proposes a program analysis tool that automatically identifies the data structures used within an application and selects an alternative data structure better suited for the particular application input and the underlying architecture. In contrast to that, our work does not seek to identify optimal data structures for real-world PHP applications.

Type specialization in dynamic languages. There is a large body of research in type specialization of dynamic scripting languages. SELF [15, 14] and Smalltalk [17] are the early pioneers. They introduced the inline caching and polymorphic inline caching [37] techniques to specialize a code section with any previously observed types and thus optimize access to dynamically typed objects. We have already summarized inline caching and discussed its adaptation to the hash maps in section 3.2.2 and 3.2.3. There are many recent research proposals in JavaScript specialization [30, 106, 36, 82, 47, 3]. All these works exploit type inference in conjunction with type feedback in different ways to generate efficient native code. One of the most recent works is [3] by Ahn et al. that examines the way the Chrome V8 compiler defines types, and identifies the key design decisions behind

its poor type predictability of class objects in Javascript code from real websites. However all these type specialization techniques do not address the issues with type unpredictability of hash maps in server-side PHP scripts, which our work focuses on.

Note that the adaptation of the well-known inline caching idea to the realm of hash maps is not our primary contribution. Rather, the key contribution here is identifying issues with adaptation of polymorphic inline caching to hash maps in real-world applications and providing enhancements to the JIT engine to mitigate those. Furthermore, there are many recent proposals [68, 4, 2] for providing architectural support to optimize the execution of scripting languages. Mehrara et al. [68], and Anderson et al. [4] propose microarchitectural changes to avoid the runtime checks associated with jitted execution, whereas Agrawal et al. [2] improves the energy efficiency of PHP servers by aligning the execution of similar requests together. They are orthogonal to our compiler modifications and their associated benefits.

Note that the well-known perfect hashing or dynamic perfect hashing techniques [29, 18] provide hash functions that can map keys to a hash map with no collisions. Thus they can avoid any potential overhead from traversing a collision chain in case two keys map to the same entry in a hash map. However they cannot avoid the overheads associated with hash map allocation, release, resizing, hash computation etc., which HMI completely eliminates and gets most of its benefit from.

3.7 Summary

In this chapter, we propose Hash Map Inlining to eliminate the overheads associated with accessing hash maps, the most commonly occurring data structure in scripting languages. This chapter describes compiler enhancements to achieve that for PHP in HHVM compiler. HMI opens up the opportunity of parallelizing HTML generation within a single request (for example, loops in our workloads) and across multiple client requests as [2] envisions. Thus HMI can improve the efficiency of web servers and in turn can directly influence the throughput of data centers.

4 EVALUATION OF HASH MAP INLINING

This chapter evaluates HMI on real hardware and discusses the results. Section 4.1 describes the methodology in detail. Section 4.2 presents the impact on performance of benchmark suites with Extended HMI for SQL (Ext_HMI) implementation and compares it against our initial HMI (Init_HMI) (section 3.2.3) implementation. Section 4.3 investigates the performance bottlenecks in a few benchmark scripts and provides solution to mitigate them. Section 4.4 illustrates the detailed breakdown of execution time of the benchmark scripts. Finally, Section 4.5 summarizes the chapter.

4.1 Methodology

We evaluate our enhancements on five real-world, server-side PHP benchmark suites (Table 4.1). We are showing the results for a subset of scripts that spend noticeable time accessing hash maps. The remaining scripts from each suite spend little or no time accessing hash maps leaving no opportunity for HMI. Note that for the SPECWeb2005 benchmark suite we replace its Besim emulator with an actual SQL server interface in order to account for the overall activity of a PHP script's execution. BeSim emulates a back-end database server that PHP scripts in the SPECWeb2005 suite communicate with to retrieve required database results. We used the latest release of HHVM [1] at the time of this writing with its *Repo.Authoritative* mode turned on for all our evaluations. It activates all the member instruction optimizations present in HHVM. We measure the performance of the benchmarks natively on a 3.6 GHz AMD FX(tm)-8150 eight core machine with 8MB

Table 4.1: Server-side PHP benchmark suites

Benchmark	Description	Hash map-intensive PHP scripts
RUBiS [80]	An auction site prototype modeled after eBay.com, implements core functionality of an auction site: selling, browsing, bidding for different items.	BrowseCategories(BC), BrowseRegions(BR) SearchItemsByCategory(SIC), SearchItemsByRegion(SIR), ViewBidHistory(VBH), ViewItem(VI), ViewUserInfo(VUI), AboutMe(AM)
RUBBoS [79]	An online news forum benchmark modeled after Slashdot.org, provides essential bulletin board features such as featuring news stories (associated comments) on various topics.	BrowseCategories(BC), BrowseStoriesByCategory(BSC), OlderStories(OS), ReviewStories(RS), Search-keyword1(S-K1), Search-keyword2(S-K2), StoriesOfTheDay(SD), SubmitStory(SS), ViewComment-small/expanded(VC-S/E)
Tiger Php News System [96]	Mimics a typical news site such as displaying news with options for browsing, searching for a specific news etc.	Printnews(User)(PU), Printnews(Moderator)(PM), Search(S), PrintComments(PC), GetUserList(GUL), GetMenu(GM), BrowserStat(BS)
SPECWeb2005 [87] (Banking)	Simulates typical requests to an on-line bank	login_welcome(LW), account_summary(AS), bill_pay(BP), bill_pay_status_output(BPS), order_check(OC), transfer(T)
SPECWeb2005 [87] (E-commerce)	Simulates a web store that sells computer systems; allows users to search, browse, customize, and purchase products.	index(I), search(S), browse(B), browse_productline(BP), productdetail(PD), customize(C)

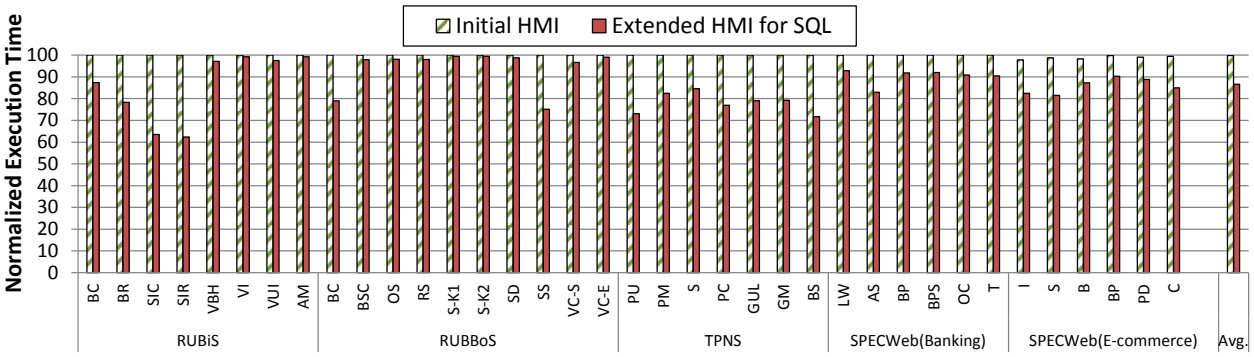


Figure 4.1: Performance improvement with HMI normalized to unmodified HHVM. Averaged across scripts in Table 4.1. Ext_HMI improves SPECWeb banking and e-commerce throughput by 7.71% and 11.71% respectively.

last-level cache, running the 64-bit version of Ubuntu 12.04. The PHP benchmark scripts interact with a MySQL database server installed on the native machine. We used the available test harness to generate client requests. This setup should closely imitate the environment of commercial servers.

4.2 Performance Improvement

Figure 4.1 shows the improvement in performance with Ext_HMI. Init_HMI brings down the average execution time to only 99.8% of the time obtained with unmodified HHVM, whereas Ext_HMI brings down the execution time to 86.63%. This results in throughput[1]

[1]Throughput for SPECWeb is measured using an available test harness that generates requests for all the scripts (6 hash map-intensive scripts from Banking and E-commerce each as shown in Figure 4.1 along with the remaining non-hash map-intensive scripts) from SPECWeb.

improvement of 7.71% and 11.71% for SPECWeb(Banking) and SPECWeb(E-commerce) suites respectively. The *SearchItemsByRegion* script from RUBiS obtains maximum benefit of 37.6% with Ext_HMI implementation. Note that Init_HMI provides marginal benefits for a few scripts such as *index*, *search* and *browse* from SPECWeb(E-commerce) suite. The subset of scripts such as *BuyNow, RegisterItem, RegisterUser, StoreBid* etc. from RUBiS, *PostComment*,*StoreStory* etc. from RUBBoS, *add_payee, change_profile* etc. from SPECWeb(Banking) and *cart, login, shipping* etc. from SPECWeb(E-commerce) suites are omitted since they spend little or no time accessing hash maps. Instead, they spend most of their time executing a database query to retrieve or save a single record; they spend little or no time in PHP scripts, leaving no opportunity for any JIT optimization. Hence, those scripts show no improvements, as expected, so we omit them from our results. However, we note that our modifications did not cause overhead either.

Note that several scripts, such as *SearchItemsByCategory* from RUBiS and *printnews* from TPNS, show substantial improvement with Ext_HMI, whereas for scripts like *ViewItem* from RUBiS, *bill_pay* from SPECWeb(Banking) in Figure 4.1, performance improves modestly. In addition to that, there are a subset of scripts such as *BrowseStoriesByCategory, OlderStories* for which Ext_HMI shows little or no improvement. The reasons behind this uneven improvement in performance will be discussed in the next two subsections.

Note that as the number of fields in class objects changes in Figure 1.1, the branch and cache MPKI (mispredictions or misses per 1000 instructions) with accessing class objects reduce from 6.84 to 3.72 and from 9.31 to 8.56 respectively. However the branch and cache MPKI with accessing hash maps stays around 4.2 and 16 respectively as the absolute number of mispredictions or misses change almost in the same proportion as the number

```
$result=mysql_query("SELECT story_id, writer FROM comments WHERE …");

while ($row = mysql_fetch_array($result)) {
  $user_query=mysql_query("SELECT nickname FROM users WHERE
                          id=$row["writer"]);
  …
}
```

```
$result = mysql_query("SELECT comments.story_id, users.nickname
  FROM comments, users WHERE comments.writer = users.id AND …");

while ($row = mysql_fetch_array($result)) {
  …
}
```

Figure 4.2: Merging nested queries (above) to a single query (bottom).

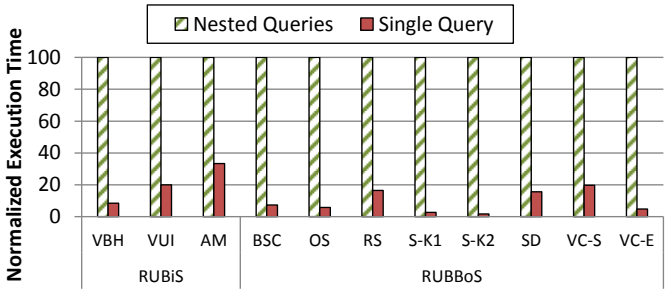


Figure 4.3: Performance gain after merging nested queries into a single query. Only 11 Scripts have nested queries.

of instructions. As we apply HMI to our workloads in Figure 4.1, the branch mispredictions and data cache misses in those workloads are reduced to the level of equivalent C++ objects.

4.3 Nested Queries

Careful examination of the subset of scripts such as *BrowseStoriesByCategory, OlderStories* that do not benefit from our Ext_HMI implementation reveals that they contain nested queries in them that require repeated SQL query invocations for each of the rows extracted from the result tables of parent queries. Not surprisingly, these scripts spend most of their execution time within the SQL query execution function, leaving little or no opportunity for dynamic optimization of PHP code. When a database query is invoked, the DBMS engine begins with building hash tables of the *records* before performing any further operations

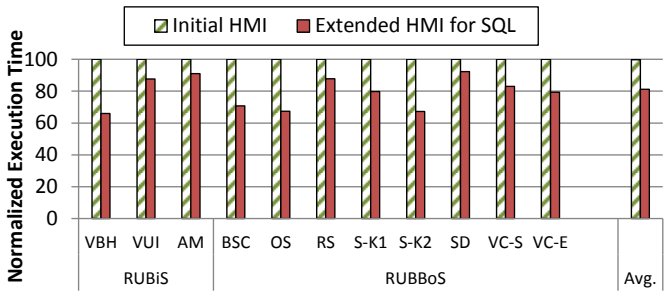


Figure 4.4: HMI applied to scripts after merging their nested queries. Perf. normalized to unmodified HHVM. Avg. shown considers the improvements to 11 modified scripts from Figure 4.3 and remaining 26 unmodified scripts with no nested queries from Figure 4.1 using Ext_HMI.

such as scan, join and sort on them. In the case of nested queries, the child query requires building these expensive hash tables for the entire set of database *records* in order to simply look for a single *record* from them, in spite of the fact that the shape of those hash tables stays the same across invocations. This is an unfortunate yet common performance bug in PHP/SQL scripts, and reflects a lack of experience and expertise in SQL on the part of the programmer.

To better evaluate the impact of HMI on code that has already been re-factored to avoid such basic mistakes, we rewrote these scripts to eliminate the unnecessary nested queries by merging the nested queries in them into a single top-level query, as shown in Figure 4.2. Once the nested queries were merged, these scripts no longer spent virtually all of their time executing redundant SQL, and the performance benefits of the HMI technique were realized for them as well. Figure 4.3 demonstrates the substantial improvement in execution time due to merging the nested queries for the eleven scripts that contain nested queries. As shown, the unmodified scripts spend most of their time in database query execution. Merging the nested queries improves their execution time to just $8-10\%$ of their original execution time.

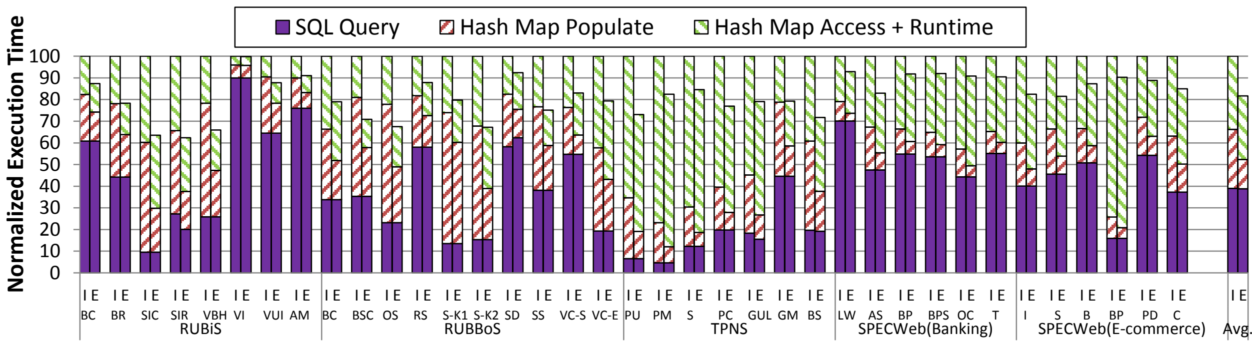


Figure 4.5: Breakdown of execution time normalized to Init_HMI. *I* and *E* refer to Init_HMI and Ext_HMI implementations respectively. Runtime = time consumed in executing string operations, regular expressions, and miscellaneous operations.

Furthermore, after merging nested queries, scripts such as *BrowseStoriesByCategory, Search-keyword* etc. become amenable to the benefits of Ext_HMI (Figure 4.4). So, with applying Ext_HMI to scripts in Figure 4.3 after merging their nested queries, the average execution time across all hash map-intensive scripts comes down to 81.18% in Figure 4.4 from 86.63% obtained before merging nested queries in Figure 4.1. Note that $10-12\%$ of the overhead in populating hash maps comes from resizing and consequently only 2.6% performance benefit, instead of the 18.81% shown, can be obtained with optimized initial sizing.

4.4 Breakdown of Execution Time

Figure 4.5 shows the breakdown of execution time for the benchmark scripts. *mysql_query* represents the time taken in executing the SQL queries. *Hash Map Populate* and *Hash Map Access* denote the times consumed in populating and accessing hash maps in those scripts. We observe that the majority of the improvement in execution time comes from the lower overhead in populating vector-like arrays inside the DBMS library function enabled by our

Ext_HMI. This breakdown essentially validates our initial motivation and confirms the fact that hash map processing consumes a significant portion of the overall execution time. Scripts that require retrieval of many rows (and hence populate hash maps many times) from the query result table observe major improvement in execution time. Scripts that are showing only marginal improvement require retrieval of the fewer rows. The *ViewItem* script retrieves only 2 rows and hence does not find any improvement with Ext_HMI. Scripts from the SPECWeb(Banking) suite retrieve few rows (5 rows on average) and hence observe marginal benefit with Ext_HMI. On the other hand, *SearchItemsByCategory*-like scripts pull out hundreds of rows from the result table and show a substantial improvement. All the scripts observe expected improvement (5.09% on average from Figure 4.5) from inlined lookups to the populated hash maps.

4.5 Summary

This chapter presents an evaluation of the HMI technique. Results show that HMI delivers performance benefits up to 37.6% and averaging 18.81% over a set of hash map-intensive server-side PHP scripts. This results in modest throughput improvement for SPECWeb banking and e-commerce workloads.

5 ARCHITECTURAL SUPPORT FOR SERVER-SIDE PHP PROCESSING

Server-side, content-rich PHP applications occupy a huge footprint in the world of web development. This chapter performs an in-depth analysis of such large-scale, content-rich PHP applications and demonstrates potential for specialized hardware to accelerate these PHP applications in a future server SoC. Section 5.1 provides an overview of our proposed specialized hardware . The remainder of this chapter is organized as follows. Section 5.2 performs an in-depth microarchitectural analysis of these content-rich 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 5.3. In Section 5.4 we design specialized hardware accelerators for them. Section 5.5 discusses related work and Section 5.6 summarizes the chapter.

5.1 Overview

PHP is the dominant scripting language used for server-side web development in today's programming world. In particular, server-side, content-rich PHP web applications have created an ecosystem of their own in the world of web development. PHP powers many of the most popular web applications that run on live datacenters. Despite its considerable increase in popularity, its performance is still the main impediment for building large, CPU-intensive applications.

The desire to reduce datacenter load inspired the design of the HipHop JIT compiler (HHVM) [115, 1], which translates PHP to native code. HHVM demonstrates a significant

performance increase for a set of server-side PHP applications with the help of our Hash Map Inlining technique and other state-of-the-art code optimization techniques. However, runtime characteristics of an important class of large-scale, server-side PHP web applications that spend most of their time in rendering web pages are found to be dramatically different than the de-facto benchmark suites SPECWeb2005 [87], bench.php [111], and the computer language benchmarks [93] 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 [2]. Furthermore, these micro-benchmarks spend most of their time in JIT-generated compiled code, contrary to the real-world, content-rich PHP applications that tend to spend most of their time in various library routines of the VM.

As shown in Figure 1.2 of Chapter 1, the SPECWeb2005 workloads contain significant hotspots in their distribution of CPU cycles, with very few functions responsible for about 90% of their execution time. However, the realistic, content-rich PHP applications instead exhibit very flat execution profiles, having significant diversity – the hottest single function (JIT compiled code) is responsible for only $10-12\%$ of cycles, and they require 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 of these large-scale, content-rich PHP applications, we performed an in-depth architectural characterization of them and found that there is little opportunity for traditional microarchitectural enhancements.

As the processor industry continues to lean towards customization and heterogeneity [101, 35, 57, 88] to improve performance and energy efficiency, we seek to embrace domain-specific specializations for these content-rich PHP applications. We note that

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 [4], hash table accesses for user-defined types [15, 14], etc.) or the associated overhead of garbage collected languages [8]. These observations guide us to apply several hardware and software optimization techniques from prior works [4, 15, 14, 33] *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 content-rich 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 [15, 14, 33]. 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 patterns of these PHP applications, contrary to prior works deploying a hash table that supports only GET requests in a memcached environment [58]. 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 comes from memory allocation and deallocation, despite significant efforts to optimize them in software [27, 54]. 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 [103] 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 state-of-the-art software with SSE-optimized implementation.

Regular Expressions. These PHP applications also use regular expressions (regexps) to

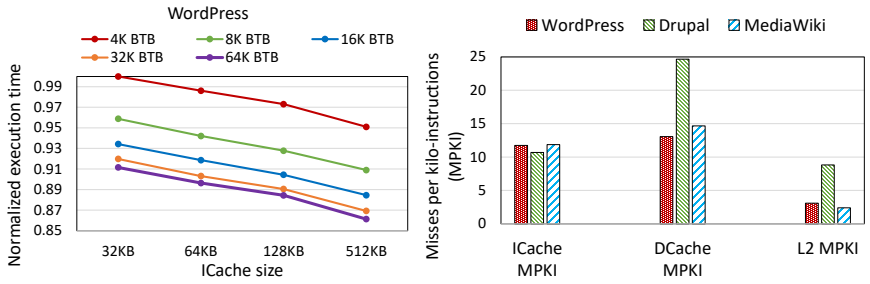
dynamically generate HTML content from the unstructured textual data described above. Software-based regexp engines [73, 60] or recent hardware regexp accelerators [64, 23, 91, 32] are overly generic in nature for these PHP applications as they do not take into consideration the inherent characteristics 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.

5.2 Microarchitectural Analysis

We begin by performing an in-depth architectural characterization of the content-rich PHP applications to identify performance and energy-efficiency bottlenecks (if any) in them. We use gem5 [94] 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 instruction or data caches show significant opportunity for improving performance.

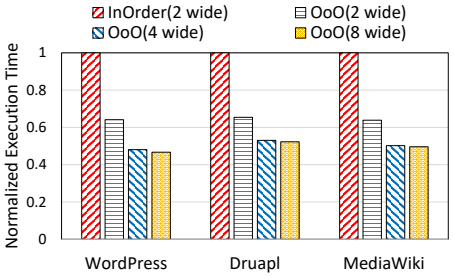
Branch predictor bottlenecks. We experimented with the state-of-the-art TAGE branch predictor [83][1] with 32KB storage budget. The branch mispredictions per kilo-instructions (MPKI) for the three content-rich PHP applications considered in this dissertation are 17.26, 14.48, and 15.14. Compared to the 2.9 MPKI average for the SPEC CPU2006 benchmarks,

[1]The branch prediction accuracy observed on Intel server processors and TAGE are in the same range [78]



(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 5.1: Microarchitectural characterization of the content-rich PHP applications.

these applications clearly suffer from high misprediction rates. The poor predictor performance is primarily due to the presence of a 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 process during most of their execution time. Prior work on predicting data-dependent branches [24] may improve the MPKI of the PHP applications.

Branch target buffer bottlenecks. Large-scale, content-rich 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 [116], whereas in the PHP applications about 22% of all instructions are branches, thus adding more pressure on BTB. Figure 5.1(a) shows the execution time of one of the content-rich 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 5.1(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 [116] or memcached workloads [58]). 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 most of the hot leaf functions 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 5.1(a) shows the potential of minor performance gain with very large instruction caches.

In-order vs. out-of-order. Figure 5.1c 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 content-rich 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.

5.3 Mitigating PHP 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 [4, 15, 14, 33] 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 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 [15, 14] and Hash Map Inlining [33]. Modern JIT compilers [1, 100] 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 [33]. Further, we adopt the recent proposal on *hash map inlining* [33] (HMI) (see Chapter 3) 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 [4] to mitigate this overhead in hardware. With this technique, the cache subsystem performs the required type check for a variable before returning its value (see Chapter 2).

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.

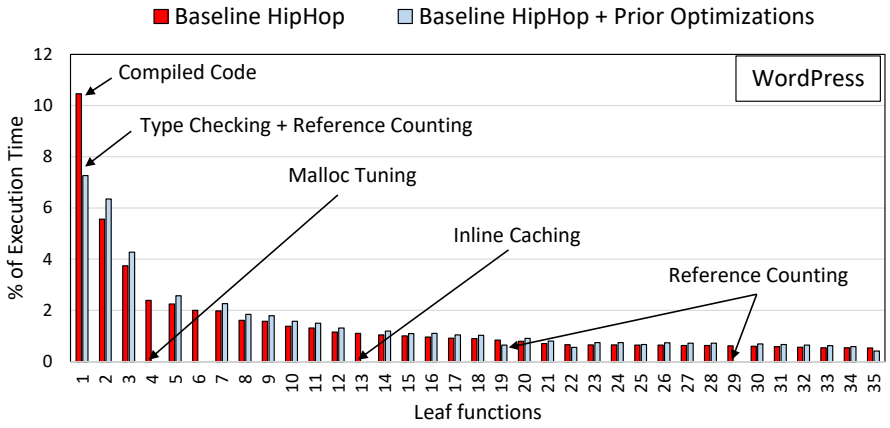


Figure 5.2: Contribution of leaf functions to the execution time of WordPress before and after applying all optimizations.

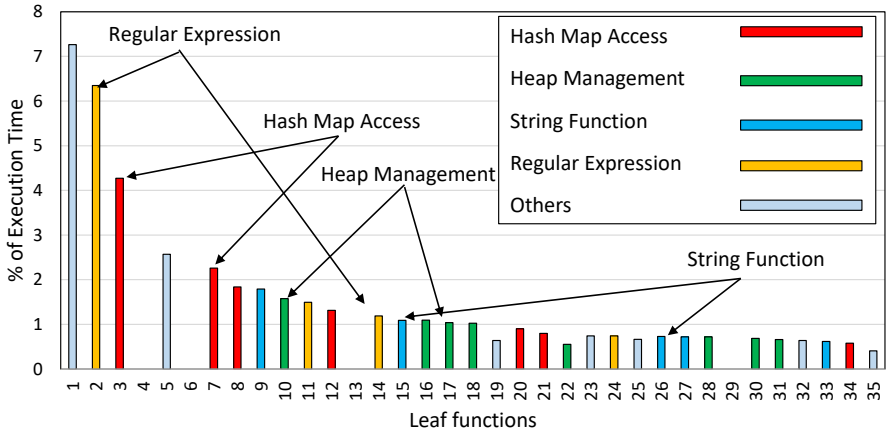
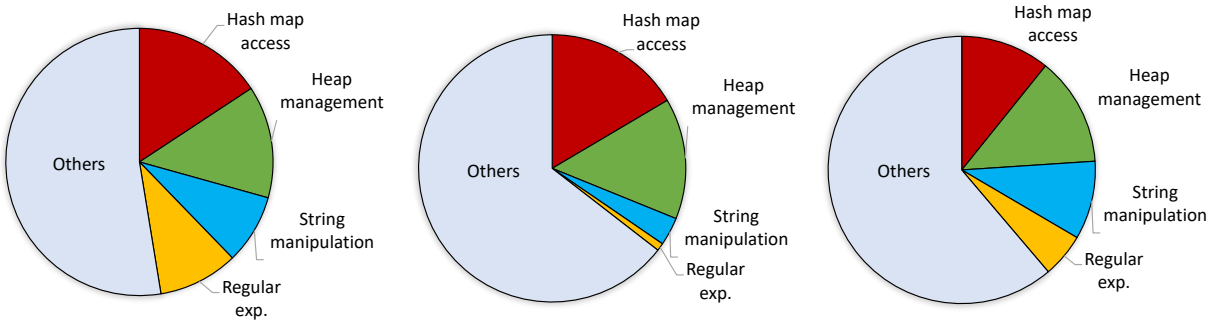


Figure 5.3: Categorization of leaf functions of WordPress into major categories.

We adopted a hardware proposal from prior work that introduces minimal changes to the cache subsystem to mitigate this overhead [42] (see Chapter 2).

In addition to applying the above four optimizations, we tuned the software heap manager of the baseline HHVM infrastructure to reduce the overhead from expensive memory allocation and deallocation calls to the kernel in these PHP applications.

Figure 5.2 demonstrates the effect of applying these above optimizations to the leaf functions of one of the PHP applications, WordPress [107]. The left bar shows the contribution of the leaf functions to the overall execution time before applying the optimizations,



(a) WordPress (b) Drupal (c) MediaWiki

Figure 5.4: Execution time breakdown after mitigating the abstraction overheads.

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 5.3. This consequently presents opportunities to accelerate them in hardware to obtain performance and energy efficiency. Figure 5.4 shows the execution time breakdown of a few content-rich 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.

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

5.4 Specializing the General-Purpose Core

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

5.4.1 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 [85]. 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.

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.

5.4.2 Hash Table

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

Overview. Content-rich PHP applications frequently access hash maps with dynamic key names; such accesses cannot be converted to efficient offset references by software methods [15, 14, 33]. 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 5.5 shows the hash table accelerator design. A critical

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

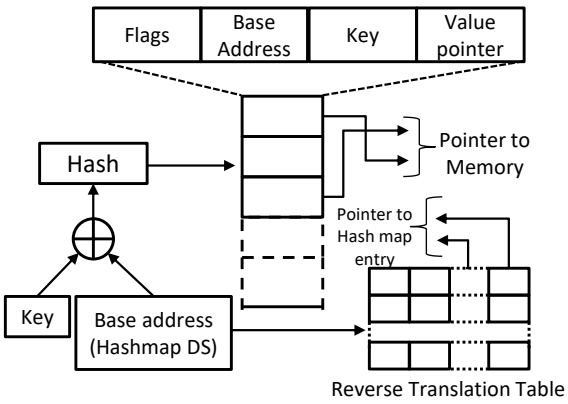


Figure 5.5: Hardware hash table.

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 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

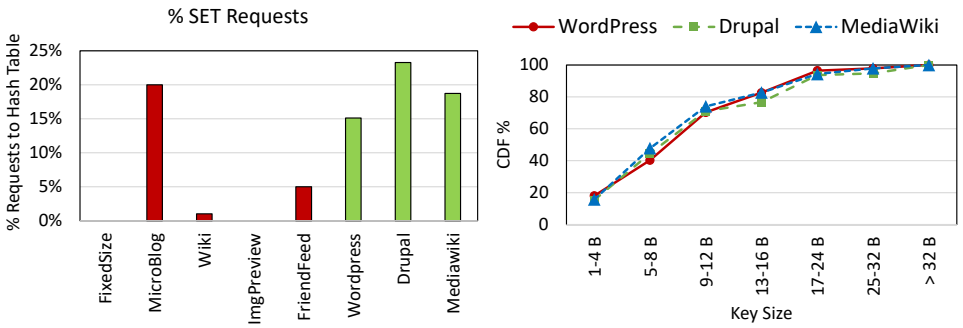
Table 5.1: Table comparing the hash table hit rate of the simplified hash function against the original HHVM implementation.

	Hash table size (#entries and associativity)							
Hash Function	1(1)	4(1)	16(1)	64(1)	128(4)	256(4)	512(4)	1024(4)
HHVM (WordPress)	28.25	46.53	60.54	72.31	80.99	85.29	88.1	89.89
Simplified (WordPress)	28.25	44.08	61.52	71.9	80.68	84.6	87.59	89.39
HHVM (Drupal)	36.19	54.3	66.10	74.59	80.14	82.2	84.06	85.99
Simplified (Drupal)	36.19	53.13	66.70	73.44	79.9	81.96	83.77	85.72
HHVM (MediaWiki)	27.7	41.34	55.73	67.76	75.53	81.49	87.17	89.48
Simplified (MediaWiki)	27.7	40.96	56.02	64.22	75.42	81.64	86.34	88.68

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 [1] 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. Table 5.1 compares the hit rate of the simplified hash function against the original HHVM implementation and demonstrates negligible loss in performance. Our design leverages several inherent characteristics of these PHP applications.

First, in contrast to most large-scale memcached deployments [5, 58] where GET requests vastly outnumber SET and other request types, these PHP applications observe a relatively higher percentage of SET requests (ranging from $15-25\%$) when generating



(a) Comparison of the percentage of SET requests.

(b) Cumulative distribution of hash table key sizes.

Figure 5.6: Few characteristics of the PHP applications.

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. Figure 5.6a compares the percentage of SETs from the memcached workloads [58] against the PHP applications. The first five workloads are memcached workloads. The percentage of SETs is negligible for all but one of the memcached workloads whereas the PHP applications observe high percentage of SETs ranging from $15-25\%$.

Second, the majority (about 95%) of the hash map keys accessed in these PHP applications are at most 24 bytes in length. Figure 5.6b shows the cumulative distribution of key sizes. As a result, we store the keys in the hash table itself, unlike the hash table designed for memcached deployments [58]. 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

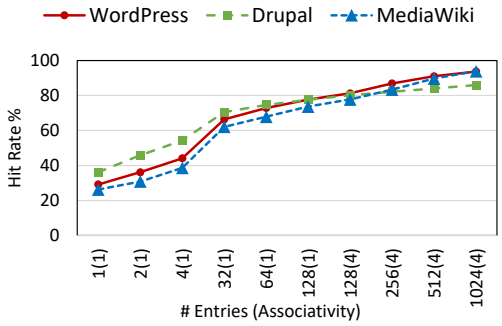


Figure 5.7: Hash table hit rate.

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 5.7 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 5.5 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.

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 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.

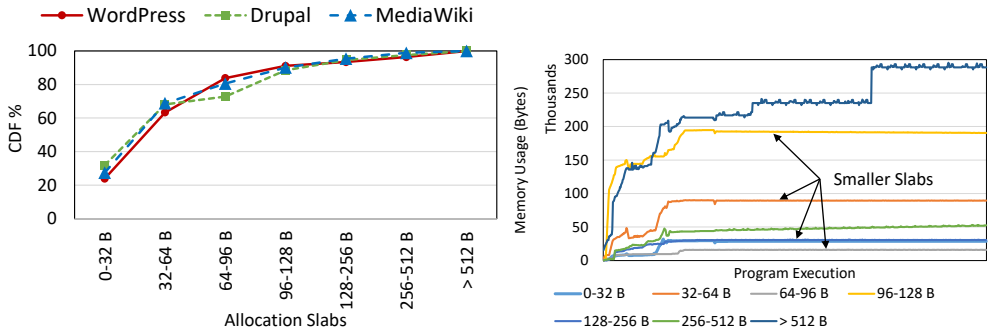
5.4.3 Heap Manager

Memory allocations and deallocations are ubiquitous in content-rich 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 singly-linked free list – list with addresses to free chunks of memory of the same size.

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.

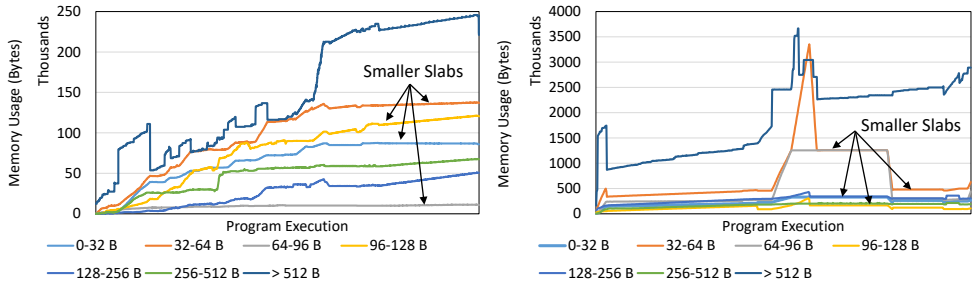
First, a majority of the allocation and deallocation requests retrieve at most 128 bytes, which reflects heavy usage of small memory objects. Figure 5.8a shows the cumulative distribution of memory usage with different memory allocation slabs.

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 is recycled. (b) These applications typically process



(a) Cumulative memory usage with different allocation slabs.

(b) Memory usage of WordPress.



(c) Memory usage of Drupal.

(d) Memory usage of MediaWiki.

Figure 5.8: Memory usage pattern of the PHP applications.

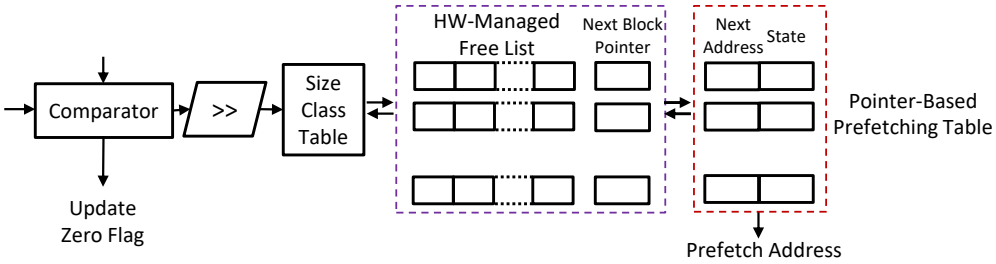


Figure 5.9: Hardware heap manager.

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 5.8b, Figure 5.8c, and Figure 5.8d show memory usage pattern of three workloads during the course of their execution. We see that memory usage mostly 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.

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 5.9 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 hardware-managed 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. Upon finding a miss in the hardware free list, control transfers to software to satisfy the memory allocation request from the software heap manager. The hardware-managed free list for each size class has head and tail pointers to orchestrate allocation and deallocation of memory blocks. Each free list has a next block pointer also, the requirement of it is discussed later. Memory deallocation follows a similar path. When an allocated memory block is being freed, the size class table identifies an appropriate free list for the freed object and pushes the object to the top of the free list. If adding the deallocated memory block overflows the maximum number of entries for the given size class, the software handler returns the evicted block back to the software heap manager.

At the start of a program, upon finding a miss for an available memory block in the hardware-managed free list, the software handler retrieves a memory block from the software heap manager and returns it to the caller. Additionally, it stores a copy of the current head pointer (associated with the free list of the software heap manager it retrieved the current memory block from) in the next block pointer field of the corresponding hardware-managed free list. A prefetcher retrieves available memory blocks from the

software heap manager and keeps the hardware-managed free lists populated with them 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. Since modern memory allocators provide the location of the next available block when dereferencing the current block pointer[4], the pointer-based prefetcher uses the value returned by the current load to determine the next iteration's prefetch address. The core uses the size-class head pointer for push and pop requests to/from a hardware-managed free list, and the prefetcher pushes to the location of the tail pointer.

When a hardware free list is empty, the next block pointer field (populated by the software handler upon finding a miss in the hardware heap manager accelerator) helps the prefetcher to initiate prefetching available blocks from the software heap manager. As the prefetcher prefetches blocks to the heap manager accelerator, it updates the next block pointer field of the associated hardware-managed free list so that the field always points to next available block of the corresponding free list in the software heap manager. Later, upon finding a miss for an available block in the heap manager accelerator, the software handler can consult this next block pointer field (if it is non-empty) to retrieve blocks from memory. As long as the heap manager accelerator can satisfy a memory allocation or a deallocation request, the head pointer of a free list in the software heap manager can stay stale until an overflow from the hardware heap manager table or a context switch occurs

[4]The software free lists of modern memory allocators (TCMalloc [31] and jemalloc [22]) store the location of the next available block at the address of the current memory block it is about to return, instead of allocating a separate field in a `struct` for it. In addition to saving memory taken up by the free lists, dereferencing the current head to get the next pointer has the side effect of prefetching the returned memory block itself, which can likely help the caller.

or another process attempts to "steal" memory from the current process. So in summary, upon finding a miss if the software handler finds the next block pointer field in the heap manager accelerator as non-empty, it considers that as the up-to-date head pointer of the associated free list of the memory-resident software heap manager structure and uses that to retrieve a block from memory. Otherwise it relies on the head pointer field of the software heap manager to retrieve blocks from memory.

When a hardware-managed free list in the heap manager accelerator overflows, the software handler returns the evicted block back to the software heap manager and updates the corresponding next block pointer field in the heap manager accelerator. In this case, the software handler essentially updates the content of the second-to-last block (which becomes the last block for the given free list post-eviction) in memory to point to the evicted block (which can be done using a single `str` instruction). This essentially preserves the linked list invariant of a software-managed free list in the heap manager data structure, where the free list stores the location of the next available block at the address of the current memory block it is going to return. Note that a memory allocation request from the software handler may race with an inflight prefetch request to the same size class. The heap manager accelerator keeps track of such prefetch requests. In the case when the response to such a prefetch request reaches the heap manager accelerator, it is ignored to maintain consistency with the software heap manager data structure.

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, frequent memory reuse

means that in the common case it satisfies the requests from the hardware-managed free list, resulting in very infrequent fall-backs to the software routine that handles any complex cases. Concurrent research work on accelerating memory allocation [45] eagerly updates head pointer and linked list of software heap manager on all malloc and free requests to keep coherency of hardware heap manager accelerator with software heap manager. However, due to the high memory reuse of these workloads, we instead lazily update the software heap manager data structure only on overflow or during context switches. This avoids the overheads of constantly updating memory to be coherent with the heap manager accelerator during periods that the heap manager accelerator is servicing the common case requests, while not causing any correctness errors or memory leaks.

Ideally, a large number of entries in a hardware-managed free list should help the heap manager accelerator in the following ways. First, it will result in fewer prefetch requests and overflows from the hardware free lists. This is especially true for applications that allocate and deallocate at similar rates, since their memory requests can mostly be satisfied from the hardware free lists without any software involvement. Second, If an application generates bursts of memory allocation or deallocation requests in a short span of time, having more entries in a hardware free list will allow the heap manager accelerator to service the requests mostly from the free list. This is due to the fact that a wider free list will provide the prefetcher enough time to prefetch available blocks from the memory-resident software heap manager data structure while free blocks are being meanwhile served to the bursty application from the head of the list.

At context switches, a software handler can flush the entries from the hardware-managed free lists and reconstruct the free lists (while meeting the linked list invariant)

of the software heap manager in memory using a series of `str` instructions[5]. However performing this in software will require many `str` instructions to execute and thus will incur many processor cycles. Instead, a hardware state machine in our design performs this seemingly expensive operation and flushes the hardware-managed free list entries to memory while meeting the necessary linked list invariant of them. Once the hardware heap manager accelerator is flushed, the head pointers (possibly stale) of the software-managed free lists in the heap manager data structure are updated to point to the available memory blocks.

Each hardware-managed free list has a corresponding prefetcher entry in the Pointer-Based Prefetching Table (PPT) in our design (Figure 5.9). The PPT table contains fields for the next address to be accessed, and state of the given PPT entry. For a given free list the prefetcher entry can be in one of two primary states: *untrained*, or *trained*. Initially *untrained*, upon satisfying a miss by the software handler, the address recorded in the next block pointer field of a hardware free list is copied to the associated prefetcher entry and the state transitions to *trained*. Once a prefetcher entry transitions to the *trained* state, it is allowed to prefetch memory blocks to its associated hardware free list without restriction up to a *high watermark*. The *high watermark* is enforced on a per free list basis and limits the number of entries in the hardware list that can be occupied at any given time with prefetched blocks. Upon satisfying memory allocation requests from a hardware list, whenever the number of available blocks in the list drops below the *high watermark*, the prefetcher uses the next address field to initiate prefetching blocks from memory. The *high watermark* threshold is

[5]A `str` instruction here updates the contents of the flushed blocks in a similar way as it does in case of an overflow from the hardware heap manager table.

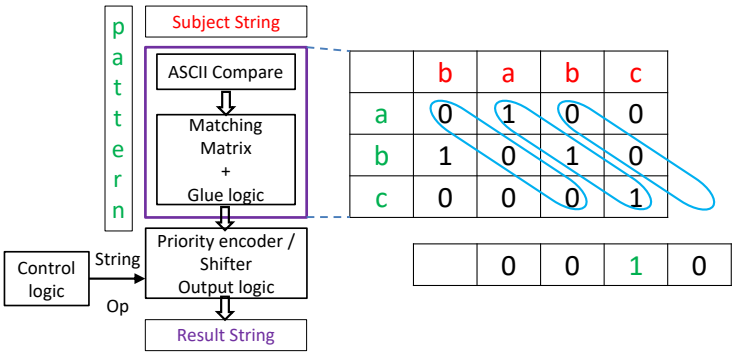


Figure 5.10: Block diagram of string accelerator with string_find example searching for 'abc' in a subject string 'babc'.

set to half the size of a free list in our design. This ensures that future memory deallocation requests can push freed blocks to the top of a free list without excessively overflowing it.

5.4.4 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 [103], 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 5.10 shows a high-level block diagram of our string accelerator. It uses combinational logic to find the presence of pattern characters within the subject string to populate a matrix (matching matrix in Figure 5.10). The matching matrix in turn makes use of a glue logic to perform a string operation. For example, a string operation that attempts to find the presence of a given pattern withing a subject string and thus requires matching of multiple characters of the pattern uses AND gates of diagonal entries within the matrix to find the position of consecutive character matches. Figure 5.10 shows the subject string 'babc' doing a *string_find* for 'abc.'

In the next paragraphs, we will present our accelerator with the example of six PHP string manipulation functions, and explain our choice of these functions.

- *memchr* - Gives the position of the first instance of a character within a string.
- *string_find* - Gives the position of the first instance of a sub-string within a string.
- *string_to_lower* - Replaces all upper-case characters to lower-case characters.
- *string_translate* - Replaces all instances of character(s) with given character(s) within a string.

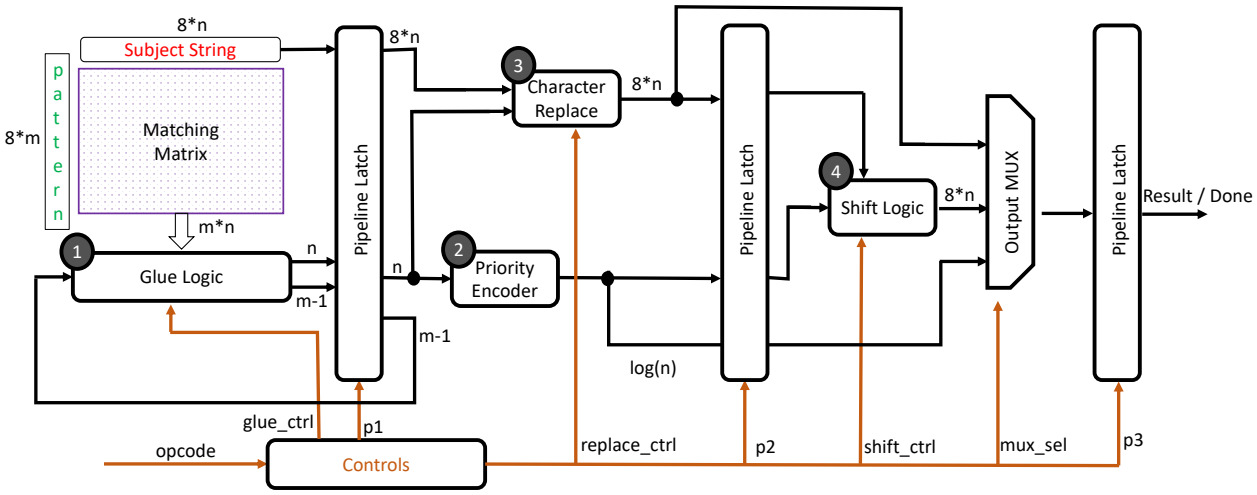


Figure 5.11: Datapath of the string accelerator with control signals. The string accelerator is either a 2- or 3-stage pipeline based on the given string operation.

- *string_replace* - Replaces all instances of a matching sub-string with a given sub-string within a string.

- *string_trim* - Removes all consecutive instances of specified character(s) (typically whitespace) from the beginning and end of a string.

Datapath and control. Figure 5.11 shows the more detailed datapath of the subblocks and control used within our accelerator design. The numbers following each subblock name corresponds with their number as shown in Figure 5.11. n and m correspond to the input string width (in bytes) and number of pattern characters (in bytes), respectively. The datapath numbers from Figure 5.11 are measured in number of bits.

ASCII compare (Matching Matrix) 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. The matching matrix is populated either by <, >, or

= comparisons set by the accelerator configuration. This component is used in all string manipulation functions.

Glue Logic (1) uses combinational logic to populate the matching bitvector based on the operation provided. The term "matching bitvector" is synonymous to the output of the glue logic block of width n. For single character matching such as *memchr*, the matching bitvector is simply the matching matrix's vector for the character's row. Operations that require multiple conditions being met (such as *string_to_lower*) calculate the matching bitvector by using an AND gate over multiple matching matrix rows (for example, >'A'-1 and <'Z'+1) for the same input character (vertical AND). Operations may look for a match of any number of characters (as in *string_trim*), and use a logical OR for the same input character (vertical OR) to populate the matching bitvector. Operations that require matching of consecutive characters use AND gates of diagonal entries within the matching matrix to find the position of consecutive character matches. For example, Figure 5.10 shows the subject string 'babc' doing a *string_find* for 'abc.' In order to support wrap-around, a partial-match bitvector of size $m-1$ is used the next cycle in order to calculate the full matching bitvector.

The *Priority Encoder (2)* is used for string operations that require index calculation of pattern/character matches. It takes the output from (1) (the matching bitvector) as its input, and outputs the binary encoding of the position of the first logical '1'. There is also a valid bit for when there is a hit (the input bitvector is non-zero). For the *string_trim* operation, it also has the capability of outputting the reverse priority encoding.

Character Replace (3) is used for string operations that require modification of the input string. It takes the matching bitvector and input string as input, and replaces matching

input characters with the replacement character(s) as designated by the string function opcode and source operands. If a *string_replace* operation has a replacement string larger than the pattern string, a buffer contains the overflow characters (if any) to output the following cycle.

Shift Logic (4) aligns the subject string to the correct address offset for string manipulation that requires (re-)writing a resulting string to memory. It is also used for aligning characters to the correct position for operations such as *string_trim* (where the positional shifting is dependent on the output of (3)). The shifting logic block also contains output bitmasks.

The datapath is organized in such a way that control logic determines the correct combination of subblocks enabled based on the incoming string operation. Our string accelerator engine supports several PHP string functions. However, all possible datapath flows of these functions can be expressed with the above-mentioned six PHP string functions. We chose these six functions because all additional string functions are slight variations from these six operations, and can be mapped to the hardware using the same datapath structure. Below we discuss how these six representative string functions use the different subblocks of our string accelerator to perform their respective operations. Note that any block not used by a function can be completely power-gated, since it is purely based on the string function opcode. Multi-byte character sets (Unicode) can be handled by grouping the single-byte characters comparisons in the simplified ASCII example shown. For the six string functions, the (#) items refer to the hardware blocks that they use in Figure 5.11. For example, "*string_trim* - (1), (2), and (4)" means that function *string_trim* uses the glue logic, priority encoder, and shifting logic subblocks.

- *memchr* - (1) and (2). The glue logic control is set to directly copy the matching matrix pattern row to be the matching bitvector. This is fed into the priority encoder, its result being the final output. Pipeline stages p1 and p3 are clocked. The output MUX selects the bypassed PE output.

- *string_find* - (1) and (2). Same as memchr, except the glue logic control is set to use AND gates of diagonal entries to calculate the matching bitvector and partial-matching bitvector. Note that partial match bitvectors at the end of the input string are fed back into the glue logic to find potential matches next cycle.

- *string_to_lower* - (1) and (3). The glue logic uses vertical AND gates (satisfying multiple conditions) to populate the matching bitvector for characters that are lowercase. All bitvector hits replace their character with its complimentary uppercase character. Pipeline stages p1 and p3 are clocked. The output MUX selects the bypassed (3) output.

- *string_translate* - (1) and (3). This operation is slightly more complicated than the other operations listed from a datapath perspective, since the position of replaced characters is dynamic. The matching matrix output is passed to the second stage, which means it has a larger glue logic output. From there, a crossbar is needed to route the correct character to the correct bitvector matches. (3) uses the output of the crossbar to replace the correct characters. Pipeline stages p1 and p3 are clocked. The output MUX selects the bypassed (3) output.

- *string_replace* - (1), (3), and (4). The glue logic uses diagonal AND gates to calculate

the matching bitvector and partial-matching bitvector. Positive matches (of the full matching bitvector) are replaced by the given substring from the operation. Note that the order of replaced characters is known by the function call, and the position is known by a single bitvector, so the crossbar logic from *string_translate* is not needed. Ideally, the character replacement would occur in the second cycle. In order to support wrap-around, if the partial bitvector becomes a positive match the following cycle, the character replacement occurs on the third cycle, and the input string is buffered. Any overflow characters are placed into a buffer to output next cycle. Shift logic aligns the output string to write back to memory. Pipeline stages p1, p2, and p3 are clocked. The output MUX selects the output of the shift logic.

- *string_trim* - (1), (2), and (4). The glue logic looks to see if the input string matches any of the function trimmed characters. This is done through a vertical OR operation of the characters of interest. However, since the function looks for the first NON-match, we invert this result so the priority encoder does not have to be modified from other string functions. The priority encoder finds the first (and/or last) non-matching character for the input string. The input string can be directly fed into the shift logic without using any of the character replace (3) logic. The shift logic uses these results to align and mask the final output string accordingly. Pipeline stages p1, p2, and p3 are clocked. The output MUX selects the output of the shift logic.

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.[6] 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 a partial-matching bitvector from the glue logic output and feeding them back into the next-cycle glue logic sub-block. For example, in Figure 5.10, the right-most diagonal AND of 2 elements would be the first bit in the partial-matching bitvector. 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.

[6] I thank my lab colleague David Schlais for his help with designing and implementing the string accelerator in Verilog-like hardware description language.

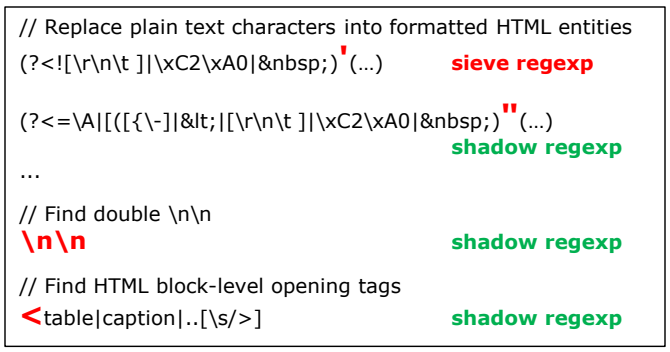


Figure 5.12: Code snippet from WordPress. All four regexps look for special characters – apostrophe, double quote, newline character and opening angle bracket (highlighted in red).

5.4.5 Regular Expression Accelerator

Traditional regular expression (regexp) processing engines [73] are built around a character-at-a-time sequential processing model that introduces high microarchitectural costs. Straightforward parallelization to multi-character inputs leads to exponential growth in the state space [40]. Recent software-based solutions [9, 60, 81] use SIMD parallelism to mitigate that and accelerate regexp processing, but they fail to exploit the regexp-intrinsic characteristics in content-rich PHP applications. On the other hand, the high hardware cost associated with parallel regexp accelerators [64, 91, 32] may deter their commercial deployment in the near future. Instead, we exploit the inherent characteristics of the regexps in the content-rich 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 large-scale, content-rich PHP applications.

Content Sifting. These content-rich 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 5.12 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[7]. 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

[7]The first two regexps look for an apostrophe or double quote before they *look behind* [97] to find a match for the first segment inside the parenthesis.

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, $(\texttt{SEGMENT\_SIZE} - \texttt{inserted\_text.length}())$ 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 content-rich 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 the same content for the regexps that process them.

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

Figure 5.13: Hardware content reuse table.

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, which the content sifting technique can not. We use a reuse table to capture this opportunity (Figure 5.13). 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

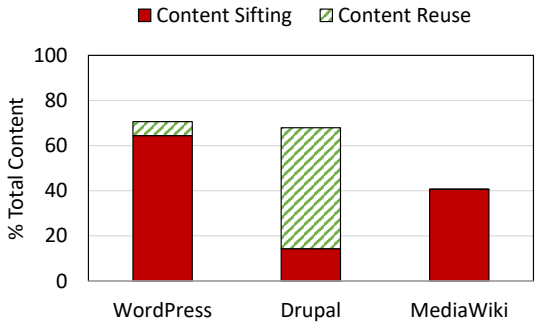


Figure 5.14: Opportunity with content sifting and content reuse. y-axis shows the percentage of total textual content in the entire application regexps can skip processing using content sifting or content reuse.

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 5.13 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 5.14 shows the percentage of textual content that regexps can avoid processing using either content sifting or content reuse.

5.4.6 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 and requires involvement of the software handler to return the evicted block back to the software heap manager. At context switches, the hardware heap manager must flush its entries to the software heap manager data structure. We do this by adding the instruction `hmflush`. `hmflush` invokes a hardware state machine (as discussed in Section 5.4.3) in our heap manager accelerator to perform the flush. `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[8], 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 also used to reinitialize the string accelerator after a context switch.

To perform any regular expression matching, we replace the Perl Compatible Regular

[8]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 invoked at the start of the program and after context switches.

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 5.4.5.

5.5 Related Work

This chapter touches on topics across a broad spectrum of computer systems related topics, including, but not limited to: server core design, scripting language optimizations, and domain-specific accelerators. We briefly discuss these areas below.

Server core design. In recent years, numerous research efforts have been devoted to optimizing warehouse-scale(WSC) and big data workloads [46, 59, 58, 26, 6, 104] developed in C++-like compiled languages. [46] has demonstrated in-depth microarchitectural characterization of WSC workloads and provided several possible directions for architects to accelerate them. Specialized interconnects [62] and customized hardware accelerators [75] have been developed to optimize datacenters. Recently, distributed in-memory key-value stores, such as memcached, have become a critical data serving layer for large scale Internet-

oriented datacenters. [58, 55] have identified microarchitectural inefficiencies with running memcached workloads and proposed specialized hardware to mitigate them. Further, there have been multiple recent efforts to address instruction cache bottlenecks [52, 41] in datacenter workloads. [52] exploits the return (return from function calls) address history to design much accurate instruction cache prefetchers, whereas [41] modifies cache replacement policies to mitigate overheads from cache misses.

However, the software community is increasingly leaning towards scripting languages due to their high programmer productivity [10]. 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 [116] concentrated on server-side Javascript applications. However, PHP is most commonly used [115, 105], representing over 80% [99] of all web applications. In this context, our work is the first to present a comprehensive analysis of the microarchitectural bottlenecks of large-scale, content-rich, 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 [17, 15, 14, 37, 30, 106, 36, 82, 47, 3, 33] attempt to mitigate abstraction overheads (see Section 5.3) 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 [99]. [68, 4] propose microarchitectural changes to avoid runtime checks in jitted execution of Javascript programs. [10] demonstrates the potential of asymmetric multi-

processors 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 [12], pre-execution techniques [11] and modifying cache insertion policy [116] to mitigate this. However, surprisingly the content-rich 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[2]. [2] improves the energy efficiency of SPECWeb2005 workloads by aligning the execution of similar requests together. Instead our work focuses on large-scale, content-rich 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 [58]. Furthermore, [13] 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 [46]. 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.

Accelerating regular expression processing. Regular expression matches are traditionally found using deterministic finite automaton (DFAs) or non-deterministic finite automaton (NFAs). DFAs use finite state machines and transition to different states based on the input character and current state pairs. It is considered deterministic because the output state for a given state/input-character combination will always be the same. However, the memory footprint of DFAs can be extremely large due to the "state explosion" or "exponential blowup" in creating a deterministic output for each state. Alternatively, NFAs allow multiple state transitions per input, and therefore are non-deterministic. In other words, the NFA can simultaneously be in multiple states at the same time. NFAs solve the exponential state space problem of DFAs, but is harder to implement the non-deterministic nature, and can also be slower to reach the desired output. Both of these methods traditionally process a single character at a time. If attempting to process multiple characters at once, the number of transitions grows exponentially when processing multiple characters per cycle. More specifically,the number of possible transitions is α^n, where α is the number of characters in the automaton language, and n is the number of characters processed per byte. For regular expression matching of typical text (natural language), this becomes quickly infeasible to implement. However, due to the increasing demand to keep improving performance, researchers have attempted to create accelerators to process multiple bytes per cycle without these state explosions or exponential transitions.

In 2012, IBM released the RegX accelerator [64], a regular expression accelerator using

multiple parallel Balanced Routing Table based Finite State Machines (B-FSMs). It allows separate patterns to update a general-purpose register for partial pattern matches. RegX handles the exponential DFA state explosion problem by splitting multiple patterns to different B-FSM machines and combined in a later reduction stage. It maintains high pattern matching scanning rates even when searching thousands of patterns. In 2016, Tandon et al. presented HAWK [91], a hardware accelerator for pattern matching. HAWK is based on the Aho-Corasick algorithm to search for multiple regexp patterns simultaneously. It captures concurrency by processing multiple characters per byte while using a bit-split automata [90] to reduce transitions per state and number of states. In order to process multiple bytes per cycle, it searches for all potential alignments of the designated pattern for the input text. The patterns were limited to exact string matches of characters and fixed-length wildcards. HAWK was later expanded to HARE (Hardware Accelerator for Regular Expressions) [32] to also support character classes and partially support Kleene operators (+,*). A character class unit (CCU) determined which classes each ASCII character belonged to through populating a bitvector at compile-time of what character belonged to the given class.

We can see that these previous methods [9, 60, 81, 64, 91, 32] are very generic in nature, and still require the entire text to be processed. However, the generic nature of these optimizations has inherent overheads. Our regexp accelerator approaches the PHP regexps from a different angle. Instead of optimizing generic regexps and generic input data, we take advantage of known characteristics of the input text and regular expressions to skip large portions of FA processing. Only when the input data cannot be skipped (either through content sifting or at the end of content reuse), we fall back to traditional FA processing. In summary, this work is orthogonal to the previous optimizations listed above. In other

words, our regexp accelerator can benefit further by adopting these prior proposals.

5.6 Summary

Server-side, content-rich 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, large-scale, content-rich PHP applications in this dissertation exist across a wide-range of other PHP web applications such as Laravel [53], Symfony [89], Yii [110], Phalcon [74] etc. and hence will all gain execution efficiency when using our proposed accelerators.

6 EVALUATION OF PHP ACCELERATORS

This chapter evaluates our proposed domain-specific PHP accelerators and presents the simulation results. Section 6.1 describes the large-scale, content-rich, server-side PHP applications that we study. Section 6.2 describes the simulation infrastructure and approach. Section 6.3 details the configurations of baseline processor and proposed accelerators. Section 6.4 evaluates our propsed accelerators. Finally, Section 6.5 summarizes the chapter.

6.1 Experimental Workloads

We study three popular large-scale, content-rich PHP web applications – WordPress [107], Drupal [20], and MediaWiki [67] from the *oss-performance* suite [39]. WordPress is reportedly the easiest and most popular blogging platform in use today supporting more than 60 million websites [108], capturing 59% of the market share of PHP web frameworks [98]. Drupal powers at least 2.3% of all web sites worldwide (captures 4.7% of the market share [98]), including some of the busiest sites on the web, ranging from discussion forums and personal blogs to corporate sites [21]. MediaWiki serves as the platform for Wikipedia and many other wikis, including some of the largest and most popular ones.

6.2 Simulation Infrastructure

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 [95], configured to use the HHVM (with all its optimizations turned on) via FastCGI [25].

We use gem5 [94] for microarchitectural characterization of the PHP applications. We evaluate our proposal using an in-house trace-driven simulator, which is described next.

6.2.1 Trace-Driven Simulator

All experiments are run using a trace-driven simulator, configured for an aggressive out-of-order core modeled after an Intel Xeon-based (4 wide out-of-order) server microarchitecture. The trace captures committed (completed) instructions along with their operand values and simulation time stamps (*commit* time during execution). The instruction traces thus recorded are used to simulate the performance benefits and energy savings from using our proposed accelerators. In order to model the execution of our accelerators, we identified the potential sites in the HHVM object code that can be offloaded to our proposed accelerators. Later, when the simulator exercises a potential accelerator site in the recorded trace, it replaces the elapsed time of the accelerated code in our baseline core with the access latency (provided by CACTI and Verilog synthesis) of an accelerator (where it is offloaded to) in case the accelerator satisfies the request.

Note that the simulator assumes that incorporating these accelerators in the baseline core does not change the data cache access pattern as broadly observed in previous studies

on accelerators. As a result, memory references from these accelerators are assumed to incur the same latencies as they do observe in the baseline server processor without accelerators. Hence, while estimating the execution time of an offloaded code in our specialized architecture, the simulator considers the access latency of the appropriate accelerator in conjunction with the additional latencies (the same corresponding latencies as in baseline core) in retrieving the "necessary" memory references from the accelerator.

However, since the hardware heap manager in our specialized architecture stores a small size class table along with corresponding free list entries in hardware, it does not access the data cache to retrieve these components from the software heap manager data structure while satisfying a request from the hardware table. Hence, in case of a hardware heap manager hit, the accelerated core does not incur any additional latencies from accessing the software heap manager structure, which the simulator does model. In case of deallocating a memory block to the hardware heap manager, if adding the new block overflows the maximum number of entries for a given size class, then the software handler updates the software heap manager structure with the evicted block (using a single `str` instruction). The simulator accounts for this extra instruction.

When the hash table in our specialized core is invoked with the pointer (physical memory address in our design) to a key and the base address of a requested hash map, it retrieves the bytes corresponding to the requested key from memory and incurs the same latencies as it would in our baseline core in retrieving that key from memory. Other than that, the accelerated hash map access site does not experience any overhead from traversing the memory-resident software hash map structure. Furthermore, when a dirty entry is evicted from the hash table accelerator to make space for an incoming key-value

pair, a software handler requires to update the software hash map data structure with the evicted key-value pair. The simulator takes into account the cycle costs associated with the eviction.

The string accelerator and the regular expression accelerator are invoked with the pointers of the various required string operands (subject string, pattern string and/or replacement string) rather than the data itself. Hence, memory references from the string accelerator and the regular expression accelerator do incur the same latencies as they would in our baseline processor in accessing the data from memory. Our simulator models these latencies in detail.

6.2.2 Area and Power Estimation

We use McPAT [56] to collect core power and energy. We use CACTI 6.5+ [56] to estimate energy and area of three accelerators – hash table, heap manager, and regular expression accelerator. We implement the string accelerator in Verilog and synthesize using TMSC 45nm standard cell library operating at 2GHz to estimate its area and energy per access.

6.3 Simulator Configuration

Our simulated server processor is configured similarly to the Intel Xeon-based (4 wide out-of-order) server. Table 6.1 details the processor configuration. 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+ [56] to

Table 6.1: Processor configuration

Category	Configuration
Out-of-Order Core	2GHz, 4-wide fetch/commit, 5-wide issue Reorder buffer: 168 entries Instruction queue: 54 entries Physical registers: 160 INT/144 FP Load queue: 64 entries, Store queue: 36 entries Branch Predictor: 32KB LTAGE Branch target buffer: 4K entries
Execution Units	Integer ALUs: 3 (1 cycle, 3 cycle multiply) Memory: 2 (1 cycle AGU) FP adder/multiplier: 2 (5 cycles) FP div/square-root: 1 (10 cycles)
Memory/Caches	L1 instruction: 32 KB, 4-way, 4 MSHRs, 1 cycle, 2-ahead tagged prefetcher L1 data: 32 KB, 4-way, 16 MSHRs, 3 cycles, 2-ahead stride prefetcher L2: 2 MB, 8-way, 16 MSHRs, 12 cycles, 2-ahead stride prefetcher Off-Chip Memory: 2GB DDR3-1600

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. The hash table, the heap manager, the string accelerator, and the regular expression accelerator take about 0.12 mm^2, 0.03 mm^2, 0.05 mm^2, and 0.02 mm^2 of area respectively. 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.

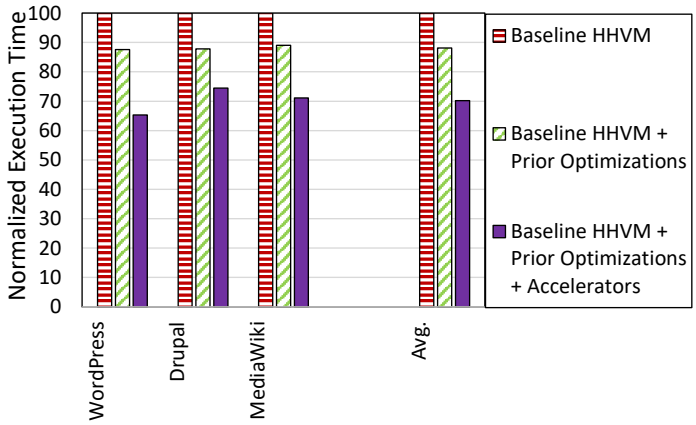


Figure 6.1: Improvement in execution time with applying prior optimizations and our specialized hardware. Execution time is normalized to unmodified HHVM.

6.4 Results

Our evaluation is divided into five subsections. Section 6.4.1 presents the performance and energy benefits obtained with deploying our specialized hardware. Section 6.4.2 illustrates the detailed breakdown of performance benefits. Section 6.4.3 demonstrates the sensitivity of the hash table's hit rate, eviction rate and the resultant benefit to its size. Section 6.4.4 presents in detail the heap manager accelerator's sensitivity to its free list size and number of size classes. Finally, Section 6.4.5 details the string accelerator's sensitivity to various string functions and different input string sizes.

6.4.1 Performance and Energy Improvement

Figure 6.1 shows the improvement in execution from our specialized architecture. Applying the prior research proposals as discussed in Section 5.3 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

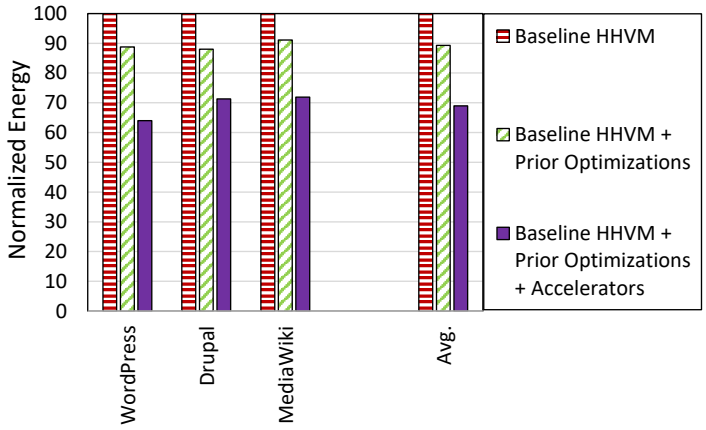


Figure 6.2: Improvement in energy with applying prior optimizations and our specialized hardware. Energy is normalized to unmodified HHVM.

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 5.4), and consequently benefits least from 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 energy savings. Figure 6.2 shows the improvement in energy from our specialized architecture. We calculate total energy consumption of our accelerators by using simulation counters of the cycles offloaded to each accelerator, multiplied by the accelerator energy numbers provided by CACTI and Verilog synthesis. Applying the prior research proposals as discussed in Section 5.3 brings down the average energy to about 89.3% of the energy observed in baseline unmodified HHVM. 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

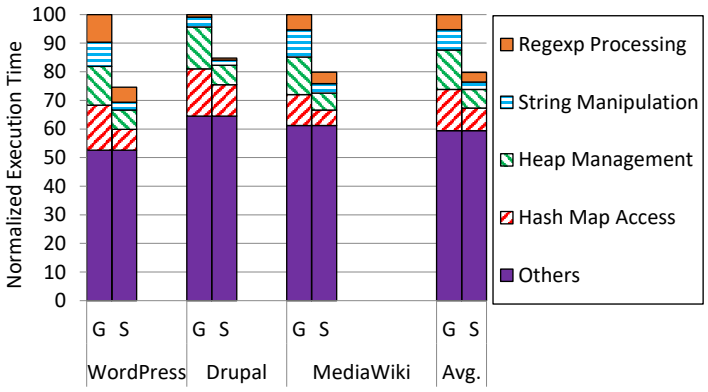


Figure 6.3: Breakdown of execution time. *G* refers to the execution time with applying optimizations from prior works as discussed in Section 5.3, *S* refers to the execution time with all our proposed accelerators. Execution time is normalized to *G*.

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.

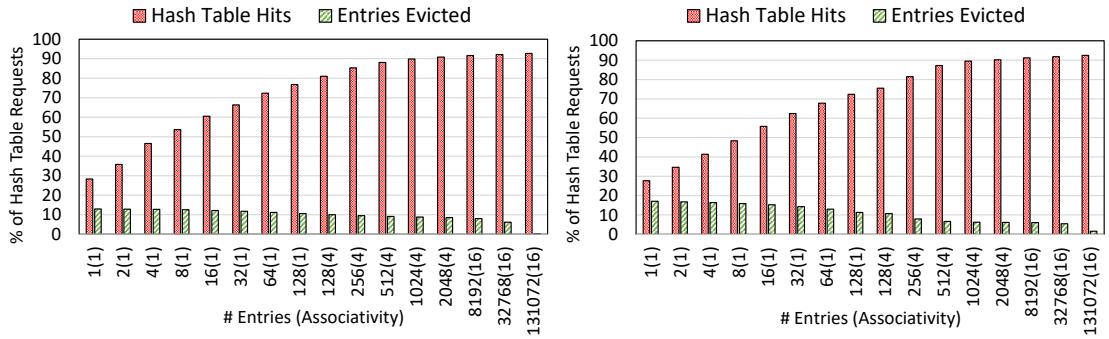
Memory allocation requests (*malloc* and *free*) require on average 69 and 37 x86 micro-ops, respectively, in software to execute (assuming cache hits). Hash map walks in software require on average 90.66 x86 micro-ops. Our specialized architecture saves energy from accelerating the frequently executed 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.

6.4.2 Breakdown of Execution Time

Figure 6.3 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 5.14 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.

6.4.3 Sensitivity to the Hash Table Size

In order to observe the effects of hash table sizing on its hit and eviction rate and the resultant speedup from the hash table accelerator, we sweep the size of the hardware hash table from very small to very high number of entries with varying associativity. The results of this sweep are shown in Figure 6.4. Since SET operations never miss in our design, a hash table with very few entries (1, 2 or 4) exhibits such modest hit rates for WordPress and MediaWiki in Figure 6.4a and Figure 6.4b respectively. Having support of a SET operation



(a) Hash table hit and eviction rate (WordPress).(b) Hash table hit and eviction rate (MediaWiki).

Figure 6.4: Effect of hash table size on its hit and eviction rate.

in hardware helps in serving a major fraction of the short-lived hash map accesses from the hash table. In order to make space for incoming key-value pairs in the hardware hash table, a software handler writes back (evicts) dirty key-value pairs to memory-resident software hash maps. As the eviction process must be performed in software, a low eviction rate is desired to maximize the performance benefit obtained from using a hash table accelerator with high hit rate. The eviction rate drops modestly with increase in the size of the hash table for WordPress, whereas it decreases sharply for MediaWiki. A larger hash table offers more time to a dirty key-value pair to reside in the hash table without being ever written back before getting invalidated by a subsequent hash map Free request. The hit rate improves promptly with increase in size of the hash table. However, once the hash table is sized to 512 entries, the hit rate does not improve significantly with adding more entries. With 512 entries, the eviction rate drops to 9.1% and 6.64% for WordPress and MediaWiki, respectively. It drops to very low values of 0.2% and 1.6% with using much larger hash tables (131072 entries). A hash table with 512 entries is close to achieve a high enough hit rate and low enough eviction rate that matches an idealized design with unlimited hash table entries.

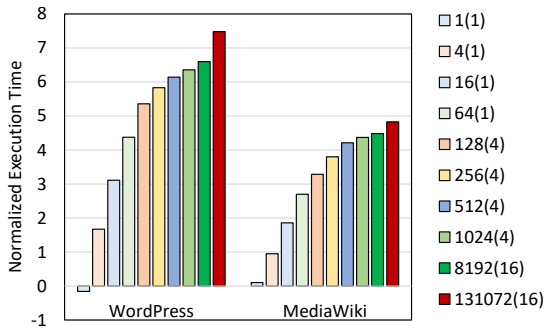


Figure 6.5: Effect of hash table size on speedup. Execution time is normalized to unmodified HHVM.

Unsurprisingly, Figure 6.5 demonstrates that too small of a hash table results in almost no speedup. With a 1 entry hash table, WordPress in fact observes slowdown rather than speedup despite experiencing a modest hit rate of 28.25% (Figure 6.4a). The performance benefit numbers in Figure 6.5 is normalized to the execution time of the PHP applications obtained with unmodified HHVM framework. At a low hit rate with few hash table entries, not only control frequently falls back to the software to perform regular hash map access (the same instructions that we intended to optimize away) in memory but also the additional cycles spent on performing hardware hash table lookups to determine a hit or miss add significantly to the total cycle count. Besides, high enough eviction rates of dirty key-value pairs from smaller hash tables (up to 64 entries) contribute significantly to the total instruction count, overshadowing the modest hit rates of smaller tables and thus causing only marginal benefit from deploying them. As the performance benefit does not not change significantly beyond 512 entries, we choose a hash table accelerator with 512 entries in our design.

6.4.4 Sensitivity to the Heap Manager Size

Sensitivity to the size of free lists. Table 6.2 shows the distribution of time typically provided to the pointer-based prefetcher to prefetch a block from the software heap manager structure as the size of the hardware free list varies in our heap manager accelerator. A hardware free list with fewer entries will provide its prefetcher less time and flexibility for retrieving a block from memory, whereas a list with large number of entries will offer more time to its prefetcher and will help it to be more timely. For example, for a hardware free list of four entries, as the *high watermark* is set to two, typically in steady state the free list will contain two available blocks and generate a prefetch request upon servicing a memory allocation request. In this case, as long as the prefetch request can retrieve an available block from memory before the third consecutive allocation request to the same hardware list (assuming no intervening deallocation), it is capable of hiding the latency of the third allocation request (which would have found a miss otherwise in the associated hardware free list). The larger the size of a free list, the longer the difference between when a prefetch request is generated and when it is likely to be used by an application.

As these PHP applications put less pressure on the cache hierarchy (Figure 5.1b), the software free lists of the heap manager data structure are very likely to reside in the caches during the course of a PHP script's execution, if not in the private L1 cache all the time. Hence, chances of prefetch requests experiencing more than a last level cache access latency in retrieving available blocks from software free lists are very low. As shown in Table 6.2a, even for a free list of size 2, WordPress only requires 0.45% of prefetch requests to retrieve blocks within 15 cycles, whereas it offers $15-30$, $30-60$, $60-120$, $120-240$, $240-480$, and

Table 6.2: Table showing the prefetch timeliness of the heap manager accelerator.

(a) Prefetch Timeliness (WordPress).

	Free List Size (Number of Entries)					
Time-to-use (cycles)	2	4	8	16	32	64
0-15	0.45	0	0	0	0	0
15-30	0.21	0	0	0	0	0
30-60	0.7	0	0	0	0	0
60-120	0.6	0	0	0	0	0
120-240	17.97	0	0	0	0	0
240-480	7.26	15.06	0	0	0	0
> 480	72.81	84.94	100	100	100	100

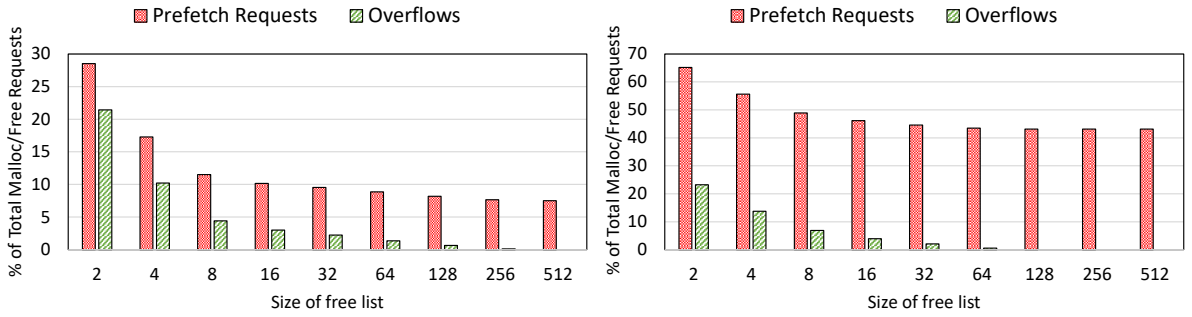
(b) Prefetch Timeliness (Drupal).

	Free List Size (Number of Entries)					
Time-to-use (cycles)	2	4	8	16	32	64
0-15	0	0	0	0	0	0
15-30	0	0	0	0	0	0
30-60	0.68	0	0	0	0	0
60-120	1.34	0	0	0	0	0
120-240	2.54	0	0	0	0	0
240-480	6.59	1.46	0.07	0	0	0
> 480	88.85	98.44	99.93	100	100	100

(c) Prefetch Timeliness (MediaWiki).

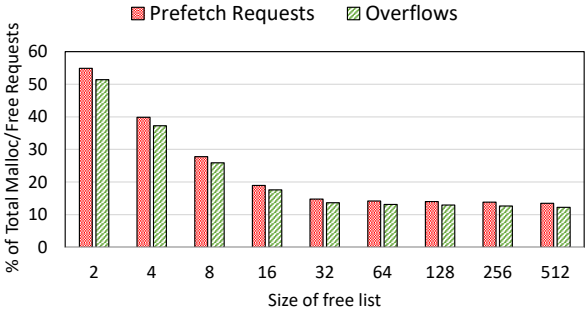
	Free List Size (Number of Entries)					
Time-to-use (cycles)	2	4	8	16	32	64
0-15	0.02	0	0	0	0	0
15-30	0.08	0	0	0	0	0
30-60	1.04	0	0	0	0	0
60-120	4.66	0.05	0	0	0	0
120-240	7.17	0.68	0	0	0	0
240-480	7.03	4.19	0	0	0	0
> 480	80.01	95.08	100	100	100	100

at least 480 cycles of latencies to 0.21%, 0.7%, 0.6%, 17.97%, 7.26%, and 72.81% of its prefetch requests respectively to prefetch blocks from memory before they are used. A hardware free list with more than 4 entries always offers its prefetch requests at least 480 cycles of latency to retrieve blocks from software free lists and place it into the hardware table while meanwhile servicing memory allocation requests from the top of the hardware free list. Free lists of bigger sizes should provide more flexibility to the associated prefetchers in hiding latencies and Table 6.2a confirms that. The same trend holds true for the other two applications as well as shown in Table 6.2b and Table 6.2c for Drupal and MediaWiki respectively. In short, smaller free lists do not in general affect the timeliness of prefetch



(a) Prefetch requests and overflows from the heap manager accelerator (WordPress).

(b) Prefetch requests and overflows from the heap manager accelerator (Drupal).



(c) Prefetch requests and overflows from the heap manager accelerator (MediaWiki).

Figure 6.6: Effect of free list size on number of prefetch requests and overflows generated from the heap manager accelerator.

requests generated from them. Hence, the sensitivity study to free list size establishes the fact that even a hardware list with fewer entries is capable of hiding the latencies of prefetching blocks from memory. However, this comes at the expense of generating higher number of prefetch requests and overflows from the hardware lists as discussed next. Hence, a wider hardware free list primarily assists in saving energy for our experimental workloads rather than improving their performance.

Since a memory allocation or a deallocation request typically requires on average only a few tens of cycles[1] in software, and our heap manager accelerator is tightly coupled to a core, a high latency in accessing it will erase any gains from its deployment. This in

[1] A *malloc* and a *free* request typically require on average 38.83 and 19.94 cycles, respectively, in software to execute (assuming hits in software free lists).

turn restricts the number of size classes and the length of their corresponding free lists in hardware. In order to find the effects of free list sizing (with fixed number of size classes) on the number of prefetch requests and overflows and subsequently on performance, we sweep the sizes of free lists (with fixed number of size classes, for example, 8 in this case) to contain from 2 to 512 entries. The results of this sweep are shown in Figure 6.6. It shows the percentage of times a memory allocation request triggers a prefetch request or a memory deallocation request overflows the hardware table as the number of entries in a free list is varied.

Note that the amount of prefetch requests generated by the hardware free lists is normalized to the number of memory allocation requests observed by the heap manager accelerator whereas the number of overflows from them is normalized to the number of memory blocks an application releases to the hardware table. A larger free list provides enough flexibility to the applications to release memory blocks to the hardware heap manager without necessarily overflowing it. As expected, as the size of the hardware free lists increases in Figure 6.6, the number of generated prefetch requests and overflows from them gradually goes down. An important point to note here is that Drupal generates a much higher number of prefetch requests than the other two applications. As shown in Figure 6.6b, on average in 65.18% cases, a memory allocation request in Drupal triggers a prefetch request (upon the number of available blocks falling below the *high watermark* level) for free lists with 2 entries in contrast to 28.52% and 54.85% cases for allocation requests in WordPress and MediaWiki respectively, as shown in Figure 6.6a and Figure 6.6c. This is attributed to the fact that Drupal generates memory allocation requests at a much higher rate than it releases blocks to the free lists. This causes fewer reuse of memory blocks and

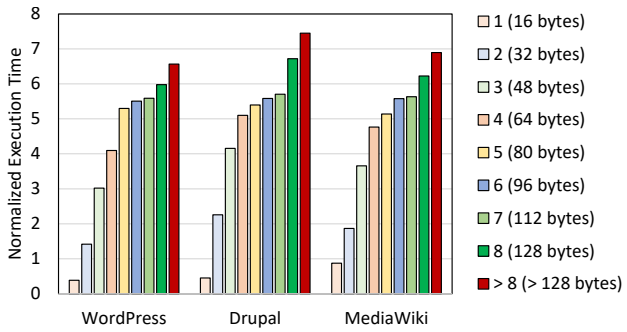


Figure 6.7: Effect of varying number of size classes of the heap manager accelerator on speedup. Execution time is normalized to unmodified HHVM.

results in generating more prefetch requests from the hardware list. The memory usage of the smaller memory slabs in Drupal (Figure 5.8c) confirms this fact where its memory usage never plateaus completely during its execution, as opposed to the memory usage shown for the other two applications (Figure 5.8b and Figure 5.8d). Nevertheless, Drupal offers enough time to the prefetcher to refill the hardware lists in a timely manner and thus exploits the heap manager accelerator to achieve speedup even with smaller free lists, as confirmed in Table 6.2b.

As shown in Figure 6.6, the free lists of 32 entries (with 8 size classes) generate fewer prefetch requests and overflows (within 10-20% of prefetch requests and overflows generated by free lists with 512 entries) without increasing the access latency of the heap manager accelerator. As a result, we choose a size class in our heap manager accelerator to contain 32 entries in its corresponding free list.

Sensitivity to the number of size classes. Figure 6.7 shows how varying the number of size classes, between 1 to more than 8 in the heap manager accelerator changes its performance benefit. Note that the size of each size class is fixed to 32 free list entries as decided from the sensitivity study to free list size. The addition of a size class increases

the maximum size of a memory allocation (or deallocation) request that the hardware heap manager can satisfy by 16 bytes. For example, the heap manager accelerator with 1 size class can only service memory allocation requests of size up to 16 bytes. Similarly 2 size classes can service requests of size up to 32 bytes. Thus the heap manager accelerator with 8 size classes in Figure 6.7 can serve requests that are at most 128 bytes in size. More than 8 size classes in Figure 6.7 refers to caching all the size classes from the conventional software heap manager to the hardware heap manager accelerator. The performance benefit from varying size classes in Figure 6.7 is normalized to the execution time of the PHP applications obtained with unmodified HHVM framework. Since a majority of the allocation and deallocation requests in these PHP applications retrieve at most 128 bytes (Figure 5.8a), a heap manager accelerator with first 8 size classes can capture most of the performance benefit that can be achieved by caching all the size classes from the software heap manager. As a result we consider to deploy a heap manager accelerator with first 8 size classes in our design. The first 4 size classes satisfy a substantial fraction of the allocation requests for Drupal and MediaWiki. Later, when the number of size classes is increased from 7 to 8, these two applications exhibit a sharp jump in speedup.

6.4.5 Sensitivity to the String Accelerator

Table 6.3 displays the number of execution cycles required to execute the following string operations.[2] Each of the other string functions not listed in the chart can be mapped to

[2]string_replace execution time has a data dependence on the size of the replacing substring, replaced substring, and number of replacements made. These numbers [(a)-(b)] designate [(the execution time if the replacing substring is $\leqslant$ the size of the replaced substring with no wrap-around hits)] - [(execution time if replacing substring is greater than the replaced substring and the total string length is $\leqslant$ 32 additional bytes with wrap-around hits)]. Each additional 32 bytes greater than the original string takes an extra cycle of execution time and buffering space in the accelerator.

Function	Subject String Length (Bytes) 32	64	96	128	256	512
memchr	2	2-3	2-4	2-5	2-9	2-17
string_find	2	2-3	2-4	2-5	2-9	2-17
string_replace	3-4	4-6	5-7	6-8	10-12	18-20
string_translate	2	3	4	5	9	17
string_to_lower	2	3	4	5	9	17
string_trim	3	4	5	5-6	5-10	5-18

Table 6.3: Table showing the string accelerator execution time of various functions based on input string size.

one of the string functions in the chart to determine total execution time. For example, *string_find* takes the same number of cycles as the *memchr* function. The numbers listed are for a string engine accelerator having 32 columns (which means that it can insert 32 bytes of subject string into the accelerator per cycle), and 32 rows (which means it can support a pattern string of up to 32 bytes). However, most operations in these applications search for significantly smaller patterns, so unused rows can be power-gated. The number of columns and rows desired is configurable, and can be easily modified for other workloads with different characteristics without breaking its functionality. The string engine can be thought of as a variable-length pipeline. In this configuration, 32 bytes can be fed into the pipeline every cycle, and the total latency to flow through the pipeline depends on the current string operation being performed. Since pointers to the string are passed to the string engine rather than the data itself, the execution time in this chart assumes data is immediately available (L1 cache hits).

Some of the operations such as *string_translate* and *string_to_lower* have constant execution time based on the size of the subject string. This is because the entire subject string must be read, the character replacement is well-defined, and no corner cases varies its execution path.

Alternatively, some of the operations have variable execution time based on data-

dependencies. For these functions, the full range of possible execution time is shown. In some cases of these functions, the hardware can preemptively return the result before reading through the entire subject string. For example, *memchr* and *string_find* return the position of the first match of the pattern character or substring, respectively. Regardless of the subject string length, if the match is found in the first 32 bytes, the result is known and the completed/ready bit can be set two cycles after invocation. However, if the match is not found until the end of the subject string, the completed/ready bit will be set later. Trim is the most complicated of the variable latency functions. It searches to remove leading and trailing instances of a character (typically whitespace) in a given string. The pipeline latency of the *string_trim* operation is 3 cycles. If the first non-matching character is detected in the first 32 bytes, the accelerator can start searching from the end of the string (assuming string size is known). The detection occurs within 2 cycles. For strings larger than 96 bytes, there is potential to skip reading all of the input string. For these large strings, the execution time could be completed in 5 cycles if a non-trimmed character exists in the first and last block. (2 cycles for initial detection (pipeline fill) and 3 cycles for trim latency for the end block processing). In the worst case, every 32-byte block must be read.

6.5 Summary

This chapter presents an evaluation of our proposed domain-specific accelerators for realistic, content-rich, server-side PHP applications. Results show an average 17.93% improvement in performance and 21.01% savings in energy while executing these large-scale, content-rich PHP workloads on our simulation environment. The chapter includes

several sensitivity studies and contains a design-space exploration of the accelerators and illustrates various design tradeoffs with their use.

7 CONCLUSION AND FUTURE DIRECTIONS

7.1 Conclusion

The explosive growth in the scale of users and digital data volumes in the last decade has necessitated a corresponding increase in compute resources to extract and serve information from the raw data. This has created a huge demand to expand existing data centers and build more data centers. For example, Facebook utilizes more than 100,000 servers to provide approximately one trillion page views per month (about 350K per second) and 1.2 million photo views per second [2]. The multi-megawatt power budgets of Internet companies to drive those data centers have brought the spotlight on the performance of PHP scripts in order to improve the energy economics of their web services. Considering the fact that a web page access is typically dominated by the network latency, improving the execution time of PHP scripts will matter very little from the perspective of an end user. However, improving PHP's execution time will play a significant role in drastically improving the server efficiency of Internet companies, leading to significant cost reductions for both provisioning and operating large data centers. In this regard, this dissertation makes two key contributions to improve the execution efficiency of server-side PHP applications.

We first propose a compiler optimization technique called Hash Map Inlining to eliminate the overheads associated with accessing hash maps with variable key names, one of the dominant overheads in server-side PHP applications. This is achieved by specializing accesses to hash maps with variable key names at an access site, as long as the variable at the access site sequences through a number of different, though predictable, fixed key

names at run time. This condition is trivially satisfied for the SQL runtime library functions responsible for populating hash maps with variable key names and causing significant obstacles to good performance in our experimental workloads. A prototype implementation of HMI in the HipHop VM infrastructure shows promising performance benefits on real hardware for a broad array of hash map-intensive server-side PHP applications.

However, we observe that our HMI technique is not very effective for large-scale, content-rich PHP applications that spend most of their time in rendering web pages. In an attempt to improve the execution efficiency of such server-side, content-rich PHP applications, we shift our focus towards designing specialized hardware, as we could not find easy opportunity for conventional microarchitectural enhancements while characterizing them. Four fine-grained PHP activities - hash table accesses, heap management, string manipulation, and regular expression handling are then identified as common building blocks across leaf functions that constitute a significant fraction of total server cycles in these realistic, large-scale PHP applications. Novel, inexpensive hardware accelerators are proposed for these activities that accrue substantial performance and energy gains across dozens of functions. Simulation results reflect that our specialized hardware offers a significant improvement in performance and a considerable reduction in energy while executing the complex, content-rich PHP applications on a state-of-the-art software and hardware platform.

The contributions made in this dissertation should significantly improve the efficiency of web servers and thus in turn directly influence the throughput of data centers.

7.2 Future Work

This section discusses a number of potential avenues for future work that extend the research conducted in this dissertation.

Explore the design space of tightly-coupled accelerators. Through our exploration of tightly-coupled accelerators, we see many future directions that have previously been unexplored for generic tightly-coupled accelerator analysis. Designing tightly-coupled accelerators gave us many different implementation considerations that were difficult using existing design methods. Design considerations may include issue/dispatch logic, data forwarding, interrupts, coherency, consistency models, memory arbitration, memory hierarchy integration, pipeline integration, power/clock-gating, etc. We can see that all the possible combinations of accelerator design choices can grow exponentially, making it difficult to choose the optimal design. For example, looking specifically at the issue/dispatch logic, there are many design choices. There is a tradeoff between performance and hardware complexity based on dispatch/issue logic implementation for speculative execution. For non-speculative execution, the accelerator design is simpler, since there is no need for recovery logic or checkpointing. Speculative execution increases hardware complexity, but can increase performance.

One non-speculative dispatch policy is to make the accelerator instructions in-order by stalling any future instructions after decoding an accelerator invocation instruction. This requires the lowest accelerator hardware complexity since they do not have to worry about data dependencies, checkpointing, branch mispredictions, or reordering between

accelerator invocations. Another non-speculative method could allow for instructions to fill the reorder buffer, which requires register renaming logic as well as support for out of order memory accesses (store queue), but captures ILP around the accelerator invocation. Alternatively, to support speculative execution, there is a potential performance gain, but at the cost of added hardware complexity to support squashes after branch mispredictions. A speculative policy could require accelerator invocations to be in-order, while another could allow reordering of accelerator invocations at the cost of complex checkpointing recovery. As one can imagine, some accelerators may gain no benefit from allowing reordering between accelerator invocations (if each invocation is dependent on the result of the previous), and would use unnecessary power and area to support that functionality. We can quickly see how the design choice is not obvious for a single characteristic, let alone for combining all characteristics.

There are a few examples of added hardware (such as floating-point units, encryption blocks, etc.) that have been integrated into cores over the years which can be thought of as tightly-coupled accelerators. Currently, these accelerators appear to be individually analyzed accelerator by accelerator, since we do not currently know of any prior work which proposes an analytical model to predict which features to support for a given tightly-coupled accelerator.

Since accelerators are currently a very hot topic in the computer architecture community and ever increasing in order to keep up with expected performance increases every computer generation, we expect accelerators to become more and more common for smaller and smaller groups of instructions (motivating tightly-coupled accelerators). We expect analyzing accelerator designs individually to eventually become impractical,

motivating the need for an analytical model for quick design-space exploration. We also expect speculative, out of order designs with frequent accelerator invocations to benefit not only from the instruction acceleration, but also gain a significant benefit from ILP previously unexploited. This is because reducing instruction count increases the effective ROB size. More specifically, ILP can be exploited by instructions before and after the accelerator invocation that previously would not have fit in the same ROB window. We imagine that ROB size, accelerated instruction length, acceleration factor, workload ILP, and other factors could all be used together to create this analytical model for quickly modeling accelerator design optimizations. We see lots of potential in analyzing the many design-space choices of tightly-coupled accelerators and finding relations and analytical models for them.

Investigate the applicability of hash table accelerator in SpGEMM computation. General sparse matrix-matrix multiplication (SpGEMM) is a fundamental building block in various applications such as machine learning algorithms, algebraic multigrid method, breadth first search and shortest path problem. In order to deal with the irregularity of sparse matrices, a parallel SpGEMM algorithm requires to perform updates at random positions in the output sparse matrix, which dominates the execution time [61]. Prior works [61] attempt to mitigate this dominant overhead by parallel merge algorithms. Instead we would to like to explore the potential of our tightly-coupled hash table accelerator to perform the parallel insert operations in hardware without incurring the associated software overheads. Needless to say, deployment of such hash table accelerators will require ensuring load balance of computation and coherency of generated data.

Explore the true potential of Content Sifting technique. Our regexp accelerator takes advantage of known characteristics of the input text and regular expressions to skip large portions of regexp processing in server-side PHP applications. In future, we would like to investigate the true potential of this approach for a diverse set of regexp workloads especially beyond the domain of scripting languages. ANMLZoo [102] is one such diverse benchmark suite of regular expressions that we would like to experiment on in near future. We would also like to see how our Content Sifting technique (used in our regexp accelerator) embraces other prior works on parallelizing regexp processing [64, 91, 32] to further improve the execution efficiency of regexp processing.

Explore the potential of considering strings as primitive data types. In this thesis, we observe that the large-scale, content-rich PHP applications spend significant fraction of their execution time in string functions and a major percentage of the string functions often process few characters (bytes) that can easily fit in CPU registers. However, since string values in general may contain any number of characters, they are allocated from or deallocated to heap memory in modern PHP VM infrastructures. Instead if strings are considered as primitive data types like integers and floating-point numbers, most of the overheads in managing heap-resident strings may be avoided, as they will then be stored in and operated from (if possible) CPU register file across operations without necessarily accessing them from heap memory.

7.3 Closing Remarks

While the performance overheads associated with scripting languages have been significantly mitigated by a commendable growth in JIT compiler performance in recent years, this source of mitigation is gradually drying up just as scripting languages are becoming ubiquitous in today's programming world. This will in turn not only require performing synergistic hardware/software cross-layer optimizations to sustain such significant improvement in execution efficiency, but also will add more burden on the hardware community to design future server processor architecture well-suited for running such scripting language applications. Modern web services combine scripting languages, such as PHP on the server side and JavaScript on the client side. Besides PHP and Javascript, other scripting languages such as Python, Ruby, Lua, Perl, and MATLAB are being widely used in various important domains across the computing industry.

Recent works have demonstrated significant improvement in Javascript's execution efficiency by proposing changes either to the underlying microarchitecture [11] or to the cache subsystem [12, 116]. In this thesis, we follow a hardware/software co-design approach to improve the execution efficiency of realistic, large-scale PHP applications.

Two primary features of the research presented in this thesis make me optimistic as to its future relevancy: (1) it is based on observing characteristics in real-world, large-scale applications, as opposed to characteristics observed in micro-benchmark suites, and (2) domain-specific accelerators have a relatively high probability of seeing widespread adoption in the years to come to sustain the improvement in execution efficiency of scripting

language applications. It would be exceptionally gratifying if at some point hindsight indicates that the insights and the design philosophy presented in this thesis performed a meaningful contribution to improving the execution efficiency of other important real-world applications developed in either scripting or compiled languages beyond the boundary of PHP scripting language.

BIBLIOGRAPHY

[1] K. Adams et al. "The Hiphop Virtual Machine". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '14. Portland, Oregon, USA, 2014, pp. 777–790. DOI: 10.1145/2660193.2660199.

[2] S. R. Agrawal et al. "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. Salt Lake City, Utah, USA, 2014, pp. 19–34. DOI: 10.1145/2541940.2541956.

[3] W. Ahn et al. "Improving JavaScript Performance by Deconstructing the Type System". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United Kingdom, 2014, pp. 496–507. DOI: 10.1145/2594291.2594332.

[4] O. Anderson et al. "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, 2011, pp. 419–430.

[5] B. Atikoglu et al. "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. London, England, UK: ACM, 2012, pp. 53–64. ISBN: 978-1-4503-1097-0. DOI: 10.1145/2254756.2254766. URL: http://doi.acm.org/10.1145/2254756.2254766.

[6] I. Atta et al. "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. Vancouver, B.C., CANADA: IEEE Computer Society, 2012, pp. 188–198. ISBN: 978-0-7695-4924-8. DOI: 10.1109/MICRO.2012.26. URL: http://dx.doi.org/10.1109/MICRO.2012.26.

[7] P. Biggar, E. de Vries, and D. Gregg. "A Practical Solution for Scripting Language Compilers". In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. Honolulu, Hawaii, 2009, pp. 1916–1923. DOI: 10.1145/1529282.1529709.

[8] S. M. Blackburn, P. Cheng, and K. S. McKinley. "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. New York, NY, USA: ACM, 2004, pp. 25–36. ISBN: 1-58113-873-3. DOI: 10.1145/1005686.1005693. URL: http://doi.acm.org/10.1145/1005686.1005693.

[9] R. D. Cameron and D. Lin. "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. Washington, DC, USA: ACM, 2009, pp. 337–348. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508283. URL: http://doi.acm.org/10.1145/1508244.1508283.

[10] T. Cao et al. “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. Portland, Oregon: IEEE Computer Society, 2012, pp. 225–236. ISBN: 978-1-4503-1642-2. URL: http://dl.acm.org/citation.cfm?id=2337159.2337185.

[11] G. Chadha, S. Mahlke, and S. Narayanasamy. “Accelerating Asynchronous Programs Through Event Sneak Peek”. In: *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ISCA ’15. Portland, Oregon: ACM, 2015, pp. 642–654. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750373. URL: http://doi.acm.org/10.1145/2749469.2750373.

[12] G. Chadha, S. Mahlke, and S. Narayanasamy. “EFetch: Optimizing Instruction Fetch for Event-driven Webapplications”. In: *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*. PACT ’14. Edmonton, AB, Canada: ACM, 2014, pp. 75–86. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628103. URL: http://doi.acm.org/10.1145/2628071.2628103.

[13] S. R. Chalamalasetti et al. “An FPGA Memcached Appliance”. In: *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA ’13. Monterey, California, USA: ACM, 2013, pp. 245–254. ISBN: 978-1-4503-1887-7. DOI: 10.1145/2435264.2435306. URL: http://doi.acm.org/10.1145/2435264.2435306.

[14] C. Chambers and D. Ungar. “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. Portland, Oregon, USA: ACM, 1989, pp. 146–160. ISBN: 0-89791-306-X. DOI: 10.1145/73141.74831. URL: http://doi.acm.org/10.1145/73141.74831.

[15] C. Chambers, D. Ungar, and E. Lee. “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 Orleans, Louisiana, USA, 1989, pp. 49–70. DOI: 10.1145/74877.74884.

[16] G. E. Collins. “A Method for Overlapping and Erasure of Lists”. In: *Commun. ACM* 3.12 (Dec. 1960), pp. 655–657. ISSN: 0001-0782. DOI: 10.1145/367487.367501. URL: http://doi.acm.org/10.1145/367487.367501.

[17] L. P. Deutsch and A. M. Schiffman. “Efficient Implementation of the Smalltalk-80 System”. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’84. Salt Lake City, Utah, USA: ACM, 1984, pp. 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542. URL: http://doi.acm.org/10.1145/800017.800542.

[18] M. Dietzfelbinger et al. “Dynamic Perfect Hashing: Upper and Lower Bounds”. In: *SIAM J. Comput.* 23.4 (Aug. 1994), pp. 738–761. ISSN: 0097-5397. DOI: 10.1137/S0097539791194094. URL: http://dx.doi.org/10.1137/S0097539791194094.

[19] G. Dot, A. Martínez, and A. González. "Analysis and optimization of engines for dynamically typed languages". In: *27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2015*. IEEE. 2015, pp. 41–48.

[20] "Drupal. https://www.drupal.org/". In:

[21] "Drupal wikipedia. https://en.wikipedia.org/wiki/Drupal". In:

[22] J. Evans. "Scalable memory allocation using jemalloc, https://goo.gl/rvl2oK". In: 2011.

[23] Y. Fang et al. "Fast Support for Unstructured Data Processing: The Unified Automata Processor". In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 533–545. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830809. URL: http://doi.acm.org/10.1145/2830772.2830809.

[24] M. U. Farooq, Khubaib, and L. K. John. "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. Washington, DC, USA: IEEE Computer Society, 2013, pp. 59–70. ISBN: 978-1-4673-5585-8. DOI: 10.1109/HPCA.2013.6522307. URL: http://dx.doi.org/10.1109/HPCA.2013.6522307.

[25] "FastCGI. https://en.wikipedia.org/wiki/FastCGI". In:

[26] M. Ferdman et al. "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. London, England, UK: ACM, 2012, pp. 37–48. ISBN: 978-1-4503-0759-8. DOI: 10.1145/2150976.2150982. URL: http://doi.acm.org/10.1145/2150976.2150982.

[27] T. B. Ferreira et al. "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. 2011, pp. 92–98.

[28] "Flow – a static type checker for JavaScript. https://github.com/facebook/flow". In:

[29] M. L. Fredman, J. Komlós, and E. Szemerédi. "Storing a Sparse Table with 0(1) Worst Case Access Time". In: *J. ACM* 31.3 (June 1984), pp. 538–544. ISSN: 0004-5411. DOI: 10.1145/828.1884. URL: http://doi.acm.org/10.1145/828.1884.

[30] A. Gal et al. "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. Dublin, Ireland, 2009, pp. 465–478. DOI: 10.1145/1542476.1542528.

[31] S. Ghemawat and P. Menage. "TCMalloc : Thread-Caching Malloc, http://goog-perftools.sourceforge.net/doc/tcmalloc.html". In: 2007.

[32] V. Gogte et al. "HARE: Hardware Accelerator for Regular Expressions". In: *Proceedings of the 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Computer Society, 2016.

[33] D. Gope and M. H. Lipasti. "Hash Map Inlining". In: *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*. PACT '16. Haifa, Israel: ACM, 2016, pp. 235–246. ISBN: 978-1-4503-4121-9. DOI: 10.1145/2967938.2967949. URL: http://doi.acm.org/10.1145/2967938.2967949.

[34] D. Gope, D. J. Schlais, and M. H. Lipasti. "Architectural Support for Server-Side PHP Processing". In: *Proceedings of the 44th International Symposium on Computer Architecture*. ISCA '17. Toronto, Canada, 2017.

[35] N. Goulding-Hotta et al. "The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future". In: *IEEE Micro* 31.2 (Mar. 2011), pp. 86–95. ISSN: 0272-1732. DOI: 10.1109/MM.2011.18. URL: http://dx.doi.org/10.1109/MM.2011.18.

[36] B. Hackett and S.-y. Guo. "Fast and Precise Hybrid Type Inference for JavaScript". In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '12. Beijing, China: ACM, 2012, pp. 239–250. ISBN: 978-1-4503-1205-9. DOI: 10.1145/2254064.2254094. URL: http://doi.acm.org/10.1145/2254064.2254094.

[37] U. Hölzle, C. Chambers, and D. Ungar. "Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches". In: *Proceedings of the European Conference on Object-Oriented Programming*. ECOOP '91. London, UK, UK: Springer-Verlag, 1991, pp. 21–38. ISBN: 3-540-54262-0. URL: http://dl.acm.org/citation.cfm?id=646149.679193.

[38] A. Homescu and A. Şuhan. "HappyJIT: A Tracing JIT Compiler for PHP". In: *Proceedings of the 7th Symposium on Dynamic Languages*. DLS '11. Portland, Oregon, USA, 2011, pp. 25–36. DOI: 10.1145/2047849.2047854.

[39] "https://github.com/hhvm/oss-performance". In:

[40] N. Hua, H. Song, and T. Lakshman. "Variable-stride multi-pattern matching for scalable deep packet inspection". In: *INFOCOM 2009, IEEE*. IEEE. 2009, pp. 415–423.

[41] A. Jaleel et al. "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. 2015, pp. 343–353.

[42] J. A. Joao, O. Mutlu, and Y. N. Patt. "Flexible Reference-counting-based Hardware Acceleration for Garbage Collection". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. Austin, TX, USA: ACM, 2009, pp. 418–428. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555806. URL: http://doi.acm.org/10.1145/1555754.1555806.

[43] C. Jung and N. Clark. "DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage". In: *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 42. New York, New York, 2009, pp. 56–66. DOI: 10.1145/1669112.1669122.

[44] C. Jung et al. “Brainy: Effective Selection of Data Structures”. In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’11. San Jose, California, USA, 2011, pp. 86–97. DOI: 10.1145/1993498.1993509.

[45] S. Kanev et al. “Mallacc: Accelerating Memory Allocation”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: ACM, 2017, pp. 33–45. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037736. URL: http://doi.acm.org/10.1145/3037697.3037736.

[46] S. Kanev et al. “Profiling a Warehouse-scale Computer”. In: *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ISCA ’15. Portland, Oregon: ACM, 2015, pp. 158–169. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750392. URL: http://doi.acm.org/10.1145/2749469.2750392.

[47] M. N. Kedlaya et al. “Improved Type Specialization for Dynamic Scripting Languages”. In: *Proceedings of the 9th Symposium on Dynamic Languages*. DLS ’13. Indianapolis, Indiana, USA, 2013, pp. 37–48. DOI: 10.1145/2508168.2508177.

[48] C. Kim et al. “Short-circuit Dispatch: Accelerating Virtual Machine Interpreters on Embedded Processors”. In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA ’16. Seoul, Republic of Korea: IEEE Press, 2016, pp. 291–303. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.34. URL: http://dx.doi.org/10.1109/ISCA.2016.34.

[49] C. Kim et al. “Typed Architectures: Architectural Support for Lightweight Scripting”. In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’17. Xi’an, China: ACM, 2017, pp. 77–90. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037726. URL: http://doi.acm.org/10.1145/3037697.3037726.

[50] Y. Klonatos et al. “Building Efficient Query Engines in a High-level Language”. In: *Proc. VLDB Endow.* 7.10 (June 2014), pp. 853–864. ISSN: 2150-8097. DOI: 10.14778/2732951.2732959.

[51] O. Kocberber et al. “Meet the Walkers: Accelerating Index Traversals for In-memory Databases”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 468–479. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540748. URL: http://doi.acm.org/10.1145/2540708.2540748.

[52] A. Kolli, A. Saidi, and T. F. Wenisch. “RDIP: Return-address-stack Directed Instruction Prefetching”. In: *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-46. Davis, California: ACM, 2013, pp. 260–271. ISBN: 978-1-4503-2638-4. DOI: 10.1145/2540708.2540731. URL: http://doi.acm.org/10.1145/2540708.2540731.

[53] “Laravel. https://laravel.com/”. In:

[54] S. Lee, T. Johnson, and E. Raman. "Feedback Directed Optimization of TCMalloc". In: *Proceedings of the Workshop on Memory Systems Performance and Correctness*. MSPC '14. Edinburgh, United Kingdom: ACM, 2014, 3:1–3:8. ISBN: 978-1-4503-2917-0. DOI: 10.1145/2618128.2618131. URL: http://doi.acm.org/10.1145/2618128.2618131.

[55] S. Li et al. "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. Portland, Oregon: ACM, 2015, pp. 476–488. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750416. URL: http://doi.acm.org/10.1145/2749469.2750416.

[56] S. Li et al. "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. New York, New York: ACM, 2009, pp. 469–480. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669172. URL: http://doi.acm.org/10.1145/1669112.1669172.

[57] T. Li et al. "Operating system support for overlapping-ISA heterogeneous multicore architectures". In: *2010 IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2010, pp. 1–12.

[58] K. Lim et al. "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. Tel-Aviv, Israel: ACM, 2013, pp. 36–47. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485926. URL: http://doi.acm.org/10.1145/2485922.2485926.

[59] K. Lim et al. "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments". In: *Proceedings of the 35th Annual International Symposium on Computer Architecture*. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 315–326. ISBN: 978-0-7695-3174-8. DOI: 10.1109/ISCA.2008.37. URL: http://dx.doi.org/10.1109/ISCA.2008.37.

[60] D. Lin et al. "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*. 2012, pp. 373–384. DOI: 10.1109/HPCA.2012.6169041. URL: http://dx.doi.org/10.1109/HPCA.2012.6169041.

[61] W. Liu and B. Vinter. "An efficient GPU general sparse matrix-matrix multiplication for irregular data". In: *IEEE 28th International Parallel and Distributed Processing Symposium, 2014*. IEEE. 2014, pp. 370–381.

[62] P. Lotfi-Kamran et al. "Scale-out Processors". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Portland, Oregon: IEEE Computer Society, 2012, pp. 500–511. ISBN: 978-1-4503-1642-2. URL: http://dl.acm.org/citation.cfm?id=2337159.2337217.

[63] C.-K. Luk et al. “Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation”. In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’05. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[64] J. V. Lunteren et al. “Designing a Programmable Wire-Speed Regular-Expression Matching Accelerator”. In: *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-45. Vancouver, B.C., CANADA: IEEE Computer Society, 2012, pp. 461–472. ISBN: 978-0-7695-4924-8. DOI: 10.1109/MICRO.2012.49. URL: http://dx.doi.org/10.1109/MICRO.2012.49.

[65] S. Manegold, M. L. Kersten, and P. Boncz. “Database Architecture Evolution: Mammals Flourished Long Before Dinosaurs Became Extinct”. In: *Proc. VLDB Endow.* 2.2 (Aug. 2009), pp. 1648–1653. ISSN: 2150-8097. DOI: 10.14778/1687553.1687618. URL: http://dx.doi.org/10.14778/1687553.1687618.

[66] J. McCarthy. “Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I”. In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: http://doi.acm.org/10.1145/367177.367199.

[67] “MediaWiki. https://www.mediawiki.org/wiki/MediaWiki”. In:

[68] M. Mehrara and S. Mahlke. “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, 2011, pp. 74–84.

[69] “MongoDB. https://www.mongodb.org/”. In:

[70] “MySQL DBMS. https://www.mysql.com/”. In:

[71] “Oracle Database. https://www.oracle.com/database/index.html”. In:

[72] S. Padmanabhan et al. “Block Oriented Processing of Relational Database Operations in Modern Computer Architectures”. In: *Proceedings of the 17th International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 567–574. ISBN: 0-7695-1001-9. URL: http://dl.acm.org/citation.cfm?id=645484.656552.

[73] “PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/”. In:

[74] “Phalcon. https://phalconphp.com/en/”. In:

[75] A. Putnam et al. “A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services”. In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA ’14. Minneapolis, Minnesota, USA: IEEE Press, 2014, pp. 13–24. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665678.

[76] V. Raman et al. “Constant-Time Query Processing”. In: *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*. ICDE ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 60–69. ISBN: 978-1-4244-1836-7. DOI: 10.1109/ICDE.2008.4497414. URL: http://dx.doi.org/10.1109/ICDE.2008.4497414.

[77] A. Rigo and S. Pedroni. "PyPy's Approach to Virtual Machine Construction". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA, 2006, pp. 944–953. DOI: 10.1145/1176617.1176753.

[78] E. Rohou, B. N. Swamy, and A. Seznec. "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. San Francisco, California: IEEE Computer Society, 2015, pp. 103–114. ISBN: 978-1-4799-8161-8. URL: http://dl.acm.org/citation.cfm?id=2738600.2738614.

[79] "RUBBoS: Bulletin Board Benchmark. http://jmob.ow2.org/rubbos.html". In:

[80] "RUBiS: Rice University Bidding System. http://rubis.ow2.org/". In:

[81] V. Salapura et al. "Accelerating Business Analytics Applications". In: *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*. HPCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 1–10. ISBN: 978-1-4673-0827-4. DOI: 10.1109/HPCA.2012.6169044. URL: http://dx.doi.org/10.1109/HPCA.2012.6169044.

[82] H. N. Santos et al. "Just-in-time Value Specialization". In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. CGO '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 1–11. ISBN: 978-1-4673-5524-7. DOI: 10.1109/CGO.2013.6495006. URL: http://dx.doi.org/10.1109/CGO.2013.6495006.

[83] A. Seznec. "A New Case for the TAGE Branch Predictor". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: ACM, 2011, pp. 117–127. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155635. URL: http://doi.acm.org/10.1145/2155620.2155635.

[84] O. Shacham, M. Vechev, and E. Yahav. "Chameleon: Adaptive Selection of Collections". In: *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. Dublin, Ireland, 2009, pp. 408–418. DOI: 10.1145/1542476.1542522.

[85] Y. S. Shao et al. "Toward cache-friendly hardware accelerators". In:

[86] J. Sompolski, M. Zukowski, and P. Boncz. "Vectorization vs. Compilation in Query Execution". In: *Proceedings of the Seventh International Workshop on Data Management on New Hardware*. DaMoN '11. Athens, Greece: ACM, 2011, pp. 33–40. ISBN: 978-1-4503-0658-4. DOI: 10.1145/1995441.1995446. URL: http://doi.acm.org/10.1145/1995441.1995446.

[87] "Standard Performance Evaluation Corporation. SPECweb2005. https://www.spec.org/web2005/ In:

[88] M. A. Suleman et al. "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. Washington, DC, USA: ACM, 2009, pp. 253–264. ISBN: 978-1-60558-406-5. DOI: 10.1145/1508244.1508274. URL: http://doi.acm.org/10.1145/1508244.1508274.

[89] "Symfony. https://symfony.com/". In:

[90] L. Tan and T. Sherwood. "A High Throughput String Matching Architecture for Intrusion Detection and Prevention". In: *Proceedings of the 32Nd Annual International Symposium on Computer Architecture*. ISCA '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 112–122. ISBN: 0-7695-2270-X. DOI: 10.1109/ISCA.2005.5. URL: https://doi.org/10.1109/ISCA.2005.5.

[91] P. Tandon et al. "Hawk: Hardware support for unstructured log processing". In: *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE. 2016, pp. 469–480.

[92] M. Tatsubori et al. "Evaluation of a Just-in-time Compiler Retrofitted for PHP". In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '10. Pittsburgh, Pennsylvania, USA, 2010, pp. 121–132. DOI: 10.1145/1735997.1736015.

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

[94] "The Gem5 simulator: http://gem5.org/". In:

[95] "The nginx web server. https://www.nginx.com/". In:

[96] "The Tiger Php News System Benchmark Suite. http://sourceforge.net/projects/tpns/". In:

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

[98] "Usage of content management systems for websites. https://w3techs.com/technologies/overview In:

[99] "Usage of server-side programming languages for websites. https://w3techs.com/technologies/o In:

[100] "V8 JavaScript Engine. https://developers.google.com/v8/". In:

[101] G. Venkatesh et al. "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. Pittsburgh, Pennsylvania, USA: ACM, 2010, pp. 205–218. ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736044. URL: http://doi.acm.org/10.1145/1736020.1736044.

[102] J. Wadden et al. "ANMLzoo: a benchmark suite for exploring bottlenecks in automata processing engines and architectures". In: *IEEE International Symposium on Workload Characterization (IISWC), 2016*. IEEE. 2016, pp. 1–12.

[103] S. Wakabayashi et al. "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, Information Processing Association, 2009 Annual Summit, and Conference, International Organizing Committee. 2009.

[104] L. Wang et al. “Bigdatabench: A big data benchmark suite from internet services”. In: *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2014, pp. 488–499.

[105] S. R. Warner and J. S. Worley. “SPECweb2005 in the real world: Using IIS and PHP”. In: *Proceedings of SPEC Benchmark Workshop*. 2008.

[106] K. Williams, J. McCandless, and D. Gregg. “Dynamic Interpretation for Dynamic Scripting Languages”. In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’10. Toronto, Ontario, Canada, 2010, pp. 278–287. DOI: 10.1145/1772954.1772993.

[107] “WordPress. https://wordpress.com/”. In:

[108] “WordPress wikipedia. https://en.wikipedia.org/wiki/WordPress”. In:

[109] L. Wu et al. “Q100: The Architecture and Design of a Database Processing Unit”. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’14. Salt Lake City, Utah, USA: ACM, 2014, pp. 255–268. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541961. URL: http://doi.acm.org/10.1145/2541940.2541961.

[110] “Yii. http://www.yiiframework.com/”. In:

[111] “Zend PHP, http://php.net/”. In:

[112] R. Zhang, S. Debray, and R. T. Snodgrass. “Micro-specialization: Dynamic Code Specialization of Database Management Systems”. In: *Proceedings of the Tenth International Symposium on Code Generation and Optimization*. CGO ’12. San Jose, California, 2012, pp. 63–73. DOI: 10.1145/2259016.2259025.

[113] R. Zhang, R. T. Snodgrass, and S. Debray. “Application of Micro-specialization to Query Evaluation Operators”. In: *Workshops Proceedings of the 28th International Conference on Data Engineering*. 2012, pp. 315–321. DOI: 10.1109/ICDEW.2012.43.

[114] R. Zhang, R. T. Snodgrass, and S. Debray. “Micro-Specialization in DBMSes”. In: *Proceedings of the 28th International Conference on Data Engineering*. 2012, pp. 690–701. DOI: 10.1109/ICDE.2012.110.

[115] H. Zhao et al. “The HipHop Compiler for PHP”. In: *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA ’12. Tucson, Arizona, USA, 2012, pp. 575–586. DOI: 10.1145/2384616.2384658.

[116] Y. Zhu et al. “Microarchitectural Implications of Event-driven Server-side Web Applications”. In: *Proceedings of the 48th International Symposium on Microarchitecture*. MICRO-48. Waikiki, Hawaii: ACM, 2015, pp. 762–774. ISBN: 978-1-4503-4034-2. DOI: 10.1145/2830772.2830792. URL: http://doi.acm.org/10.1145/2830772.2830792.