PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X

Xusheng Zhan[1,2], Yungang Bao[1], Christian Bienia[3], and Kai Li[3]

[1]State Key Laboratory of Computer Architecture, ICT, CAS
[2]University of Chinese Academy of Sciences
[3]Department of Computer Science, Princeton University

Abstract

Benchmarks play a very important role in accelerating the development and research of CMP. As one of them, the PARSEC suite continues to be updated and revised over and over again so that it can offer better support for researchers. The former versions of PARSEC have enough workloads to evaluate the property of CMP about CPU, cache and memory, but it lacks of applications based on network stack to assess the performance of CMPs in respect of network. In this work, we introduce PARSEC3.0, a new version of PARSEC suite that implements a user-level network stack and generates three network workloads with this stack to cover network domain. We explore the input sets of splash-2 and expand them to multiple scales, a.k.a, splash-2x. We integrate splash-2 and splash-2x into PARSEC framework so that researchers use these benchmark suite conveniently. Finally, we evaluate the u-TCP/IP stack and new network workloads, and analyze the characterizes of splash-2 and splash-2x.

1. Introduction

Benchmarking is a fundamental methodology for computer architecture research. Architects evaluate their new design using benchmark suites that usually consist of multiple programs selected out from a diversity of domains. The PARSEC benchmark suite, containing 13 applications from six emerging domains, has been widely used for evaluating chip multiprocessors (CMPs) systems since it is released in 2008. According to Google Scholar, the PARSEC paper [4] has been cited nearly 2200 times.

Our experience of maintaining PARSEC over the past years suggests several reasons of the popularity of PARSEC. First, PARSEC covers many emerging domains that are representative for future applications. For example, more than half of 13 PARSEC applications can be represented by deep neural-networks (DNN) [6], which were still in incubative stage around 2008. Second, PARSEC benchmark supports multiple parallel paradigms including pthread, OpenMP and TBB, which facilitate researchers to investigate parallel issues on multicore systems. Third, in retrospect, it is worthwhile that the PARSEC team devoted many efforts in making PARSEC easy to use.

As PARSEC spreads in the community, we receive more and more feedbacks and new demands. In general, there are two requirements that supporting operating system (OS) kernel functionalities, especially file systems and network stacks, and facilitating comparisons of SPLASH-2 and PARSEC (which made us sort of embarrassed). Actually, SPLASH-2 [42] was a very successful benchmark suite for multiprocessor systems. Google Scholar shows that SPLASH-2 has been cited more than 3700 times over the past two decades.

To response the community's requirements, we develop PARSEC 3.0 significant updates. The latest PARSEC 3.0 released recently. Specifically, PARSEC 3.0 exhibits four major changes:

- A user-level TCP/IP stack (u-TCP/IP) is extracted from FreeBSD kernel and is ported to the PARSEC framework.
- Three network benchmarks, which run on top of the u-TCP/IP stack, are introduced.
- A SPLASH-2x suite is created by augmenting SPLASH-2 with five different scales of input sets.
- Both SPLASH-2 and SPLASH-2x are integrated into PARSEC framework and can be invoked directly by the same command *parsecmgmt* as PARSEC applications.

In this paper, we present several challenges in how to build a user-level TCP/IP stack as a benchmark and how to explore large input space of SPLASH-2x and select reasonable input sets. We did comprehensive experiments to characterize the new components of PARSEC 3.0. Experimental results demonstrate that (1) network applications with u-TCP/IP exhibit different behaviors in terms of speedup, locality, and cache/off-chip traffic and (2) SPLASH-2x with multiple scales of input sets still keep the similar behaviors to the original SPLASH-2.

The rest of the paper is organized as follows. Section 2 introduces our motivation. We present an overview of PARSEC 3.0 in Section 3, and then describe the user-level TCP/IP stack in section 4. We show the study of the input sets of SPLASH-2x in Section 5. Section 6 and section 7 are about methodology and experimental results respectively. Related work is discussed in Section 8. Section 9 summarizes this work. Finally, our acknowledgement is shown in Section 10.

2. Motivation

This section briefly describes the previous versions of PARSEC and then discusses new requirements PARSEC users raised.

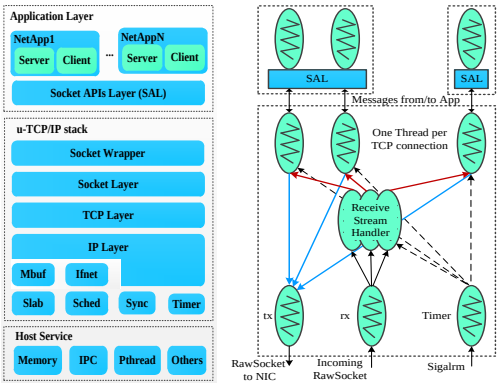
2.1. Previous versions of PARSEC

The goal of PARSEC benchmark suite is to build multi-threaded, emerging and diverse applications that employ state-of-the-art techniques for research. To this end, the initial version of PARSEC 1.0 consisted of 12 workloads and supported two parallelism models: OpenMP [8] and pthreads [28]. PARSEC 2.0 supported a new parallelization model, i.e., Intel Threading Building Blocks (TBB) [31], and added a new application raytrace. PARSEC2.1 does not changes too much in the benchmark suite, except for repairing detected bugs and updating several libraries of some benchmarks in details.

2.2. Network stack

The TCP/IP stack is a set of networking protocols for communication. The BSD TCP/IP stack [25] has been the most influential implementation since it was released in the middle of the 1980s. Some other OSs, e.g., Microsoft Windows, have used BSD-derived code in their implementation of TCP/IP.

Figure 1(a) illustrates the diagram of the BSD TCP/IP stack that consists of three layers and each layer has several auxiliary components. From the perspective of computer architects, the TCP/IP stack, not to mention its importance, is a good candidate for benchmarking on multicore systems. Taking packet receiving as an example, the processing flow in the FreeBSD implementation of the TCP/IP stack involves the following aspects:



(a) the architecture (b) the execution flow

Figure 1: The overview of u-TCP/IP stack

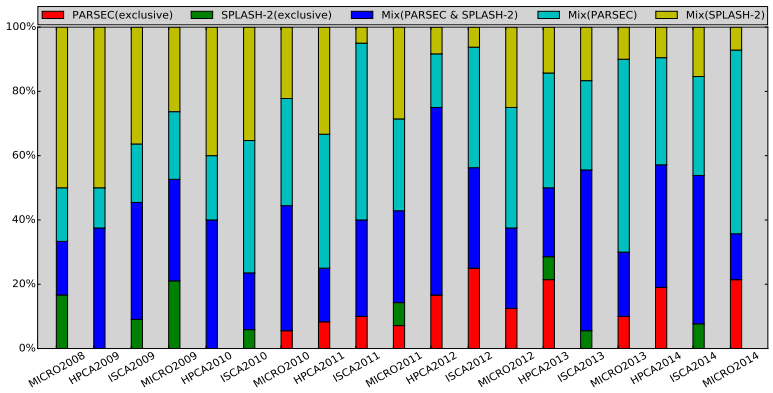


Figure 2: The citations of SPLASH-2 and PARSEC in three top conferences of computer architecture in the past few years

1. There are two levels of parallelization. One is a three-stage pipelined processing flow. Firstly, the packet is received and stored to a kernel buffer (i.e., mbuf)in an interrupt handler and then is moved to a network stream queue that is processed by a FreeBSD kernel task. Finally, the content of the packet is copied into a user buffer. Another level of parallelization is that multiple network streams queue can be processed concurrently by multiple kernel tasks.
2. Various shared resources exist in the parallelized implementation. Although different network streams can be processed in parallel, the FreeBSD implementation of the TCP/IP stack relies on the extensive use of kernel locks for shared resources such as memory buffers (mbuf) allocation and free, global protocol control blocks (tcpcb, inpcb).
3. Although the TCP/IP stack is computation intensive, unlike conventional computation-intensive applications, the control flow of the TCP/IP stack is more complicated. In fact, the implementation of the TCP/IP stack is responsible for executing a complex state machine under various situations.

These characteristics make the TCP/IP stack appeal to not only computer architecture design but also compiler optimization and automatic parallelization. However, directly applying contemporary complier and automatic parallelization techniques to the source codes of the TCP/IP stack is impossible due to the strict constrains of kernel environments. Thus, a user-level TCP/IP stack is desired and will be beneficial for multiple communities, which is the motivation of our work on the design and integration of the u-TCP/IP stack into PARSEC 3.0.

2.3. Integration of SPLASH-2

Except for PARSEC benchmark suite, researchers would like to use another benchmark suite SPLASH-2 to evaluate the performance of new CMPs architecture. SPLASH-2 as a very popular benchmark suite has been extensively used to research on centralized and distributed shared address multiprocessors.

Research [3] shows that SPLASH-2 suite and PARSEC suite have some similar and complementary characteristics. We count and analyze the citation of benchmark suites in three top conferences MICRO, HPCA and ISCA after the initial version of PARSEC being published. As shown in Figure 2, many research teams mixed use these two suites to evaluate the performance of relative designs. In order to let users use SPLASH-2 suite conveniently, it's a good choice to integrate the SPLASH-2 suite into PARSEC framework. Study [5] shows, according to the runtime platforms (physical machines, simulator etc), an integrated benchmark suite should offer different input sets with various scale and fidelity to users. With the improvement of processor performance and the advent of new technologies, algorithms and applications, the input set of SPLASH-2 can't satisfy the requirements of current shared memory multiprocessor. So it's important for researching to extend the input sets of SPLASH-2.

2.4. Others

After publishing PARSEC2.1, some users have reflected several errors in the process of building or running some benchmarks of PARSEC2.1, such that workload facesim has pthread compatibility issue, workload freqmine has a segment error in running. In order to offer better services to researchers, we need to publish a new PARSEC version that revises these errors. It is our initial goal of updating PARSEC that fixes the existed bugs in PARSEC2.1, but not the whole reason. Parallel runtime models are designed to create parallel programs that play important roles in researching the performance of CMPs. The different parallelization methods of one workload can help researchers to be aware and evaluate the different behaviors at run time. PARSEC2.1 already supports three parallelization models, namely POSIX threads (Pthreads), OpenMP and Threading Building Blocks (TBB), but except for blackscholes and bodytrack, all other benchmarks only support one or two parallelization models. Making benchmarks support more parallelization models would be another demand.

3. Overview of PARSEC3.0

This section presents the overview of PARSEC3.0, including new features (§3.1), new components, i.e.,u-TCP/IP stack (§3.2) and SPLASH-2x (§3.3), and others improvements (§3.4).

3.1. New Feature Overview

To satisfy the researchers' requirements, we develop PARSEC3.0 significant updates and add several new elements and features. PARSEC3.0 presents four new key features:

1. The u-TCP/IP stack. The u-TCP/IP stack is a critical element which is extracted from FreeBSD kernel and is ported to the PARSEC framework.
2. Network benchmarks. With the support of u-TCP/IP stack, we create three network benchmarks which run on top of it, and extend the covering domains of PARSEC suite to network.
3. The SPLASH-2x. In order to meet the requirements of current studies, we generate a SPLASH-2x suite by augmenting SPLASH-2 with five different scales of input sets.
4. A uniform framework. All PARSEC applications are managed by the command *parsecmgmt* in PARSEC uniform framework. So we integrate SPLASH-2 and SPLASH-2x into PARSEC framework to simplify operations.

3.2. The u-TCP/IP stack

Kernel TCP/IP protocol stack is a key component to service the TCP/IP protocols in operate system (OS). But different kernels implement their TCP/IP protocol stack in various ways, so it is hard to run an network application on a platform which has different TCP/IP stack from the application. Kernel TCP/IP stack also exists other shortages that have been introduced in research [19]. So a user-level TCP/IP stack is needed to eliminate the weaknesses of kernel TCP/IP stack.

After analyzing the challenges of how to build a user-level TCP/IP stack (§4.4.2), we find reusing TCP/IP stack from a mature kernel is an efficient method to implement u-TCP/IP stack. FreeBSD [15] as one of general purpose OS has been developed more than thirty years. FreeBSD's networking has many advantages, such as all standard TCP/IP protocols have been realized on FreeBSD after version 7.0 [16], FreeBSD can offer stable and reliable network service. Since FreeBSD's TCP/IP stack is robust and capable, we select FreeBSD 8.0 [17] as the ideal platform to port TCP/IP stack from kernel space to user space, and implement u-TCP/IP stack (more details are shown in section 4).

Based on the u-TCP/IP stack, we extend the range of the benchmark suite to network domain and redesign current workloads: dedup, ferret and streamcluster to implement new network workloads that are named netdedup, netferret and netstreamcluster,

respectively. We choose several important properties to present the basic information of three new workloads in Table 1. These network benchmarks can reflect various network requirements from multiple perspectives, such as bandwidth demand, data packet size and parallelization model.

Program	Connections	Bandwidth	Packet Size	Parallelization Model	Latency	Working Set
netdedup	Single	High	Large	Pipeline	Insensitive	Scalable
netferret	Multiple	Low	Small	Pipeline	Sensitive	Scalable
netstreamcluster	Single	High	Large	Data-parallel	Insensitive	Medium

Table 1: Summary of the key properties of three network workloads. The "Scalable" of Working Set refers to large, and these sets can extend to bigger size to satisfy application demand.

3.3. The SPLASH-2x

Since SPLASH-2 was released in 1995, its input sets are never changed. But in the past 20 years, the processing power and manufacturing technology of processors have improved generation by generation. The original input sets of SPLASH-2 may not do well enough to evaluate the performance of current shared memory multiprocessors. So we analyze several input parameters of each programs in SPLASH-2, choose the principal parameters, scale them into different sizes and produce a new input sets of SPLASH-2 which are named SPLASH2x (more details are shown in section 5).

3.4. Others

PARSEC3.0 updates the most benchmark workloads of PARSEC2.1 except for x264 and streamcluster. We renovate these applications to new versions by fixing some issues of them and updating several their dependent libraries, such that the pthread library compatibility issue in suite blackscholes and facesim has been solved. File CHANGELOG shows the details of these updates.

4. The u-TCP/IP stack

This section presents the framework (§4.1) and execution & Communication (§4.2) of u-TCP/IP stack in detail. We next introduce the main procedures of three network programs over u-TCP/IP stack (§4.3). The experiences and lessons (§4.4) of designing and implementing u-TCP/IP stack will be shown in the end.

4.1. Architecture

As shown in Figure 1(a), we illustrate the diagram of the u-TCP/IP stack framework. This framework consists of three parts. The detailed explanation of each part is as follows:

In the top application layer, Socket APIs Layer (SAL) is a library which linked into applications. This library is responsible for forwarding socket syscall requests to the u-TCP/IP stack and providing specific socket API with the "utcpip_"prefix, e.g., utcpip_socket and utcpip_bind etc. This socket API is compatible with the POSIX socket API. Thus, the POSIX socket API can be simply replaced by the "utcpip_" API. When an application calls the "utcpip_" socket API, the library encapsulates the request and forwards it to the u-TCP/IP stack via shared memory.

The middle part is the u-TCP/IP stack extracted from FreeBSD's kernel. The u-TCP/IP stack is responsible for socket manipulations (create, bind and send etc.), TCP layer functionality (flow control, fragment/reassemble etc.) and IP layer (route lookup and fragment etc.). The u-TCP/IP stack has a total of 155K lines of codes, most of which are the same as the original source codes of the FreeBSD's TCP/IP stack. So the u-TCP/IP stack is assumed to behave the same as the FreeBSD's TCP/IP stack except for several services provided by the host system.

The bottom part refers to those host services which emulate the FreeBSD's modules that are required but not essential parts of the TCP/IP stack. For example, the FreeBSD TCP/IP stack uses a Slab system to manage memory. The Slab system requires page-level memory allocation/free functions. Therefore, there is a host service for page-level memory management. The raw socket service is responsible for sending/receiving network packets directly from NIC in order to bypass the local in-kernel TCP/IP stack. The synchronization service uses pthread synchronization primitives to replace FreeBSD's synchronization primitives.

4.2. Execution & Communication

Execution Flow: Figure 1(b) illustrates how the framework works. The u-TCP/IP stack, including the host services, runs as a daemon. The daemon consists of a number of threads. For each socket, there is a corresponding thread (Socket thread) which is

responsible for operating the socket as well as transmitting TCP/IP packets. The outgoing packets are then put into a send buffer. The tx thread collects these packets and sends them to the NIC via a raw socket. The rx thread captures all network packets from the NIC and forwards them the upper TCP/IP processing threads (RecvStream threads). After the data are copied into socket buffer, the Socket thread takes over the rest processing. The Timer thread emulates the timeout mechanism required by the TCP/IP protocols and executes the timer handler every 4 millisecond.

The library is linked into user programs and communicates with the u-TCP/IP daemon via UNIX-domain sockets as well as shared memory. The Unix-domain socket is used for small messages and synchronizations between the daemon and the library while shared memory is used for bulk data transfer between them.

Communication Mode: The communication modes based on u-TCP/IP stack are inter-node mode and intra-node mode. Take server-client communication as an example, the inter-node mode deploys the server application and the clients on different machines. The server and the clients communicate over the underlying Ethernet. For the intra-node mode, the server and the clients are run on one machine and communicate via the loopback device. It should be noted that the server uses the u-TCP/IP stack and the clients use the in-kernel TCP/IP stack, which means that the u-TCP/IP stack can communicate with other TCP/IP stacks. This feature brings portability and flexibility for the framework.

4.3. Network benchmarks

With the u-TCP/IP stack, we extend the range of the benchmark suite to network domain. In order to easily create new network workloads that can represent the typical network application, we redesign current workloads: dedup, ferret and streamcluster to implement new network workloads that are named netdedup, netferret and netstreamcluster respectively. The detailed information about the composition and workflow of each network workload is showed as follows:

netdedup: The netdedup is derived from the PARSEC benchmark dedup. It also uses the mainstream 'deduplication' method which combines global and local compression technique to compress a data stream. This workload consists of two parts: a client which is used to send data stream to server end; a server which is responsible for parallelizing the process of data compression with the pipelined programming model. The netdedup has five pipeline stages in server part, and each parallel stage has a separate thread pool which has a number of threads equal to or greater than the number of available cores in corresponding stage. Each pipeline stage finishes different tasks: the first stage is a serial kernel that receives information stream from client and breaks into coarse grained independent units; the second stage breaks the coarse grained units into fine grained data fragments in parallel way; the third stage is a parallel kernel that computes the SHA1 checksum of each fine grained data chunk to produce an unique identity, and uses global database to check for duplicate fragments; the forth stage in parallel is to compress data chunks and add available block image to database; the fifth stage uses a search tree to reconstruct the original data blocks order and generates the compressed output stream.

netferret: The netferret is an extension of the PARSEC benchmark ferret which represents the emerging next-generation search engines for retrieving non-text document data types. The netferret and ferret have many similar characteristics. They use Ferret toolkit to finish the content-based similarity search about feature rich data and leverage the pipeline model to realize parallel process. However, they have different pipeline stages and architectures. The ferret consists of six pipeline stages: input, query image segmentation, feature extraction, indexing of candidate set, ranking and output. The netferret consists of client side and server side: the client side has three main stages: input, sending data to server and receiving data from server; the server side runs seven stages: accepting, receiving data from client, query image segmentation, feature extraction, indexing of candidate set, ranking and sending processed data to client. The netferret does not changes the main procedure of data process in ferret.

netstreamcluster: The netstreamcluster is derived from the suite streamcluster that solves the online clustering problem. It also contains two parts: client and server. Client sends a stream of data points to server, and then server uses a parallelization scheme to partition these points and assigns each point to its nearest center with a number of predetermined medians. The server takes the most of time to evaluate the overhead of opening a new center. This workload uses the sum of squared distances as a metric to measure the quality of the clustering. In many research domains, such as data mining, intrusion detection and pattern recognition, stream clustering is a common operation that organizes vast data or continuous data under real-time conditions.

4.4. Experiences and lessons

To design such an architecture of u-TCP/IP stack, we encounter a series of design choices (§4.4.1) and implementation challenges (§4.4.2). We show the main points of each process in this subsection.

4.4.1. Design choices In this subsection, we would like to describe them in more detail and present our experience and lessens while making the decisions.

High-Level Organization (Library vs. Daemon): there are two approaches to let application use a user-level TCP/IP stack. One approach is implementing the whole TCP/IP stack as a library. Applications use the TCP/IP stack by linking the library

into their address spaces. The other approach is implementing the whole TCP/IP stack as a daemon process which can handle network requests from multiple application processes.

We choose the daemon approach because of the following reason: Although the library approach eases the complexity of design and implementation due to avoiding inter-process communication (IPC) between applications and the TCP/IP stack, it is unsuitable for multiple-process applications such as Apache and Oracle. On the other hand, for the daemon approach, the behavior of the TCP/IP stack is more close to that of in-kernel stack.

IPC (Asynchronized vs. Synchronized): Since we adopt the daemon approach, we need to choose a way for inter-process communication (IPC) between application process and the daemon process. Overall, there are two kinds of IPC mechanism, i.e., asynchronized IPC such as shared memory, and synchronized IPC such as message, socket, pipe and semaphore. The main pro of asynchronized IPC is that it is highly efficient for bulk data transfer while the main pro of synchronized IPC is that it is easy to synchronize multiple processes due to the wait-notify mechanism.

In practice, we choose a hybrid way which blends share memory and Unix-domain socket. The share memory IPC is used for transferring bulk data while the Unix-domain socket is used for transferring small data as well as synchronizing applications and the TCP/IP daemon process.

Memory Management (Slab vs. malloc/free): Previous studies [23] showed that the performance of the TCP/IP stack heavily depend on memory management. Therefore, FreeBSD adopts the dedicated Mbuf system for the TCP/IP stack in order to improve the performance of memory management. However, the Mbuf system is built up on top of FreeBSD's Slab system which is complicated and difficult to be extracted from FreeBSD kernel. An alternative to the slab system is using the malloc/free memory management provided by host system.

As a benchmark, it is important to meet the representative requirement. Thus, despite of the implementation complexity, we still choose adopting the FreeBSD's slab system for the u-TCP/IP stack in order to keep the same behavior as the original TCP/IP stack.

4.4.2. Implementation In this subsection, we describe the issues we have encountered while implementing the framework and briefly introduce our solutions, which might be helpful for building other benchmark frameworks with consideration of kernel services such as file system.

Identifying Dependency:It is the basis for extracting the TCP/IP stack from FreeBSD kernel. We have modified CScope [7] to output all the dependencies of global variables and functions for given source files. We write a series of Python scripts to analyze the dependency information as well as the include files. We iterate those scripts by providing several seminal TCP/IP files (in freebsd/sys/netinet/), manually check the output, and finally obtain the stable TCP/IP stack source files.

Eliminating Type Conflict: While the source codes are ported to Linux user-level and to be compiled, a lot of type redefinition conflicts occur. We build a tool based on Yacc and Flex to extract all type definitions in the TCP/IP source codes and Linux include files, and then identify those conflicts. We eliminate the type conflicts by removing the FreeBSD definitions and using Linux definitions. The same problem was also reported in this work [33] while porting Linux kernel TCP/IP stack to Linux user-level.

Substituting Locks: We have compared the definitions of FreeBSD locks and pthread locks and found that the latter is a subset of the former. Therefore, we substitute FreeBSD kernel locks with pthread locks while keeping the original definitions of FreeBSD locks except for those locks which can be upgraded or downgraded.

Bypassing Kernel TCP/IP Stack:The goals of this issue include two parts. One is that the u-TCP/IP stack can directly access the network devices. We use Linux's raw sockets and packet filters to achieve this goal. Another goal is fully bypassing kernel TCP/IP stack. For incoming packets, Linux's raw socket only makes a copy of the original packets which are still forwarded to Linux's own TCP/IP stack. We devise a shadow socket approach to tackle this challenge. Once a port is allocated in the u-TCP/IP stack, no matter whether it is a listening port or a source port, the u-TCP/IP creates a host listening socket (called as shadow socket), binds the port to the socket and creates a packet filter associated with the socket which discards any packets.

Timer: The timer implementation is the same as FreeBSD's implementation. We extract the timer module from FreeBSD kernel and encapsulate it into the main thread of the daemon process. The main thread usually sleeps and is waken up by a SIGALRM signal every 4 millisecond.

5. SPLASH-2x

SPLASH-2 benchmark suite expands and improves the workloads of the original SPLASH version which are used to quantitatively evaluate ideas or tradeoffs in several architecture studies. It mainly contains applications and kernels of high performance computing (HPC) domain. Study [3] chooses 42 program characteristics about instruction mix, different working set size and cache line shared rate to analyzes the program behavior of SPLASH-2 and PARSEC, and shows that some workloads of PARSEC and SPLASH-2 fundamentally complement each other. We integrate SPLASH-2 suite into PARSEC framework in this version,

so that researchers can conveniently choose more benchmarks to evaluate their works in a more comprehensive way. Now, users can build, use and manage all SPLASH-2 workloads with the same way as all PARSEC workloads.

Since SPLASH-2 was released in 1995, its input sets are never changed. But in the past 20 years, either processing power or manufacturing technology, the processor has very great improvement in various aspects. Like the development of Intel processor, in 1995, Intel's Pentium Pro [29] only has 150 ~200 MHz effective clock speeds and 0.35 micron manufacturing technology, it was the newest and greatest processor on 32-bit code at that time. But in 2010, 2nd generation Intel core processor has reached to 3.8GHz initial clock and 3rd generation Intel core processor with 22nm manufacturing technology in 2012. The initial datasets of SPLASH-2 may not do well enough to evaluate contemporary shared memory multiprocessors. So we explore the initial input sets of SPLASH-2 and scale them into different sizes. These new input sets of SPLASH-2 is called SPLASH-2x.

5.1. Generation of SPLASH-2x input sets

In order to scale up the input sets of SPLASH-2, we generate new input sets by executing the following steps:

1. We analyze the source codes of each initial program, extract all possible input parameters (= 81) and assign a value range for each parameter. A typical value range consists of MIN, MAX and DELTA, where DELTA already includes arithmetic operations. For example, we will analyze the following values {16K, 32K, ...,8M,16M} for range "$[16K, 16M], \Delta = *2$".

2. We select a few parameters which affect program behavior as the principle parameters and use these parameters candidates to generate refined combinations (nearly 1600).

3. We use the execution time(T) and physical resident memory size(M) as metrics to evaluate the program behavior of different input sets of same program. In the data collection stage, we use Equation(1) to collect the different values of T and M through changing input parameters, where p_i is input parameter. The correlations between two metrics and the datasets of each program are illustrated by Equation(2).

$$(T, M) = F(p_1, p_2, ..., p_n) \tag{1}$$

$$(p_1, p_2, ..., p_n) = F^{-1}(T, M) \tag{2}$$

4. We choose parameters that influence the behavior of most programs as key parameters to generate multiple scales reasonable datasets.

With above procedures, we determine the principle parameters and relevant input sets of each programs. For SPLASH-2x input sets, we adopt the input datasets criterion similar to PARSEC [4] that offers six different scales of input sets, i.e., native (<= 15 minutes), simlarge (<= 15seconds), simmedium (<=4 seconds), simsmall (<= 1second), simdev (N/A) and test (N/A) input sets. Each input set contains all input information required by the corresponding program, where the simdev and test input sets are initial input sets of SPLASH-2 that only be used for test or development; the simsmall, simdev and simlarge inputs are used for performance experiment in different scales of simulation platform; the native input is accurate approximation of real-world input which is suitable for native physical device.

6. Evaluation setup

The evaluation setup of this work focuses on replying the following two issues: What key questions should we address (§6.1)? What methodology should we use to answer these questions (§6.2)?

6.1. Questions to be addressed

In this part we introduce the questions which motivate our research:

Q1: Does the u-TCP/IP stack behave the same as a kernel stack after running in user mode?

Q2: What's the difference between network programs based on u-TCP/IP stack and corresponding original programs?

Q3: Does the program behavior of SPLASH-2x keep a high-fidelity with SPLASH-2?

6.2. Methodology

Max Socket Buffer	TCP Send Window	TCP Receive Window	MTU	MBuf Size	MBuf Ext Size
256KB	32KB	64KB	1500B	256B	2KB

Table 2: The key parameters of the u-TCP/IP stack.

All experiments no matter for u-TCP/IP or SPLASH-2 or SPLASH-2x run on same physical platform(§6.2.1) in this work. For the u-TCP/IP stack, we use the default parameters of FreeBSD TCP/IP stack in Table 2 while each network benchmark uses the u-TCP/IP stack to receive data in the server and employs the kernel TCP/IP stack to send data on the client. When we analyze the relation between network programs and their initial programs, we focus on exploring their Region-of-Interest (ROI) in the server side. In order to explore the similarity between SPLASH-2 and SPLASH-2x, we take the following two procedures to quantify their behavior: We list all program characteristics that will be used to identify different input sets in the first step(§6.2.2). We introduce the analyzed methods of removing correlations of program characteristics and measuring the similarity of various input sets in the last step(§6.2.3).

6.2.1. Experimental Setup To evaluate the similarity between u-TCP/IP and kernel stack, all measurements do on two physical machines which are interconnected with 1Gb Ethernet. Other experiments executes on either of two machines. Each machine equips with 2 Intel Xeon E5645 1.6GHz 6-core processors and 32GB memory, and runs Linux CentOS 6.5 with hyper-threading enabled. The versions of Linux kernel and gcc are 2.6.32 and 4.4.7, respectively.

We choose Pin [32, 34] to explore the nature of programs. As a free tool provided by Intel, Pin focuses on inserting instrumentation into binary programs at run time. Pin directly runs binary application without recompiling as input on the IA-32 or x86-64 instruction-set architecture. Pin provides various APIs for users to build new dynamic program analysis tools, named Pintools. A user creates a new Pintool which should include instrumentation and analysis routines. Instrumentation routines determine the instrumented way of programs. Analysis routines are called by instrumentation routines to record the customizable information of user. We use several APIs of Pin to implement new Pintools for collecting the characteristics of each workloads. After considering the tradeoff between computational cost and accuracy, we mainly use simlarge input set to describe and analyze all characteristics between network programs and their initial workloads.

Characteristic(percentage)	No.	Type	Characteristic(percentage)	No.	Type
Floating point operations	1	Instruction mix	Local store stride $\leq$ 512	16	Memory access stride
ALU operations	2		Local store stride $\leq$ 4096	17	
Branches	3		Global load stride = 0	18	
Memory read	4		Global load stride $\leq$ 8	19	
Memory write	5		Global load stride $\leq$ 64	20	
Stack read	6		Global load stride $\leq$ 512	21	
Stack write	7		Global load stride $\leq$ 4096	22	
Local load stride = 0	8	Memory access stride	Global store stride = 0	23	
Local load stride $\leq$ 8	9		Global store stride $\leq$ 8	24	
Local load stride $\leq$ 64	10		Global store stride $\leq$ 64	25	
Local load stride $\leq$ 512	11		Global store stride $\leq$ 512	26	
Local load stride $\leq$ 4096	12		Global store stride $\leq$ 4096	27	
Local store stride = 0	13		Cache miss rate	28	Cache size
Local store stride $\leq$ 8	14			⋮	
Local store stride $\leq$ 64	15			42	

Table 3: Program characteristics

6.2.2. Program characteristics We sample programs' behavior at several specific points while programs run on a given architecture. In order to evaluate the relation among different input sets, we choose 42 program characteristics from instruction mix, memory access stride and cache size, as shown in Table 3. We choose seven characteristics from instruction mix to reflect the fundamental program properties. The distribution of memory access strides are characterized to local and global strides [26], we focus on twenty characteristics that has been proposed in study [21]. Where local strides use frequency vector to record the stride distribution of each load or store, global strides only capture the load and store stride distribution of temporally adjacent memory addresses. Programs run with different sizes of working set will lead to various data cache miss rates that can reflect the universality of data usage and communication. After analyzing the relation among cache size, threads and cache miss rates in section 6, we find the inflection points of all workloads that mainly range from 8 KB to 131072 KB, so we measure the data cache miss rates with choosing all 15 different power-of-two cache size in this scope. In order to show the multi-thread at the same time, we run all splash2x workloads with different input sets in 8 threads. With these characteristics, we measure all 42 simulation inputs of SPLASH-2x and 14 initial inputs of SPLASH-2 in this work.

6.2.3. Analyzed methods The correlated characteristics of each program will mislead the redundancy analysis of characteristics [3]. Principal Component Analysis (PCA) [35, 36] is a linear transformation method that used for removing the data correlation by converting the dimensions of a n-dimensional correlated data space into a m-dimensional uncorrelated data subspace (where n

< m), meanwhile, losing information minimally. This common mathematical method will generate a new smaller characteristics space that reflects programs true features with minimum parameter estimation error and low computational cost. Each low dimension presents a Principal Component (PC), we choose Kaiser criterion to decide the PCs which were important to remain. After processing correlated data with PCA, we analyze these uncorrelated results with hierarchical cluster analysis (HCA) [20] to explore the inherent similarities of all workloads. HCA is a data mining method that explore the data features to generate a hierarchy of cluster. In this work, we use the bottom-up approach of HCA to build the cluster and compute the Euclidean distance among all programs with different input sets as the appropriate metric. When finish above steps, we will structure a dendrogram to show the similarity of all workloads. In the dendrogram, the vertical axis shows all workloads with various input sets, the horizontal axis denotes the similar linkage distance and each joint represents a merge of two inputs. The more similar two inputs the earlier they merge in the dendrogram, and the smaller intersection value of two inputs.

7. Evaluation Results

Setup	TCP/IP stack		Inter-Node	Intra-Node
	Client	Server		
s1	kernel	kernel	117.5MB/s	-
s2	kernel	User-Level	115.6MB/s	146.4MB/s
Performance Ratio[s2/s1]			98.3%	-

Table 4: The bandwidth of transferring bulk data.

We present reasonable evaluation results to answer each key questions (shown in section §6.1).

Q1:*Does the u-TCP/IP stack behave the same as a kernel stack, after running in user mode?*

Table 4 illustrates that the u-TCP/IP can achieve 115.6MB/s bandwidth which is 98.3% of the peak bandwidth(117.5MB/s) of Linux kernel TCP/IP stack over 1GbE. The theoretical peak bandwidth is 125MB/s for 1GbE network. In practice, owing to the protocols overheads, we can obtain the bandwidth of 118MB/s which is already 94.4% of the theoretical bandwidth.

For the framework with the u-TCP/IP stack, the bandwidth of the inter-node communication mode is 115.6MB/s, 98.3% of the kernel peak bandwidth. In this case, the communication path is "$sender \rightarrow NIC \rightarrow 1GbE \rightarrow NIC \rightarrow receiver$", so the bottleneck should be the underlying 1GbE network. Then we adopt the intra-node mode where server and client communicate via the loopback device. The bandwidth increases to 146.4MB/s. In this case, the communicate path is "$sender \rightarrow loopback \rightarrow memory \rightarrow Emulated_NIC \rightarrow receiver$", so the packets need to be copied from the loopback to the user-level emulated device, which may cause extra time.

We further investigate the time spent on processing single packet: the If_input is 315ns, the Ip_input is 629ns, the Tcp_input is 721ns and the Tcp_do_segment is 6199ns. Most time is spent in the tcp_do_segment() routine which is responsible for reassembling packets and copying data into socket buffer. Since the MTU of the u-TCP/IP stack is 1.5KB, the maximum bandwidth should be about 185MB/s (=1.5KB/7.9μs). Due to synchronization, this bandwidth is still far away from the performance of local kernel communication.

Q2:*What's the difference between network programs based on u-TCP/IP stack and corresponding original programs?*

Program	Input Set(simlarge)					
	Working Set 1			Working Set 2		
	Data Structures	Size	Growth Rate	Data Structure	Size	Growth Rate
dedup/netdedup	data chunks	2 MB	TPs	hash table	256 MB	DS
ferret/netferret	images	128 KB	TPs	data base	64 MB	DS
streamcluster/netstreamcluster	data points	64 KB	TPs	data block	16MB	user-def.

Table 5: Crucial working sets and their growth rates for critical programs.TPs represents the number of thread(pipeline), DS represents the data set size. The sizes of working set are chosen from Figure 4.

We use 'critical workloads' to refer all network benchmarks and their initial programs, and analyze their characteristics in three aspects: parallelization, temporal locality and traffic on/off cache.

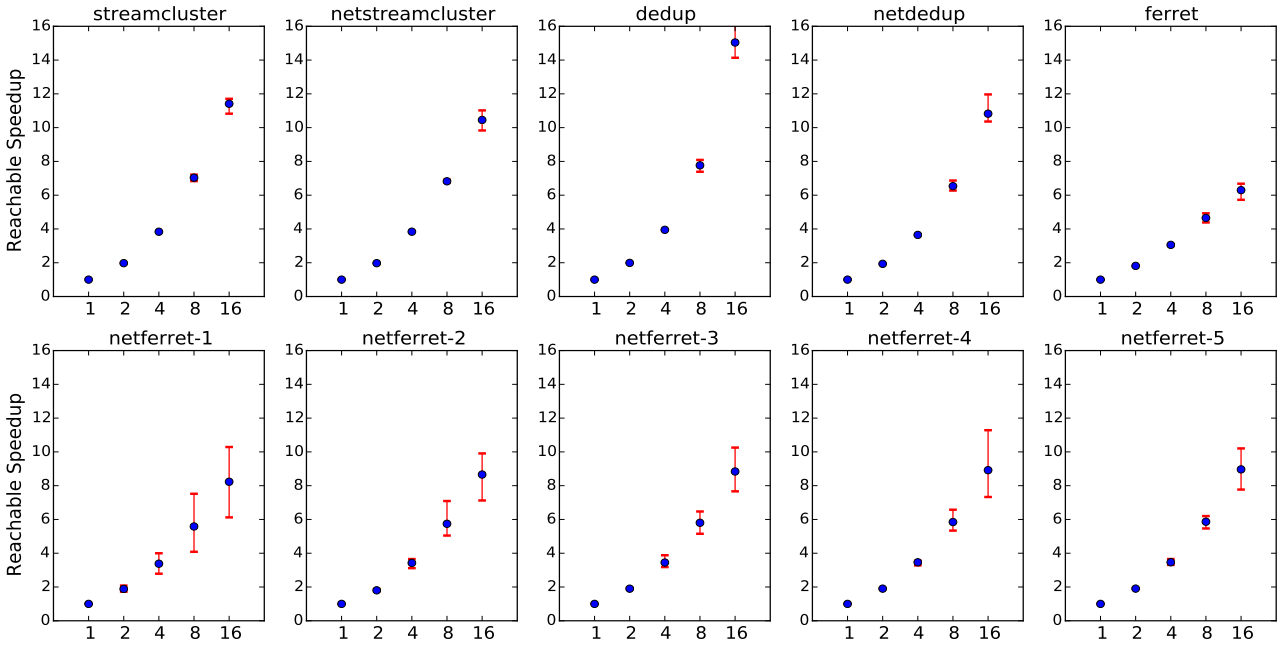


Figure 3: The speedup of critical programs.

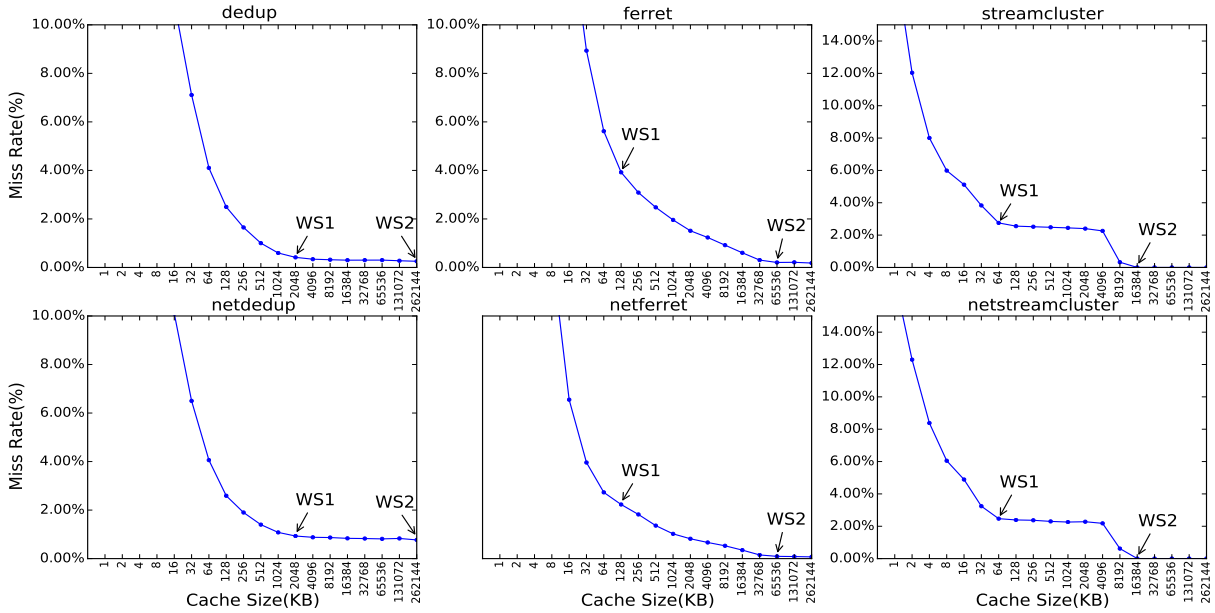


Figure 4: Miss rates versus cache size. The cache configures 4-way set associative and 64 byte lines. WS1 and WS2 refer to key working sets which we show several detail in Table 5.

7.1. Parallelization and Load Balance

The parallelization and load balance are two important characteristics that complement each other to indicate the effective running time of each participant thread. High parallelism relays on load balance, and load imbalance will lead to weak parallelism. In order to exhibit the parallelization of SPLASH-2, Woo et al. [42] evaluate the speedup with the time latency that programs spend on abstract machine. Nevertheless, Christian et al. [4] point out that using a timing model to evaluate the parallelization of programs may not an efficient way, and present a timing-independent methodology that analyzes the number of instructions that executed in ROI of parallel and serial versions. In our approach, we run these programs on a physical machine, count the instructions from all threads that executed in ROI, and then use these instructions to evaluate the parallelization of programs. For network workloads, we collect all instructions from threads in u-TCP/IP stack that can not be paralleled, such as thread Rx_Tcpip, so each network program inflected by these threads in different degrees.We use Equation (3) to calculate the speedup

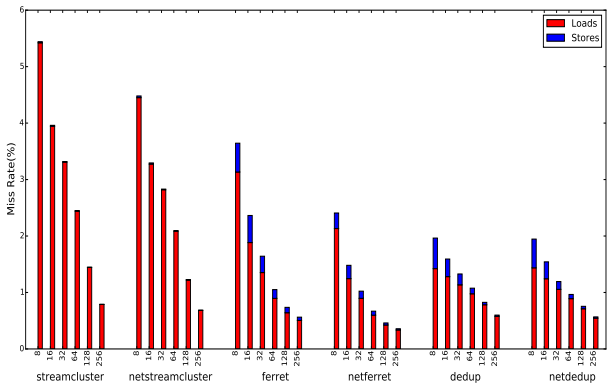


Figure 5: The miss rate of load and store versus line size. 8 threads(pipelines) shares a cache with 4MB capacity and 4-way set associative.

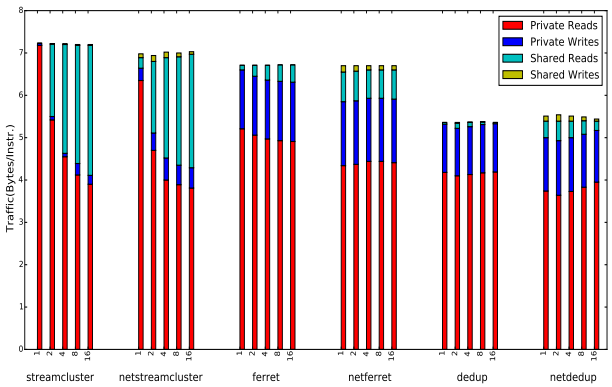


Figure 6: Traffic on cache in bytes per instruction for 1 to 16 threads(pipelines). The cache configures with 4-way set associative and 64 byte lines.

with formulation.

$$Speedup_i = \frac{Series + T_1}{Series + f(T_i)} \tag{3}$$

Where *Serial* is the number of serial instructions. i is the number of threads. T_i is a set that denotes the executed instructions of current thread, and just when i is 1, T_i is an integer instead of a set. $f(T_i)$ denotes the three functions for set T_i : mean() produces average, max() gets maximum and min() for minimum.

In Figure 3, we show the reachable speedup of network programs and their initial versions. For netferret-i, $i \in \{1,2,3,4,5\}$ denotes the number of netferret clients. The bar of each workload denotes the ratio between all instructions of single-thread and average instructions of multi-thread or pipelines in ROI. With less threads, netdedup has similar scalability to dedup, because the serial instructions that bring by u-TCP/IP stack occupy a small fraction of whole. But with the thread increasing, it's not surprising that the proportion of serial part increased. The medium work set of netstreamcluster let it implement an extension slightly lower than streamcluster. Its hard to improve the speedup of ferret and netferret, because the instructions produced in serial input stage are non-negligible. With u-TCP/IP stack, netferret reduces the influence of serial part and reaches higher average ratio than ferret. With the different number of clients, netferret exhibits various speedup. When clients is 5, netferret gets the largest speedup and the smallest gap between the maximum speedup and minimum, so we analyze netferret with 5 clients in following experiments.

7.2. Temporal Locality and Traffic on Cache

The common way to evaluate the temporal and spatial locality of a program is to analyze the change of cache miss rate as the cache capacity and cache line size varied [4, 42]. We take the same method to analyze critical workloads in this work. We present the cache miss rates of all critical programs change with the capacity of cache in Figure 4. Crucial working sets will exhibit the suitable cache capacity of significant data structure of each program, and we mark them on the knee point of cache miss rates change as a function of cache size. We summary all detailed information about each points of inflection in Table 5.

Each network program inherits the main procedure of native program and kernel data structures, so they have same knee points to their native program. With the u-TCP/IP stack, different programs have various effects on cache miss rates, where netdedup has more miss access than dedup, yet netferret and netstreamcluster have less than ferret and streamcluster respectively.

In Figure 5, we show the influence of increased cache line size to spatial locality of all critical programs. With the increasing of cache line size, all programs reduce cache miss. For network programs, they have similar variation tendency of temporal locality to initial programs, but the extents are various.

In order to exhibit the network programs obviously and partly keep compatibility to former research [4], we use a cache with 4 MB size to discuss the communication of all critical workloads on cache and off-chip. It's common that multi-thread use shared data to communicate with each other. In order to discuss the communication performance of these workloads, we show the rate of different type operations on cache in Figure 6. For network programs, we notice an apparent increase in the part of shared writes with the join of u-TCP/IP stack, and the shared reads take more proportion, except for netstreamcluster. There are more private/shared writes in netstreamcluster than streamcluster. With network benchmarks, the communication and sharing mode of PARSEC is more diversity.

Q3:*Does the program behavior of SPLASH-2x keep a high-fidelity with SPLASH-2?*

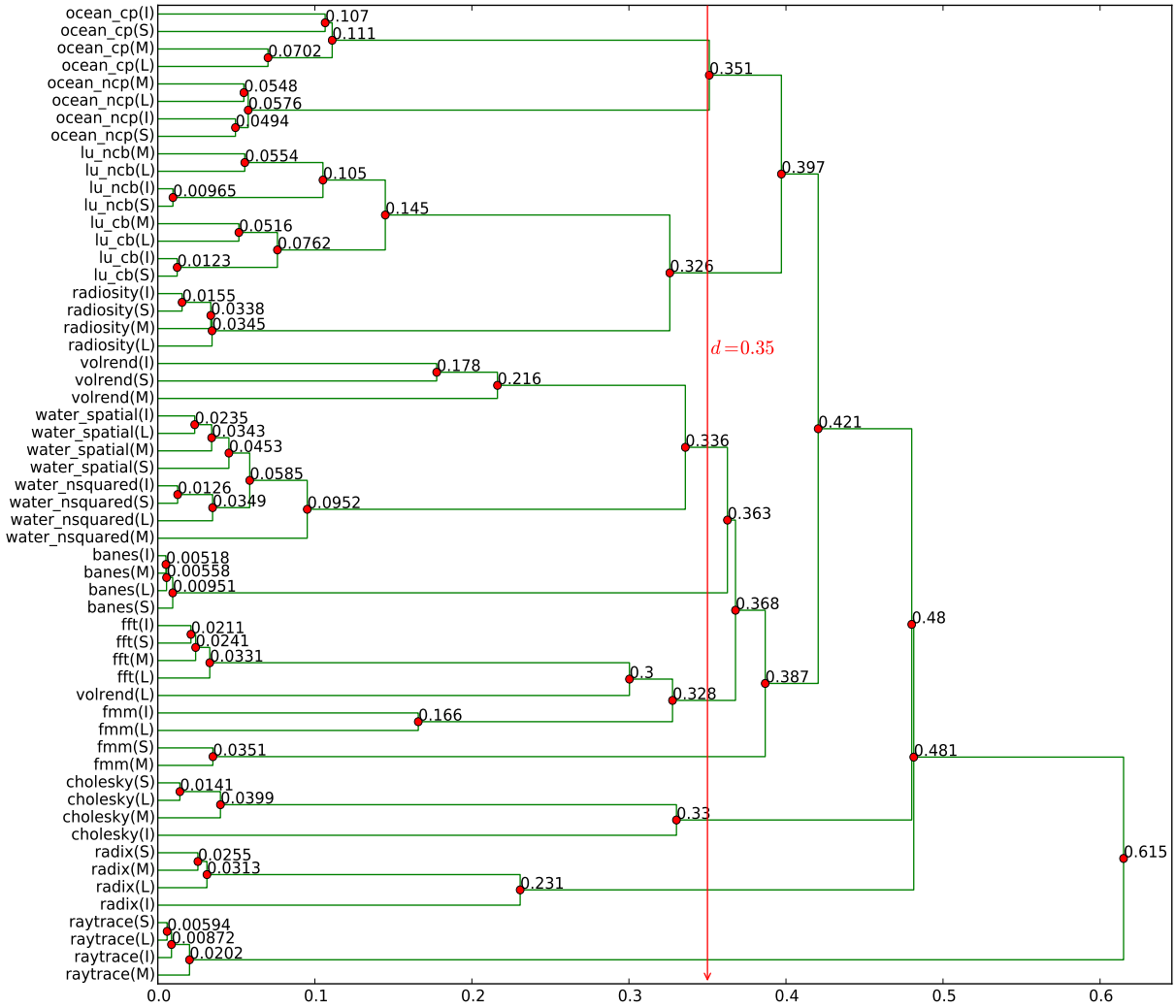


Figure 7: Similarity of SPLASH-2x and SPLASH-2 input sets.Note that alphabets in the parenthesis, where L, M and S denote simlarge set, simmedium set and simsmall set of SPLASH-2x, I denotes the initial sets of SPLASH-2, respectively.

After collecting all characteristics of the new input sets of SPLASH-2x and the initial input sets of SPLASH-2, we then use PCA to analyze their similarity in this part. It's important for PCA to retain the suitable number of PCs without incurring too much additional error. In this work, we follow one common heuristic to keep top 10 PCs so as to retain 96.57% of the variance. The hierarchical clustering algorithm processes the values of these PCs and outputs a dendrogram as shown in Figure 7. Before analyzing the dendrogram, we define three terms, Full Cluster (FCl), Pure Full Cluster (PFCl) and Pure Cluster (PCl), to express our view conveniently. A FCl denotes a cluster that all input sets of same workload fully merged. A PFCl denotes different input sets of same workload fully form a cluster earlier than other workloads with their input sets. A PCl represents different input sets (not all) of same program forming a cluster without the input sets of others. PFCls are a subset of FCls, and PCls are components of PFCls.

The dendrogram shows that vast majority workloads form PFCls, they are barnes, chokesky, fft, water_spatial, ocean_cp, ocean_ncp, lu_cb, lu_ncb, radiosity, radix and raytrace. The similar linkage distances of several FPCls are very short and less than 0.1, such as, d_{banes} = 0.00951, $d_{water\_squared}$ = 0.031 and $d_{water\_spatial}$ = 0.0453. Only d_{radix} (= 0.231) and $d_{cholesky}$ (= 0.33) are relatively long distances. In short, the new input sets of these 11 workloads can hold the characteristics of initial input sets in an efficient way. The remaining 3 programs do not generate corresponding PFCls, so we think these programs can not preserve the original characteristics as well as former programs. The simsmall and simlarge of program water_nsquared merge with its initial input set to create a PCl, but its simmedium overlaps with water_spatial before it can merge with the PCl of water_nsquared. The water_nsquared has not grouped PFCl, but $d_{water\_nsquared}$ (= 0.0952) is shorter than the linkage distances of some FPCls.

As the vertical arrow shown, the linkage distances of most PFCls or FCls are shorter than 0.35, except for volrend and fmm.

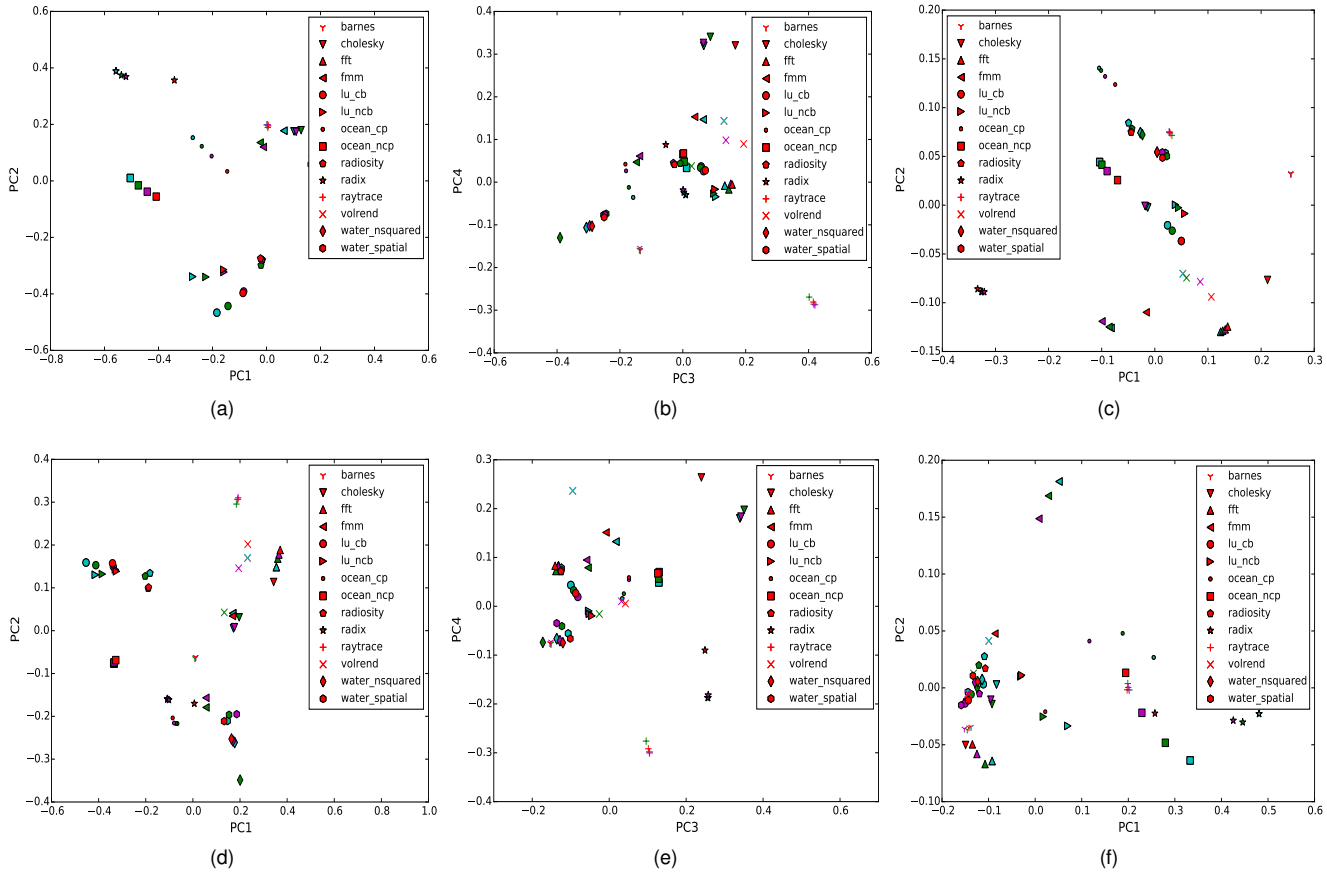


Figure 8: Scatter plot of all workloads with different input sets using the first two or four PCs of different characteristics.(a) and (b) show the first four PCs of all characteristics, (c) shows the first two PCs of instruction mix, (d) and (e) show the first four PCs of memory access stride, and (f) shows the first two PCs of cache size

For fmm, it's simsmall and simmedium input sets form a PCl at first time, then initial input set and simlarge input set form another PCl, but the first PCl merges with other clusters instead of the second PCl. The initial input set, simmedium and simlarge of last program volrend produce a PCl with a relatively short linkage distance ($d_{volrend\_PCL} = 0.216$), but this cluster need to merges with other clusters before generating a FCl with it's simlarge.

In Figure 8, we label the different input sets with disparate colors, i.e. simlarge is cyan, simmedium is green, simsmall is magenta and initial input set is red. We distinguish workloads with diverse shapes, and mark same workload with same sign in these figures. As shown in Figure 8(a)(b), we present the first four PCs retaining 76.70% of all characteristics. Figure 8(a) an 8(b) only show a subset of characteristics information about all workloads. For most workloads, their different input sets stay close to each other and follow the foregoing clustering analysis, such as barnes, radiosity and raytrace. In Figure 8(a), the input sets of fmm and cholesky are scattered, especially their initial input sets locate far away with their others. In Figure 8(b), the input sets of volrend, fmm radix and water_nsquared are discrete. For radix and water_nsquared, only one of their input sets stays long distance with others. The input sets of fmm are divided into two scattered groups. Each input set of volrend keep away from each other.

To further study the similarity between new workloads of SPLASH-2x and initial workload of SPLASH-2, we analyze separately three types of characteristics with PCA and retain more than 75% of original variables. We show the first two or four PCs of each type of characteristics, where retaining 75.64% of the instruction mix in Figure 8(c), 79.80% of memory access stride in Figure 8(d)(e) and 95.96% of cache size in Figure 8(f).

In Figure 8(c)∼8(f), the new input sets of most workloads keep very short distances with their initial input sets in different characteristic types, such as barnes, radix and raytrace. Different input sets of water_nsquared and water_spatial have similar characteristic distribution, such as the percentage of floating point operations range between 43.55% and 47.45%, 11.34% to 17.68% local memory read strides are equal 0, and inflection points are located in cachesize = 32KB or 64KB. The initial input set of cholesky are far away from its other input sets in these figures, especially in Figure 8(c). Because the percentage

of floating points operations and ALU operations of its initial input set are 31.67% and 15.47% respectively, the according values of other input sets range from 62.64% to 63.08%, and from 6.70% to 6.88%, respectively. There are also existing many similarities between lu_ncb and lu_cb, the percentage of ALU operations and branch instructions range from 18.94% to 20.56%, and from 10.76% to 10.91%, respectively. In Figure 8(d)(e), volrend's input sets are very distributed. Their most local load/store stride equal 0, percentages are 29.37%/44.5% of simlarge, 30.88%/20.88% of simmedium, 48.73%/21.49% of simsmall and 41.3%/43.54% of initial input set, respectively. In Figure 8(f), there are a half workloads are overlapped with others. For several workloads, their input sets locate in a acceptable scope like volrend etc. Initial input sets of radix, ocean_cp and fmm locate far to their new input sets in the first PC, because their have different cache miss ratio in various cache sizes.

From the above analysis, new input sets of SPLASH-2x workloads are following the characteristics of initial workloads well, except for cholesky and volrend. The new input sets of cholesky and volrend increase the diversity, but not represent effectively the original characteristics of initial input sets like others.

8. Related Work

There are a number of user-level implementation of the TCP/IP protocols. However, most previous works were done on micro-kernel operating systems where the user-level TCP/IP stack runs as a service on top of the micro-kernel. For example, both Thekkath et al. [38] and Maeda et al. [27] implemented a library-based user-level TCP stack for the Mach [1] operating system.

Since the mid-1990s, user-level TCP/IP stacks were developed for specific network devices in order for high performance network communication. HP implemented a user-level TCP/IP stack on Unix operating system for their Jetstream network [10, 11]. They also extracted the TCP/IP codes out of Unix kernel and ported them to user-level. U-Net [40]was a network interface supporting user-level TCP/IP protocols for parallel and distributed computing. Arsenic [37] was a user-level TCP/IP stack extracted from Linux kernel for a specialized gigabit network interface.

Apline [13] provided a user-level infrastructure which can run unmodified FreeBSD TCP/IP stack. Unlike HP's user-level stack which was implemented as a daemon, Apline works as library. More recently, Kantee [25]adopted the Runnable Userspace MetaProgram (rump) framework in NetBSD to run the BSD TCP/IP stack in a user-level process. In addition, some user-level TCP/IP stacks were designed and implemented from scratch, such as Minet [9].

Almost all the previous studies were motivated by leveraging the advantages of user-level implementation, such as flexibility, easing development of new protocols, customization and high performance with specific network devices. Like several benchmarks focused on datacenter infrastructure, their workloads run on various software stacks with different frameworks, including CloudSuite [14], BigDataBench [41], BigBench [18], AMP Lab Big Data Benchmark [2]. The goal of our work is to build an unified and portable benchmarking framework.

Since PCA and HCA are used to analyze program characteristics by Eeckhout et al. [12], they gradually become the common analysis techniques for workload similar behavior. Vandierendonck et al. [39] use PCA to determine the bottlenecks of machines that stressed by SPEC CPU2000 suite. Hoste et al. [22] realize PCA as one of data transformation methods to analyze performance difference. Phansalkar et al. [24, 30] use PCA to reduce the dimensionality of the data and remove the correlation among the metrics. With the inspiration of these works, we choose several representative characteristics and use these methods to analyze the similar behavior between SPLASH-2 and SPLASH-2x.

9. Conclusion and Future Work

We have presented the PARSEC3.0, a new version of PARSEC benchmark suite designed for studies of CMPs. In PARSEC3.0, we build the u-TCP/IP stack with the kernel TCP/IP stack of FreeBSD, and realize three new representative workloads based on u-TCP/IP stack. With network programs, PARSEC suite extends to network domain and increases the diversity further. We analyzed several characteristics of network programs with their initial workloads, such as parallelization, temporal locality and bandwidth demands. As another popular multithreaded benchmark suite for measuring performance of CMPs, SPLASH-2 has many similarities and differences to PARSEC. In order to facilitate researchers, we not only integrate the SPLASH-2 suite to PARSEC3.0, but also enrich the input sets of SPLASH-2 to SPLASH-2x. In the end, we analyzed the similarity between the initial input sets of SPLASH-2 and the new input sets of SPLASH-2x.

It is nearly impossible to implement an applications perfectly, so we will continue to improve existing workloads. The rapid evolution of processors will promote to build many emerging applications that have very high requirement for computing power, and we will add new workloads that can represent for emerging applications to new versions of PARSEC. Many typical programs have strong dependency on specific operating systems, but a mature benchmark suite should satisfy the platform independence and offer enough workloads for users to evaluate various aspects of their works. We will append new supports for I/O activity or file system effects. We will continue to enrich current benchmark suites, so that more covering research domains can be included.

10. Acknowledgments

PARSEC team will thank every contributor who has made contribution to any version of benchmark suite from original version 1.0 to current 3.0. The development of PARSEC is inseparable from their support. Many researchers have submitted patch that was included in PARSEC3.0 to perfect the function of benchmark suite, their names are Paul Keir, Rafael Asenjo, Joe Devietti, Joseph Greathouse, Chris Fensch, Darryl Gove, Alex Reshetov, Adam Morrison and Le Quoc Thai. In the end, we acknowledge the users who reported bugs or issues to us so that we can improve benchmark suite. This work was supported by the National Key Research and Development Program of China(NO. 2016YFB1000200) National Natural Science Foundation of China (NSFC) under grants No. 61420106013, 61221062, 61202062 and the Huawei-ICT Joint Research Lab.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," 1986.

[2] AMPLab, "Big data benchmark," https://amplab.cs.berkeley.edu/benchmark.

[3] C. Bienia, S. Kumar, and K. Li, "Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors," in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*. IEEE, 2008, pp. 47–56.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, 2008, pp. 72–81.

[5] C. Bienia and K. Li, "Fidelity and scaling of the parsec benchmark inputs," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[6] T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam, "Benchnn: On the broad potential application scope of hardware neural network accelerators," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 36–45.

[7] Cscope, http://cscope.sourceforge.net.

[8] L. Dagum and R. Enon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[9] P. A. Dinda, "The minet tcp/ip stack," *Northwestern University Department of Computer Science Technical Report NWU-CS-02-08*, 2002.

[10] A. Edwards and S. Muir, *Experiences implementing a high performance TCP in user-space*. ACM, 1995, vol. 25, no. 4.

[11] A. Edwards, G. Watson, J. Lumley, D. Banks, C. Calamvokis, and C. Dalton, *User-space protocols deliver high performance to applications on a low-cost Gb/s LAN*. ACM, 1994, vol. 24, no. 4.

[12] L. Eeckhout, H. Vandierendonck, and K. Bosschere, "Workload design: Selecting representative program-input pairs," in *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*. IEEE, 2002, pp. 83–94.

[13] D. Ely, S. Savage, and D. Wetherall, "Alpine: A user-level infrastructure for network protocol development." in *USITS*, vol. 1, 2001, pp. 15–15.

[14] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "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*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 37–48. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[15] FreeBSD, https://www.freebsd.org/internet.html.

[16] FreeBSD7.0, https://www.freebsd.org/releases/7.0R/relnotes.html.

[17] FreeBSD8.0, https://www.freebsd.org/releases/8.0R/relnotes.html.

[18] A. Ghazal, T. Rabl, M. Hu, F. Raab, M. Poess, A. Crolotte, and H.-A. Jacobsen, "Bigbench: Towards an industry standard benchmark for big data analytics," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 1197–1208. [Online]. Available: http://doi.acm.org/10.1145/2463676.2463712

[19] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy, "Megapipe: A new programming interface for scalable network i/o." in *OSDI*, 2012, pp. 135–148.

[20] HierarchicalClustering, https://en.wikipedia.org/wiki/Hierarchical_clustering.

[21] K. Hoste and L. Eeckhout, "Comparing benchmarks using key microarchitecture-independent characteristics," in *Workload Characterization, 2006 IEEE International Symposium on*. IEEE, 2006, pp. 83–92.

[22] K. Hoste, A. Phansalkar, L. Eeckhout, A. Georges, L. K. John, and K. De Bosschere, "Performance prediction based on inherent program similarity," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*. ACM, 2006, pp. 114–122.

[23] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," *Software Engineering, IEEE Transactions on*, vol. 17, no. 1, pp. 64–76, 1991.

[24] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring benchmark similarity using inherent program characteristics," *Computers, IEEE Transactions on*, vol. 55, no. 6, pp. 769–782, 2006.

[25] A. Kantee, "Environmental independence: Bsd kernel tcp/ip in userspace," *Proc. of AsiaBSDCon*, pp. 71–80, 2009.

[26] J. Lau, S. Schoemackers, and B. Calder, "Structures for phase classification," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004, pp. 57–67.

[27] C. Maeda and B. N. Bershad, *Protocol service decomposition for high-performance networking*. ACM, 1994, vol. 27, no. 5.

[28] B. Nichols, D. Buttlar, and J. Farrell, *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.

[29] PentiumPro, http://en.wikipedia.org/wiki/Pentium_Pro.

[30] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring program similarity: Experiments with spec cpu benchmark suites," in *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*. IEEE, 2005, pp. 10–20.

[31] C. Pheatt, "Intel® threading building blocks," *Journal of Computing Sciences in Colleges*, vol. 23, no. 4, pp. 298–298, 2008.

[32] Pin, https://software.intel.com/en-us/articles/pintool.

[33] I. Pratt and K. Fraser, "Arsenic: A user-accessible gigabit ethernet interface," in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 2001, pp. 67–76.

[34] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "Pin: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education: held in conjunction with the 31st International Symposium on Computer Architecture*. ACM, 2004, p. 22.

[35] J. Shlens, "A tutorial on principal component analysis," *arXiv preprint arXiv:1404.1100*, 2014.

[36] L. I. Smith, "A tutorial on principal components analysis," *Cornell University, USA*, vol. 51, p. 52, 2002.

[37] Specmail, http://www.spec.org/mail2009/.
[38] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing network protocols at user level," in *SIGCOMM*, 1993, pp. 64–73. [Online]. Available: http://doi.acm.org/10.1145/166237.166244
[39] H. Vandierendonck and K. De Bosschere, "Many benchmarks stress the same bottlenecks," in *Workshop on Computer Architecture Evaluation Using Commercial Workloads*, 2004, pp. 57–64.
[40] T. Von Eicken, A. Basu, V. Buch, and W. Vogels, *U-Net: A user-level network interface for parallel and distributed computing*. ACM, 1995, vol. 29, no. 5.
[41] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang *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.
[42] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2. ACM, 1995, pp. 24–36.