A Short Comparison of Scale-up and Scale-out*

Using new methodologies to get more from our benchmarks

Michael Sevilla
University of California, Santa Cruz
1156 High Street
Santa Cruz, California 95064-1077
msevilla@soe.ucsc.edu

ABSTRACT

When data gets too large, the focus shifts towards scaling to bigger systems, either by (1) scaling out (adding nodes to a system) or (2) scaling up (adding resources to a single node). This study presents the initial steps that we have taken in a direct scale-up vs. scale-out study. It is our hypothesis that scale-up systems will degrade differently than scale-out systems and we hope to quantify and explain these behaviors. In this paper, we select and apply current distributed systems benchmarks and workloads to a single node (scale-up) in an effort to compare the actual performance to their distributed systems (scale-out) counterparts. This paper discusses the limitations of previous studies, our long-term and short-term goals, and the progress we have made in (1) selecting applications, (2) porting applications, and (3) applying new methodologies to measure performance.

1. INTRODUCTION

When data gets too large, we scale to bigger systems, either by scaling out or scaling up. Traditionally, scale-out systems add more nodes to a system (i.e. distributed system) and scale-up systems add more resources (i.e. cores, memory, etc.) to a single node. It is understood that scale-up and scale-out have different complexities, bottlenecks, and architectures and in this study we try to paint a clearer and more detailed picture of the scaling landscape by having a more expansive methodology and using modern hardware and profiling tools.

Previous scalability studies [35, 24] enumerate the trade-offs between scale-out and scale-up but the studies use outdated hardware, methodologies, and benchmarks. In the past (early 2000s), scale-out architectures garnered extensive attention in both the research and the business communities for three reasons: (1) cost, (2) non-linear system scaling, and (3) interoperability issues. At present, we are seeing more and more challenges for scale-out architectures, resulting in workload specific distributed architectures, optimizations made at the application level, and incredible complexity/unpredictably at the system maintenance level (this is discussed in more detail in Section §2). In light of this, we propose a new scale-out vs. scale-up comparison to re-examine the benefits and drawbacks inherent in each architecture.

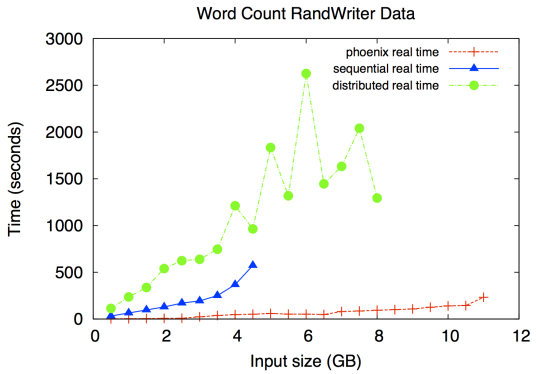


Figure 1: Using new methodologies for selecting and porting applications, we can compare different versions of the same word count application: a scale-out version (Hadoop), a scale-up sequential version (C++ libraries), and a parallel programming version (Phoenix). The benchmarks are run on increasing amounts of data until the program crashes.

To do this, we re-examine scale-up architectures in the context of their scale-out counterparts. We will apply current distributed systems benchmarks and workloads to single nodes and observe their reactions but the main contribution of this study is the formulation and preparation of our own methodologies that improve upon previous studies.

To address the narrow methodologies of previous studies (discussed in Section §3), we provide a more expansive methodology for examining slowdowns and overheads on single node systems with sufficient hardware. Our methodology is to incrementally vary the amount of data and methodically

*Project completed as part of CMPS290S: Big Data Systems.

vary the amount of resources. This exercises the inherent relationship of system speed to the number cores, amount of memory, and size of the data instead of just processing power. To address the out-dated results, mainly the identified slowdowns and bottlenecks of previous studies, we use modern hardware, software, and tracing/profiling tools.

We have selected representative applications and ported two of them between scale-out and scale-up in a fair manner. In this context, an application is a program that performs a job. Our experiments incrementally increase the amount of data that needs to be processed and we use our expansive methodology to contribute 3 perspectives: timing component breakdowns (Figure 1), CPU clock cycle time (per symbol), and the effects of various core to memory ratios

These perspectives provides a more complete view of the system response to the applications and workloads. Armed with a better understanding of what the system is actually doing, we can begin to quantify the true tradeoffs of scale-out and scale-up.

The main contributions of this paper are: (1) the expansive methodology, (2) a way to select and port applications between scale-up and scale-out, and (3) new perspectives for viewing performance.

Section §2 gives a short background on the recent scaling trends in academia. Section §3 delves into the limitations of previous studies and Section §4 explains the steps we have taken to address those limitations. Section §5 describes how we ported the applications and Section §6 discusses our results and what we can learn from our expansive methodologies. Section §7 discusses the related work done on both single node and distributed systems and Section §8 concludes.

2. BACKGROUND

To truly understand the differences between the two architectures, in the next 3 sections, we lay out how the academic community has evolved its systems for scale. We start with the reason the community moved towards scale-out in the past, then we enumerate some of the difficulties the community is facing today, and, finally, we predict a trend for the future.

Note that this traces the lineage of system design in the *academic* community. Work done in the business and high performance computing communities, like the Cray XK6 [15] and the IBM BlueGene/Q [16], are outside the scope of this paper because of the price range and scale.

The past: a push towards scale-out

The scale-out architecture garnered extensive attention in both the research and the business communities as a result of excellent work produced by web-based companies [19, 7, 22, 10, 25], the most popular being the MapReduce [9] framework. They built an infrastructure on many commodity servers because, at the time, many commodity servers were cheaper than a powerful single node and because their workloads (i.e. graph processing, iterative programming, etc.) could be easily ported to a distributed framework.

As a result, the scale-out architecture implementation, efficiency, and performance were examined thoroughly because the architecture offered new and exciting problem for researchers to explore. Around the same time, the field began to notice significant challenges for single nodes, mainly: (1) system speeds stopped scaling linearly and (2) interoperability issues

System speeds were not doubling according to Moore's law anymore because of the "memory wall" [38]; memory transfer rate (speed) and onboard connections (physical space) limited the performance of the system as a whole. As a result, hardware components do not scale in lock-step and CPU speeds outpace memory bandwidth. When a customer needed twice as much performance, the customer could no longer wait a year for a system that was twice as fast.

Current operating systems were not designed for the rapid injection of additional resources. Some work has been devoted to designing new types of systems [4, 31, 37, 3, 2, 20] and small tweaks to these systems have shown improved scalability but these improvements required extensive system knowledge and time.

The present: the difficulties of scale-out

As scale-out matured, developers realized scaling out has many of its own challenges, especially in regards to the limited programming model exposed by the MapReduce framework and overall system maintenance. If the application is formatted with two distinct steps, it fits perfectly in the MapReduce framework but, of course, this isn't always the case. Also, maintaining many computers and their interconnects is more difficult than maintaining a single node because physical limitations and additional node behaviors must be monitored.

So, despite the advantages of scale-out, like fault tolerance, work distribution, and performance, the difficulties forced a new wave of systems focused on workload specific architectures and application space optimization. Unfortunately, both these approaches introduce a new degree of complexity because the developer must either develop a distributed system for a specific computational pattern or change the program to accommodate varying degrees of parallelism, synchronization, and compute power.

For example, distributed systems, like MapReduce [9] and Dryad [19], confine the user to specific inputs and workloads, so specialized systems like Pregel [22], Spark [40], and S4 [25], were developed to handle graph processing, iterative/interactive, and steam processing computations, respectively. This complicates both the programming model and the system complexity and the only available alternative is to make optimizations at the application level, as seen in concurrent programming, parallel databases, and programs that require resource management.

The future: a re-examination of scale-up

We predict a new focus on scale-up because it potentially has the ability to:

- ✓ simplify[1] the programming model

[1]i.e. no parallelism, synchronization, and message passing

- ✓ automize dynamic optimization
- ✓ evolve with hardware architectures

The benefits of a scale-up architecture and programming model fuel a curiosity for a true scale-out vs. scale-up comparison. The original studies and conclusions drawn in the early 2000s may have changed with the wave of new hardware and software. We do not know how modern hardware speeds and benchmarks affect performance and now that big data systems are comprised of hundreds of thousands of servers, the cost benefit may not be so heavily skewed towards distributed systems on commodity hardware.

3. MOTIVATION

Our hypothesis is that new bottlenecks and slowdowns will be identified if the direct comparison between scale-up and scale-out is detailed enough. More specifically, we suggest that a new study is necessary because previous studies used (1) narrow methodologies and (2) out-dated hardware and software.

3.1 Scalability benchmark methodologies

The performance of applications on scale-out and scale-up platforms have been studied extensively for variable parameters. Usually, the parameters for distributed systems are the number of nodes (e.g. [31, 3, 33, 37, 28, 39, 34, 35, 4]) and the workload types (e.g. [8, 40, 22]) while the parameters for scale-up systems are hardware contexts, like the number of cores and threads (e.g. [30, 14, 18, 19, 7]). Most studies on existing systems and novel system architectures change one of these parameters, measure performance, and draw conclusions from their observations. We contend that varying these parameters is not enough and different hardware configurations should be applied in all scalability benchmarks.

This reasoning is fueled by two trends: (1) it is common to tune hardware configurations for a given workload and (2) newer applications look to dynamically change their architecture to accommodate different workloads. For example, in industry, system designers are often called upon to manually configure high-performance systems, showing that the right hardware specifications matter. In academia, resource provisioning in cloud environments has trended towards automated hardware resource distribution [27, 36, 42, 29, 23] and runtimes have explored adapting to accommodate application properties (e.g. [6, 34]). These trends prove that tuning hardware configurations is advantageous, leading us to conclude that varying *other* parameters may be necessary to get a good idea of system performance.

An adversary to the aforementioned arguments, that system tuners and dynamic architectures prove a need for more expansive experiment parameters, might argue that ideal performance varies for different workloads. This is true but does not discount the flaws of current methodologies - it would still be valuable to run benchmarks with larger parameter variability to observe the behavior. It is has been taken for granted that the ideal core-to-memory ratio is between 2GB and 4GB but this is based on blogs and dialogue; the problem is that rigorous studies, often pigeon-holed by the academic community's selection of applications that are deemed appropriate, keep too many variables static.

3.2 Out-dated hardware/software

Work on profiling Linux has determined that lock contention and synchronization are the performance bottlenecks in large single nodes [4, 33]. As a result, the field has diverged into two scaling camps. One camp has progressed towards new scalable architectures and operating systems, like Barrelfish [31], the factored operating system [37], Corey [3], and Cerberus [33], while others attempt to identify bottlenecks and fix them [4, 5]. Many of these systems and modifications are based off the notion that operating systems cannot scale when there are too many resources (i.e. cores).

We contend that the hardware/software landscape has changed since these studies, especially in regards to the most recent comparison between scale-out and scale-up. In the most recent scale-out vs. scale-up study, the scale-up system had 32 hardware contexts (8 × dual-core processors) and 32GB of RAM while the scale-out system was comprised of 14 nodes [24]. Also, the benchmark consisted of just one workload: Apache's Nutch. New hardware and software contexts change access and computation patterns in big data computing making the raw results of this study (no t necessarily their methodologies or their process) almost obsolete.

To conclude, we feel that a more expansive study will reveal new bottlenecks and should help explain some of the reasons that the aforementioned systems have not "swept the nation". To test our hypothesis, that current benchmarks and bottleneck studies are not sufficient, we plan to do a small case study directly comparing scale-up and scale-out.

4. METHODOLOGY

We entertained the thought of conducting a core-to-memory-to-data ratio study on scale-up systems. We would vary each component ({core, memory, data}) and observe optimal configurations for each workload. Upon further review, we realized that the problem is well-known and far too general. In today's scale-up solutions, systems are closely designed and finely tuned to the requirements of the workload and applications. Furthermore, characterizing applications along an axis that all could relate to is difficult, although others have tried [41, 21]. This forced us to re-examine distributed systems, since this has been done and accepted in the field. Eventually, we arrived at the current proposal: an empirical study comparing scale-up and scale-out.

A birds-eye view of our current methodology is:

performance grid	→	small experiment	→	select apps
(long-term goal)	→	(short-term goal)	→	(first goal)

Our goal is to overcome the methodology and hardware/software limitations (discussed in §3.1 and §3.2, respectively) of current scalability metrics. To address these limitations, we measure the system in three ways: (1) timing component breakdown, (2) CPU clock cycles (per symbol), and (3) behavior under various core to memory ratios.

The timing component analysis provides a macro level view of the system and shows the amount of time the application spends in different contexts (kernel, user, total). The CPU clock cycles per symbol illustrates the amount of time spent

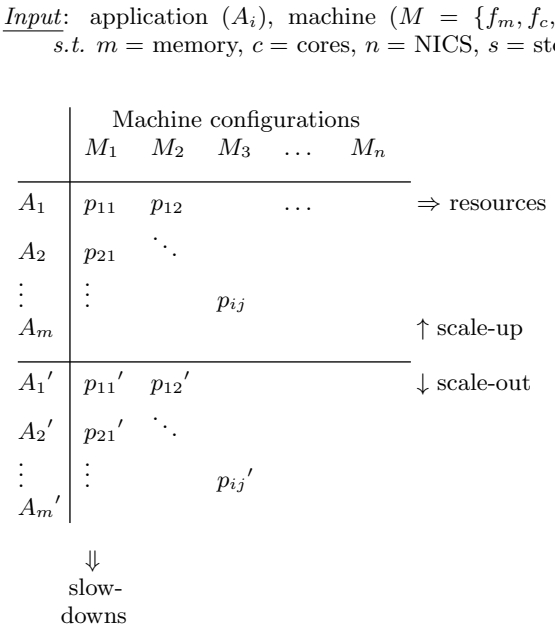


Figure 2: Our long-term goal is to create a performance grid, which can help us (1) create a cost function and (2) identify the differences between scaling out and up.

in a specific symbol, like a function call or image binary. In these studies, we took the highest event at the time right before a crash. Finally, the various core to memory ratios have the ability to show the true relationship between hardware resources, data, and performance. A description of how each of these metrics are collected is discussed in Section §5.

We started by directly trying to achieve the long-term goal and realized that we needed to break the methodology into smaller parts. The performance grid (§4.1), the small experiment (§5.1), and the application porting (§4.2), are the resulting methodology components and each have their own set of challenges. This paper only discusses the application porting and the beginning of the small experiment but the whole methodology is presented here for further motivation.

4.1 Long-term goal: performance grid

Ultimately, we hope to identify the optimal hardware ratios for scale-up and scale-out systems. We take the same notion of studying ratios for the amount of memory, cores, and data as our methodology and try to explain how to achieve maximum performance. To do this, we will construct a performance grid, as shown in Figure 2.

The grid is composed of applications (A_i), machine configurations (M_i), and performance measurements (p_{ij}). Applications are grouped based on their architecture (scale-out or scale-up) and on a loose characterization (e.g. CPU bound or memory sensitive). For example, Figure 2 could have two scale-up, CPU bound applications, denoted as A_1 and A_2, and equivalent applications ported to scale-out ,denoted as A_1' and A_2'.

What the performance grid gives us

This grid would give us a better understanding of how different resources affect performance. The rows correspond to the behavior for different resource ratios (core-to-memory-to-data). The columns help identify behavior for different slowdowns based on application demands.

From this table, we would be able to draw relationships between the performance metrics. By observing patterns and trends, we could make two contributions to the scaling discussion: (1) a cost function and (2) a comparison of the services offered by each scale technique.

A cost function would help system designers and dynamic resource allocators provision resources effectively. The performance grid would allow us to correlate the results in different cells, making it easy to determine which configurations have relationships and which applications work better on which platforms. By observing trends, it might be possible to derive a cost function, such as

$$\text{cost} = w_m f_m + w_c f_c + w_n f_n + w_s f_s$$

where w_i is the weight for a given resource, f_i is the amount of the resource, and m = memory, c = cores, n = NICS, and s = storage. A comparison of the scale techniques gives us another dimension of information for system designers. More specifically, by looking at the performance grid, we can see the cost of losing a specific component if we choose one scaling approach over the other. For example, from the performance grid, we could identify fault tolerance as a component in scale-out that is missing in scale-up and attribute a sacrificial cost of the scaling decision by examining performance.

Unfortunately, selecting applications to compare the architectures is not trivial.

4.2 Short-term goal: select applications

Constructing the performance grid and running our small experiment presents logistical challengs. We need to figure out which applications to select and a medium for porting these applications. This process is extremely difficult because our comparisons and benchmarks must be fair, feasible, and representative.

4.2.1 Which applications?

When selecting the applications for our scaling comparison, we need a process that is representative and feasible. To be representative, our applications need to cover a wide array of workloads and execution patterns while stressing different system resources. To be feasible, our applications must be relatively trivial to implement - characterizing and developing applications is beyond the scope of this paper. Hence, it would not be ideal to produce work that has already been done.

To start, we looked at the academic community's two disciplines of benchmarks: (1) operating system benchmarks and (2) distributed systems benchmarks. Operating system benchmarks, such as the Standard Performance Evaluation Corporation (SPEC) benchmarks [17], provide a wide range of applications that stress various system components, such as CPU, I/O, and network functionalities.

Despite the variety and depth of the operating system benchmarks, we decided to use distributed system benchmarks because they represent the state of the research community and are easier to port. A majority of scale-up solutions are highly specialized while distributed systems have more of an academic presence and if it is accepted by the community then it ought to be good enough for us. Also, it is easier to port from a distributed systems benchmark to a single node benchmark because the problems are well known. For example, imagine porting a CPU benchmark to a distributed system - how would that work?

We chose the HiBench [18] test suite because it was simple, yet representative. We use a subset of the benchmark: word count, sort, Terasort, PageRank, and Nutch.

SWIM [8] is a benchmark that samples a trace to build a workload because the authors contend that synthetic benchmarks are inaccurate and unrealistic. They criticize HiBench, stating that it fails to capture job mixes and arrival rates. Unfortunately, SWIM does not fit our needs because the user must supply traces and the output is specific MapReduce jobs.

4.2.2 Port applications?

In addition to selecting the correct applications, we need a way to port them between the two scaling architectures in a fair manner. To be fair, our porting must stress the differences between the two architectures, not the applications themselves. To be feasible, like in Section §4.2.1, our applications must be trivial to implement. With these goals, we ran into two questions:

1. *How do we get equivalent applications?*
2. *Should our application porting be an "optimized solution" or the "dumbest thing possible"?*

To answer question (1), we came up with two different solutions: (a) port for methodology, or (b) port for functionality.

The first approach, porting for methodology, fixes the application methodology and removes the distributed mechanisms. For example, we would take a distributed, scale-out algorithm, work hard to understand how it works, and remove the fault tolerance, work distribution, and load-balancing code. The advantages of this approach is that it is extremely fair and we would be measuring the underlying system architecture (scale-out or up) and not the application. Unfortunately, porting for methodology has many drawbacks, mainly that the process is difficult, involved, and requires intimate knowledge of the application. Furthermore, it is not entirely fair to compare a distributed algorithm to a single-node algorithm because the distributed algorithm was designed to run on multiple machines and hence, its optimizations might not be optimal for a systems where everything is local.

The second approach, porting for functionality, fixes the application functionality and uses the best and most intuitive solution for the given architecture. For example, if the task is to sort, we would compare a single-node algorithm (like Quicksort) on a scale-up system to a distributed systems algorithm (like Hadoop's sort) on a scale-out system. The advantages of this approach is it is feasible because these approaches are all well-documented (and usually the code is already available). The disadvantage of this approach is that it is not entirely fair because we are comparing two wildly different implementations that stress different resources.

The approaches for question (2) have similar drawbacks. If we port for the "dumbest thing possible", our comparisons would not be fair. On the other hand, porting for an "optimized solution" is not feasible since it would take many iterations of twiddling concurrency and synchronization primitives to get high performance (as is customary in the field).

Luckily, a system out of Stanford called Phoenix [28] answered all these questions (and more).

Background: Phoenix

The Phoenix systems are implementations of MapReduce for high performing applications on shared-memory systems. There are three iterations of the system since its introduction in 2007 and the changes they made are critical for both our benchmark and application motivation, methodology, and implementation.

Phoenix [28] is an implementation of MapReduce for shared-memory systems, consisting of a programming API and an efficient runtime system. Essentially, the libraries convert MapReduce programs to multicore and multiprocessor systems (including shared memory systems). Hence, the Phoenix system gives us a way to port applications for BOTH methodology and functionality, essentially achieving the desired goals of being fair and feasible! A component by component breakdown is shown in Figure 3.

Phoenix is motivated by programming model trends and their evolution to achieve scalability. The trend has been:

sequential → parallel → distributed

Phoenix is a reversal back to something we already had:

parallel ← distributed

This raises a fundamental question:

> *Why are going back to parallel programming on single nodes?* (- Ike Nassi)

The answer is that parallel programming is difficult - the programmer must manage:

- concurrency: synchronization, messages, locks, etc.
- resources: shared-memory, thread/process, locality, etc.
- applications: re-tuning, portability, scalability, etc.
- significantly larger code

	MapReduce	Phoenix
work distr.	master node worker nodes	parent process threads ∈ core
communication	network i-keys ∈ HDFS	shared-memory i-keys ∈ L1 cache
fault tolerance	heartbeat remote re-execute	timeout local re-execute
combiner	∈ node after map	∈ thread after map
API prog. model	`string` functional data ‖-ism	`void *` functional data ‖-ism

Figure 3: The differences between Phoenix and MapReduce. Phoenix is an open-source implementation of MapReduce for shared-memory systems.

The Phoenix system marries two important programming components: a practical *programming model* and an *efficient runtime*. As such, it provides:

- ✓ simplicity
- ✓ automatic parallelization
- ✓ automatic concurrency
- ✓ flexibility (i.e. programmer can optimize)

Phoenix 2 [39] is optimized for large-scale NUMA systems and Linux x86_64. The runtime must accommodate locality for non-uniformity, scalable/in-memory data structures for parallel computations, and OS behavior for memory management and I/O. The authors realized that, because of memory latency, poor locality, and contention, the system actually *slows down* when more threads (across multiple chips) are injected. Even with the runtime optimizations made at the application level, hardware context scalability was limited by:

- idling for data pages: memory allocation + I/O
- kernel code: `sbrk()`[2] and `mmap()`[3]

Thoughts: This study only looks at scaling the number of threads but does not look at the effects of additional data and memory. Varying different parameters would alter the core to memory ratio (currently at 2GB/thread for 256 threads). The in-memory data structure are crucial to performance, therefore, it seems intuitive that memory speed and size makes a difference. Cognitive dissonance strikes me because their data sets are relatively small (they appear to fit in memory) - so maybe more memory won't help. Despite this, it would be interesting to see what happens; will we see the excessive page faults and memory allocator pressure with additional memory?

Phoenix++ [34] is optimized for modularity, extensibility, and workload characteristics. Phoenix and Phoenix 2 have inefficient key-value storage, ineffective combiners, and poor task optimizer overheads. Phoenix code showed poor performance so users began bypassing the Phoenix pipeline by overloading the `map()` and `reduce()` functions. To compensate for this, Phoenix++ allows the user the modify the system for specific workloads by introducing:

- containers: tune framework for i-key distribution
- container objects: accommodate many <k, v> pairs
- modular functions: allow functionality swapping

Thoughts: This is interesting because they felt a need to modify the runtime for a workload. It is moving away from the idea that Phoenix is general enough to run workloads that are commonly run on Hadoop. Their analysis shows better scalability than Phoenix 2 but they are still using an out-dated kernel, far too many threads, and an even smaller core-to-memory ratio (0.5GB per core). It would be interesting to redesign their 3D performance figure with different hardware resources on each axes (i.e. x axis is number of cores, y axis is the amount of memory, and z axis is the amount of data). Such a figure would clearly show the effects of additional data and memory and the relationships between the system as a whole.

Of course, this methodology is not perfect. An adversary might say "Of course Phoenix will perform better, it replaces all the slow components of Hadoop, like the network and disk, with in-memory data structures, and it employs optimizations with prefetching, cache replacement hints, and hardware compression!". Well, this is true, but the point of this study is not to validate or discredit Phoenix. The point is to be representative, fair, and feasible in selecting and porting applications for the scalability study. We don't know what bottlenecks or slowdowns we will hit but we want to minimize the impact of the applications.

5. IMPLEMENTATION

Using Phoenix, we can fill in the "≡ functionality" and "≡ methodology" columns of the table in Figure 4. Recall that to select the applications, we use HiBench, the Hadoop benchmark, and to port the applications, we use Phoenix, the MapReduce API/runtime for shared-memory systems.

5.1 Experimental setup

In our experiments we will continually add data until the application or operating system crashes, as shown in the pseudo-code in Figure 5. For each run, we will continually add data in 0.5GB chunks. On the first iteration, we will

[2]bottlenecks `mtmalloc()` because the single user-level lock kept other threads from expanding the process's data segment while per-address space locks protected

[3]serialization as chip boundary was crossed

	scale-out	scale-up		
	Hadoop (3 nodes)	≡ functionality	≡ methodology	Hadoop (1 node)
word-count	✓ `WordCount.java`	✓ `wc-seq.cpp`	✓ `wc.cpp`	✓ `WordCount.java`
sort	✓ `Sort.java`	✓ `sort-seq.cpp`	✓ `sort.cpp`	✓ `Sort.java`
Terasort	✓ `TeraSort.java`	✗ `tsort-seq.cpp`	✗ `tsort.cpp`	✓ `TeraSort.java`
Page Rank	✓ Hama	✗ `pg_rank-seq.cpp`	✗ `pg_rank.cpp`	✓ Hama
Nutch	✓ `SolrIndex.java`	✗ `index-seq.cpp`	✓ `SolrIndex.java`	✓ `SolrIndex.java`

Figure 4: The applications for the selected benchmarks are based on HiBench. The applications are ported using Phoenix. ✓ indicates that the application is already implemented (either by me or someone else), ✗ means that I still need to implement it. Text in magenta are links to implementations that I can use as a reference.

```
foreach application
    while(!stressed)
        execute()
        measure_performance()
        ++data
```

Figure 5: Our experiments will continually add data until the system crashes or slows to a halt. This shows a wide range of behavior based solely on the data.

process a file of size 0.5GB, then a file of size 1GB, then a file of size 1.5GB, and so on. After a fresh reboot, each experiment is run once without profiling and stat collecting to warm up the OS and hardware caches and experiments are run multiple times (no average is taken because the results seemed similar enough). The experiments have different formats to the input data and is discussed below.

To start filling in the performance grid, we need a small-scale experiment to determine if our methodology is sound. We choose to start with two simple systems (using Linux Ubuntu 12.04):

M_1 : scale-up: single node with 8 cores, 12GB RAM

$M_1{}'$: scale-out: 3 nodes each with 4 cores, 8GB RAM

This should give us some initial results, including two performance measurements (data points), new bottlenecks, and different memory thresholds for scaling-out or up. We need to pick applications and port them for the opposite architecture.

Data

For the two applications we actually port and benchmark (sort and word count), we use data generation techniques from the Hadoop community. The generated data is random bytes, known as a sequence file, written to HDFS and then transferred to the local machine. The data for word count is generated using Hadoop's `RandomWriter` program and the data for sort is generated using Hadoop's `Teragen` program.

We generate 12GB of data because our system has 12GB of RAM. This should be more than enough to crash the program or operating system because the operating system takes over 1GB of RAM and other system structures reduce the amount of free RAM of a freshly rebooted system (according to the `free` command) to be a little over 10GB.

5.2 Tools for measuring performance

Recall that we have more expansive methodologies for measuring performance than previous studies:

1. timing component breakdown
2. CPU clock cycles (per symbol)
3. behavior under various core to memory to data ratioRs

The timing component breakdowns uses the Linux `time` command, which outputs three values: `real`, `user`, and `sys`. `real` is the wall-clock time and includes the time sucked up by other processes or the time the process is spent blocking. `user` is the CPU time spent in user-mode within the process and `sys` is the CPU time spent in the kernel within the process. This command gives us a macro level view the amount of time the application spends in different contexts (kernel, user, total).

The CPU clock cycles per symbol are collected with OProfile, a system wide profiler of Linux. It uses hardware performance counters, special registers built into processors, to monitor interrupts like processor cycles, TLB misses, memory references, cache misses, and branch miss predictions. By monitoring the CPU clock cycles, we can see the amount of time spent in a specific symbol, like a function call or context. In these studies, we took the highest event at the time right before a crash.

Finally, the various core to memory ratios are achieved by "turning off" different resources. We can turn off cores by echoing 0 into the `/sys/devices/system/cpu/cpu2/online` and we can limit the amount of detectable (from Linux) memory by modifying `/etc/default/grub` and then updating the grub image. By changing the amount of resources in our single node, we can see the true relationship between hardware resources, data, and performance.

5.3 Porting for methodology (Phoenix)

Porting applications using Phoenix is very simple, given that the program has steps that can be broken into Map and Reduce tasks. The algorithm of the sort program I wrote in Phoenix is based off of MapReduce's `Sort.java` from the example jar. In the comments, they say:

```
1  class SortMR : public MapReduceSort<...>{
2     // map(): execute this function at each node
3     void map(data_type s, map_container out){
4        wc_word word = { s.data+start };
5        emit_intermediate(out, word, 0);
6     }
7     // split(): split the input file into words
8     int split(wc_string& out){
9         out.data = data + splitter_pos;
10        out.len = end - splitter_pos;
11    }
12    // sort(): establish a partial ordering
13    bool sort(keyval a, keyval {b){
14        return a.val < b.val || ...;
15    }
16 }
17 int main() {
18    // open and memory map the input file
19    fd = open(fname, O_RDONLY);
20    fdata = mmap(..., fd);
21
22    // prepare the result vector and runtime
23    std::vector<SortMR::keyval> result;
24    WordsMR mapReduce(fdata, ...);
25
26    // start the Phoenix runtime
27    mapReduce.run(result);
28 }
```

Figure 6: The Phoenix sort program implements an identity mapper and reducer; this allows the Phoenix runtime to sort the input.

```
/**
 * This is the trivial map/reduce program
 * that does absolutely nothing other than
 * use the framework to fragment and sort
 * the input values.
 **/
```

Essentially, the mapper and reducer are just identity mappers and identity reducers; they do nothing except pass their inputs directly to their outputs. To port this to Phoenix, we use this algorithm design and the Phoenix word count template (`wc.cpp`).

The Phoenix runtime requires that the user specify the `map()`, `split()`, and `sort()` API functions. The word count program is already ported and distributed as one of the Phoenix tests. Sort is implemented based on the word count template except we change the `emit_intermediate()` function (line 9 in Figure 6) to act as an identity mapper; meaning it emits whatever key it ingested and lets the runtime sort the output. The reducer for the sorting program is also just the identify reducer and Phoenix does not require the user to fill in the `reduce()` API function.

Experience using Phoenix

Originally, I tried to use Phoenix-C (April, 2007) since I was more familiar with C. Everything compiled fine after I changed `-pthread` and I ran some initial benchmarks but `word_count` would `segfault` when splitting the data and checking that the length of the split ended in a ' ' or '\t'. This happened because it was using the Linux `mmap()` system call to map the file into RAM. This gives faster accesses (at the cost of increased thread interaction/communication) but limited the size of the acceptable file. You see, `mmap()` takes an `int` for the size argument - naturally, this is limited to 2^{32} values, which restricts values to $-2147483648 \leq x \leq 2147483648$. As a result, the largest memory mapped device (in our case, a file) is 2GB. 64-bit Linux instances should allow largely memory mapped devices, but I did not verify this.

I started working with Phoenix++, which boasts a C++ implementation, interface, and templates to reduce the code the user must write (in addition to the type safety/performance). To get it to work, I had to do add a couple things, which are worth noting in case some other poor, lost soul wants to follow in my footsteps:

- **TR1 compiler**: install Boost
- **Defines.mk**: `-pthread` instead of `-lpthread` (linking issue)
- **Old G++ support**: `-fpermissive` flag (variable declaration issue)

5.4 Porting for functionality

To port for functionality, we use approaches that naive programmers would try. One of the main attractions of scale-up, as noted in Section §2, is the simple programming model so we try to preserve this to do the simplest thing possible. It should be noted that we memory map the file in both cases, to get comparable performance to the Phoenix implementations.

For word count, we insert the words into an unordered map and update the values when the same word is parsed. The results are sorted by copying all the words into an array of pairs and we use the C++ `qsort` to sort the results and print the top 10. We choose to sort the entire array since this is similar to the distributed algorithm (which sorts everything). Note that we could use an alternate hash map, that uses a b-tree, but we want to be as simple and brainless as possible.

For sort, we did almost the same thing, but skipped the step of storing the data in the intermediate unordered map. We store everything directly in an array, meaning that we are getting duplicates. This may seem counterintuitive, but MapReduce and Phoenix sort duplicates so we try to preserve that. Once again, we used C++'s `qsort` to sort the values.

6. RESULTS

Our results show that our expansive methodologies reveal a great deal about what is happening in the underlying system. We benchmark two applications, word count and sort, over a range of data between 0.5GB and 12GB.

Word count: timing component breakdowns

Figures 1 and 7 show the results from the Linux `time` command.

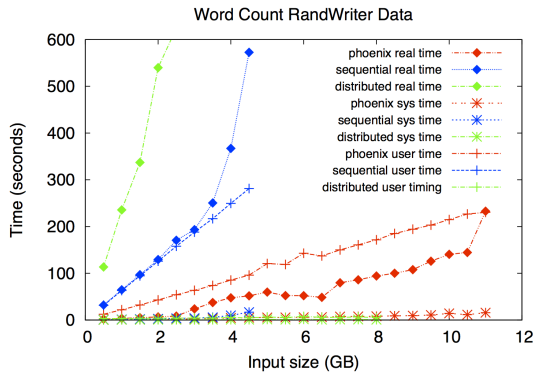


Figure 7: Using the Linux `time` tool to determine where execution spent the most time, we can see that (1) Phoenix achieves high parallelism, (2) sequential word count does not scale with `sys` time, and (3) Hadoop can be bottlenecked more by the neighboring nodes than the network itself.

Figure 1 plots a direct comparison between scale-out and scale-up. The scale-out system is Hadoop running on a 3 node cluster. The unpredictable latency at 4GB can be attributed to a number of factors. One explanation for the unpredictable performance is that the system is configured with a suboptimal parameter. The system is configured with the default settings but the block size, partitioning, and HDFS can be adjusted with the Hadoop configuration files. Another possibility for the wild performance is the network because as packets become larger, they become more prone to variable behavior - but this is unlikely since the nodes were on a LAN on neighboring racks. A more likely explanation is that one node is slower than the others and ultimately slows down the entire job. If one node is slower, it imposes a global barrier for the other map and reduce tasks. Yet another possibility is that the limited number of task trackers that are spawned on each node is underutilizing the system, leading to increased idling. Finally, Hadoop was not designed for less than 10 nodes so our 3 node cluster is not realistic in the slightest. We predict that we will see real predictable performance measures if we scale the system to the numbers that it was intended for.

These performance measures help prove a point: it is difficult for a novice distributed systems programmer to configure and manage these scale-out systems. First, there were difficulties scrounging together 3 nodes because the cluster is in a state of disarray (due to NFS problems with Ubuntu 12.04). Second, many of the nodes are getting older and parts are starting to fail. Before running the jobs, we had to look in a spreadsheet to figure out which nodes had bad RAM, broken network interfaces, or problems booting, etc. Finally, the Hadoop interaction with the file systems is difficult to interface. For example, in one weekend alone, I spent time trying to figure out why the namenode failed to format (temporary directory issue), why the task trackers would not start (can not kill Hadoop jobs in mid-flight because they hold open sockets and data structures), and inconsistent temporary directories (need to move them off NFS).

	Data GB	Time secs	Error → Event
wc.cpp	11.5	232.82	cpu throttled → `int_idle()` (+10%)
wc-seq.cpp	4.5	572.75	bad allocation → `scan_swap()` (+8%)
sort.cpp	1.5	208.11	OOM; kill → (+?%)
sort-seq.cpp	4.75	830.46	OOM; kill → `scan_swap()` (+20%)

Figure 8: Using the system's hardware performance counters, we can see that the errors and most frequent events vary according to the application and its behavior. The Phoenix word count and sort are memory intensive, so we see in increase in the number of times that swap memory is scanned - this is an artifact of paging and may be a bottleneck for these programs.

We even found a bug with Hadoop starting and putting the NFS cluster in death spirals when trying to start the namenode and data nodes, resulting in 100% utilization on our gateway and nodes infinitely looping in a request states. In short, maintaining a cluster is not fun.

Figure 7 breaks down the latency into the time (based on CPU cycles) that the program spent in each context (`real` is the total wall-clock time, `sys` is the kernel time, `user` is the user-space time). The `real` times are the same as those in Figure 1.

Three things in Figure 7 are interesting and counterintuitive. First, the Phoenix `user` time is greater than the `real` time. This is because the `user` time is based on CPU cycles. Since the work is being distributed to multiple cores, the aggregate CPU time spent in user-space is greater than the actual time it takes to complete the job. Second, the sequential word count latency increases exponentially despite the lack of exponential behavior in the user/kernel-space times. We know that CPU cycle time is different than wall-clock time, but it is interesting that we do not see a corresponding exponential trend - perhaps this is due to the system throttling itself. Finally, most of the time for Hadoop is NOT spent in the networking stack, as previously assumed. Either this node is significantly faster than the other nodes or one node is truly slowing down the system as a whole. This seems like the most likely explanation because it appears that the node we timed idles, yet the computation takes a long time.

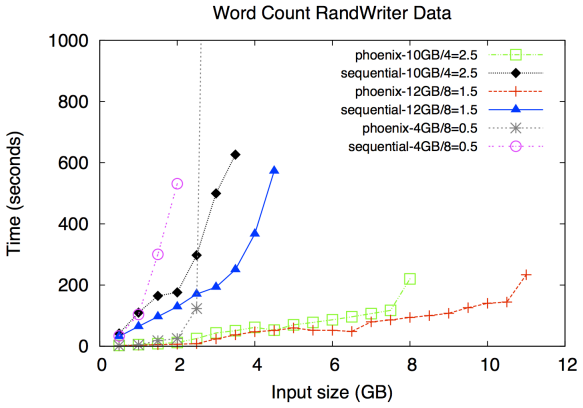
Word count: CPU clock cycle time

When the system crashes or slows to a crawl, we examine the hardware performance counter data to determine where most the time is being spent. We compare this result with a normal execution on smaller data. The results are displayed in Figure 8.

From the results, we can see data definitely has an effect on performance. Essentially, when the system runs out of memory to process and store the data, bad things start to happen. The common event that happens in both sequential programs is increased scanning of the swap space (`scan_swap()`). This makes sense since the system is low

run	$\frac{\text{memory}}{\text{core}}$ Ratio	Cores, RAM
M_1	0.5	4, 8GB
M_2	1.5	8, 12GB
M_4	2.5	4, 10GB

(a) Our technique for modifying the core to memory ratios is to methodically increment ratios and maximize the resources.



(b) The latencies for the word count program on various resource ratios.

Figure 9: Varying the core to memory ratios, according to Figure 9a, gives different performance curves for increasing data. Figure 9b shows that the relationship between cores, memory, and data has an appreciable effect on performance. Curves with similar slopes and performance can help indicate an over-provisioning of resources.

on memory so it has to start dealing with paging and pages on disk. The "bad allocation" and out of memory (OOM) errors are expected because once the system cannot dynamically allocate any memory, it kills the program. It is interesting to note that sometimes the system kernel panicked but other times it gracefully crashed - it seems that this nondeterminism may be affected by how stressed the system is.

The most disturbing result is the CPU being throttled. When examining the system logs, it appears the CPU temperatures spike above the safety threshold, forcing the system to take immediate action.

It is interesting to note that none of the experiments got close to the size of the memory aside from the Phoenix word count. It appears that the sequential implementations fell far short of utilizing the available memory. The most likely cause of this is that the C++ data structures are suboptimal for this job.

Word count: effects of $\frac{\text{core}}{\text{memory}}$ ratios

This final experiment is just a proof of concept; we want to show that core to memory to data ratios matter for performance. We continually increase data and change the detectable cores and memory, according to Section §5. The ratios we use and the results are in Figure 9.

We choose to incrementally change the ratios (Figure 9a) because of time restraints. A more balanced approach would have been to hold the memory static and incrementally change the number of cores and then to hold the cores static and incrementally change the amount of memory. We are still refining this technique but future studies will utilize this latter approach.

The results show that word count is CPU-bound until it runs out of memory. This is a relatively intuitive result and in general, things crash when there are less resources, but upon closer examination, there are are some interesting details. First, the curves of the graphs show that the ratios are not all equal because they all have different slopes. This implies that some ratios do better and that there is a certain degree of over-provisioning. For example, the sequential 2.5 ratio does very well up until 2GB, at which point the sequential 1.5 ratio starts to distance itself. Another example of over-provisioning are the Phoenix ratios. It is clear that less resources (2.5 ratio) can perform and scale almost as well as more resources (1.5 ratio). A step in the opposite direction by downgrading the ratio to 0.5 is a disaster as the result explodes off the graph and it looks like it may do worse than the sequential counterpart if the systems did not crash.

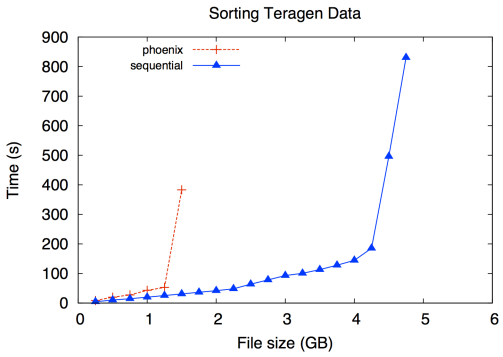
The graph needs to be extended but there are definitely relationships at play and the more we understand them, the better. This may seem like an obvious result: core to memory to data ratios matter, but are trying to show that the relationship cannot be ignored in future studies because the effects are real.

Sort: timing component breakdown

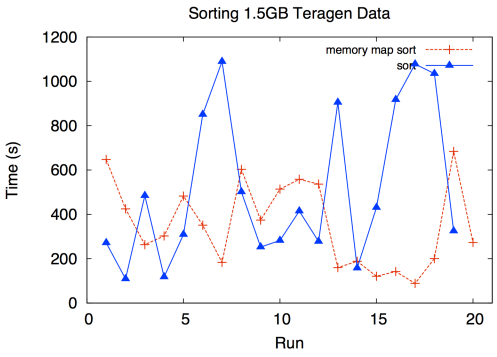
The sort benchmark has wildly unpredictable results, ranging in latency from 88.933 seconds to 683.932 seconds (7× slow down), as shown in Figure 10a. Surprisingly, sequential sort is better than the Phoenix sort for all values. This can be a consequence of two implementation details: (1) in Phoenix, we copy the entire unordered map into an array for sorting or (2) in Phoenix, we emit duplicates. Copying the entire key space is necessary because the `qsort` library function operates on a `std::vector` type. We emit duplicates because that is closely aligned with the way MapReduce performs the job, in that each mapper emits duplicates (as in the word count example). As a result of these implementation details, the key space is enormous and the combiners cannot aggregate values before sending them into the communication medium.

When we continually crashed or slowed to a crawl at 1.5GB for the Phoenix sort, we decided to hammer at that data point until we could see a trend. Figure 10b shows our attempts to weed out the problem, as we ran the same amount of data through the Phoenix sort for 20 runs. Intuition led us to try unmapping the memory mapped file to see if that had an effect. After the experiment, it is *still* unclear as to the reasons for the unpredictable performance.

We have some guesses as to why we are getting this kind of performance. First, we are running the tests consecutively, so the caches may be saving data from previous runs. This seems unlikely since the performance does not increase with each run - we doubt that we are saturating the cache or



(a) The unpredictable behavior can be attributed to a number of factors, but is most likely the consequence of a large number of key-value pairs.



(b) Further examination of the Phoenix sort implementation at 1.5GB shows no correlation between OS caching or memory mapping the file.

Figure 10: The Phoenix sort performs poorly, especially in relation to its sequential counterpart, as shown in Figure 10a. We zoomed in on the Phoenix data point at 1.5GB and tried and figure out what was going on, but an examination of the effects of memory mapping the file (Figure 10b) still did not solve the mystery.

virtual memory. Second, the Phoenix runtime may not handle an excessive amount of keys, as discussed in [39]. We might be able to improve performance if we use a different kind of container that stores the intermediate keys in a more optimal fashion.

Again, what is interesting is the interplay between data and the hardware of the system. The result is not that interesting, since the community probably doesn't care about sort but the fact that pure data can break a program should be interesting to anyone interested in scaling. If we apply the same methodologies that we applied to word count, we may be able to determine what is actually going on inside the system.

7. RELATED WORK

There is extensive literature from both academic and non-academic sources. It is not usually custom to include non-academic sources but we feel that it demonstrates such a demonstrable opinion of the current state of affairs that it is worth mentioning.

7.1 Directly comparing scale-out and scale-out

Michael et. al. [24] present a direct comparison of scale-out and scale-up by configuring equivalent cost machines ($200,000) for a narrow workload. They determined that scale-out has a resource utilization cost despite a performance advantage, which, for their workload, was up to a 4× speedup. Interestingly enough "scale-out-in-a-box" has better performance than pure scale-up.

In a similar study, Talkington et. al. [35] observed the behavior of a powerful SMP system with scientific (SPEC), commercial (TPC-H), and compute-intensive (Fluent) benchmarks. The study concluded that scale-up is limited by processor and memory speeds and resource contention while scale-out is limited by intra-node communication and workload management. More specifically, scale-up shows diminishing returns at 32 processors for the given workloads.

These studies succinctly enumerate the differences between the scaling approaches but they are limited by a number of factors. First, the studies are out-dated, and as a consequence, the benchmarks, cost, hardware specifications, and system configurations are no longer applicable. Second, the studies are relatively shallow. The benchmarks are limited and the systems are highly tuned resulting in performance measurements that make sense. It is our opinion that a more in-depth study with modern hardware, benchmarks, and system configurations will yield different results.

7.2 Scale-up: Examining System Architecture

The arrival of experimental operating systems signifies an exploration of the system design space to accommodate increasing data and hardware trends. Barrelfish [31] proposes a new OS architecture which facilitates heterogenous hardware and application demands by maintaing a system knowledge base and making execution decisions at runtime. FOS [37] proposes a new OS architecture which distributes kernel system services spatially instead of time-sharing them. Corey [3] addresses multiprocessor scalability by giving applications control of shared kernel data structures. The system introduces address ranges, kernel cores, and user-space sharing via system calls. Cerberus [33] is based on the observation that commodity operating systems cannot scale for a large number of CPU cores. The system introduces OS-clustering to provide a single OS image (via a Super-Process) to the user while virtualizing resources with Xen (share address space, file system, file contents, NICs) via shared memory and a static resource allocation (i.e. 8 cores per OS). These systems all attack new workloads and data sizes by changing how the system internals are structured and organized.

In contrast to a system design overhaul, another avenue of exploration for accommodating more data is to better un-

derstand and fix current systems. The common approach for these kinds of studies is to vary workloads and system parameters, profile and trace to characterize the behavior, and to fix any observed bottlenecks. Boyd-Wickizer et. al. [4] iteratively added cores, identified Linux bottlenecks at the 48 core threshold, and fixed micro/macro system level problems. [5] examines individual system calls and presents functions they call, how often they are called, and how long they take. [1, 26, 32, 11, 12, 13] all present tools to help the programmer "see" what is going on at the lower levels.

These studies all make contributions towards increasing visibility into the system from the application level but they are not applied to scale-out vs. scale-up studies. We leverage these tools and techniques to a get a better feel for how the system is behaving and to quantify what are causing various slowdowns in both architectures.

7.3 Blogs and Forums

The casual community has great contributions comparing scale-up vs. scale-out. They are included here because they are truly interesting but we do not cite any of their work since they are not rigorous enough and have not been critiqued by their peers. They are included here purely for the interested reader.

- Bob Plankers: pricing spreadsheet for scale up vs. scale out
  - → scale up gives us cost/time savings but you have to balance workload
- Massimo: ↑ cores, ↓ sockets; systems will be bottlenecked by memory
  - → this is scaling down and will limit our potential
  - → benchmarks that test 100% memory load (not 100% CPU utilization)
- Bas: octo-core will result in 8GB of RAM per core (limited and expensive)
  - → no benefit for provisioning a large machine
  - → get requirements, determine how application handles memory
- Robin Harris: the "memory wall" is memory speed, not its quantity
  - → references the pioneering "memory wall" publication
- Duncan Eppling: the risks of scaling up
  - → incorrect sizing results in ++ cost, downtimes, nodes (redundancy)
- Duncan Eppling: list of considerations for virtualization vs. single nodes
  - → scale up good for cost and trusting your hardware platform
  - → comments have good debate on vRAM's pricing model (discontinued)
- Ken Lassesen: TPC has configurations for ≡ performance w/ less RAM
  - → optimal SQL cost/transaction has certain hardware configurations
- VMWare Forum: balanced host has a general setup (4GB / core)

8. CONCLUSION

We gave a brief review of the scaling trends and history that we have seen up to this point and presented our prediction for the future: that there will be a re-examination of the advantages of scale-up. We posited that narrow methodologies and out-dated hardware may lead us to miss crucial bottlenecks and overheads when studying scale-up and scale-out systems.

In light of this, we addressed how we plan to overcome these limitations by expanding our methodology and using modern hardware. Our long term goal is to create a performance grid to find relationships between machine configurations and applications but we needed a way to fill in the performance grid. The biggest problem we faced was selecting applications and porting them in a way that was fair, represented, and feasible.

Using HiBench to select applications and Phoenix to port them, we showed how our methodologies show more important details about the interplay between core, data, and memory. With these new techniques, we can uncover new bottlenecks and slowdowns on bigger systems. The biggest contribution we have made in this paper, is that we can directly compare scale out and scale up while remaining fair, representative, and feasible.

Acknowledgments

Special thanks to Kleoni and Ike for helping me with this project.

9. REFERENCES

[1] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of the linux kernel. In *In Proceedings of the Real-Time Technology and Applications Symposium (RTAS02)*, page 133, 2002.

[2] A. Agarwal and M. Levy. The kill rule for multicore. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 750–753, New York, NY, USA, 2007. ACM.

[3] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[4] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, October 2010.

[5] S. S. P. G. Bridges and A. B. Maccabe. A framework for analyzing linux system overheads on hpc applications. In *Proceedings of the 2005 Los Alamos Computer Science Institute (LACSI '05)*, page 17, 2005.

[6] J. B. Buck, N. Watkins, J. LeFevre, K. Ioannidou, C. Maltzahn, N. Polyzotis, and S. Brandt. Scihadoop: array-based query processing in hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 66:1–66:11, New York, NY, USA, 2011. ACM.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[8] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 390–399, Washington, DC, USA, 2011. IEEE Computer Society.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operarting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[11] M. Desnoyers and M. R. Dagenais. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Linux Symposium 2008*, page 101, July 2008.

[12] M. Desnoyers, J. Desfossez, and D. Goulet. Lttng 2.0: Tracing for power users and developers - part 1, April 2012.

[13] M. Desnoyers, J. Desfossez, and D. Goulet. Lttng 2.0: Tracing for power users and developers - part 2, April 2012.

[14] Z. Fadika, M. Govindaraju, S. R. Canon, and L. Ramakrishnan. Evaluating hadoop for data-intensive scientific operations. In R. Chang, editor, *IEEE CLOUD*, pages 67–74. IEEE, 2012.

[15] M. Feldman. Cray unveils its first gpu supercomputer, May 2011.

[16] A. Gara, M. A. Blumrich, D. Chen, G.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, et al. Overview of the blue gene/l system architecture. *IBM Journal of Research and Development*, 49(2.3):195–212, 2005.

[17] J. L. Henning. Spec cpu2000: Measuring cpu performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[18] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In *ICDE Workshops*, pages 41–51, 2010.

[19] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[20] A. Jacobs. The pathologies of big data. *Commun. ACM*, 52(8):36–44, Aug. 2009.

[21] I. Kuz, Z. Anderson, P. Shinde, and T. Roscoe. Multicore os benchmarks: we can do better. In *Proceedings of the 13th USENIX conference on Hot topics in operating systems*, HotOS'13, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[23] K. S. Manjunath and A. Basu. Article: Effective technique for vm provisioning on paas cloud computing platform. *International Journal of Applied Information Systems*, 4(6):44–47, December 2012. Published by Foundation of Computer Science, New York, USA.

[24] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8. IEEE, 2007.

[25] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ICDMW '10, pages 170–177, Washington, DC, USA, 2010. IEEE Computer Society.

[26] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 335–348, New York, NY, USA, 2010. ACM.

[27] A. Quiroz, H. Kim, M. Parashar, N. Gnanasambandam, and N. Sharma. Towards autonomic workload provisioning for enterprise grids and clouds. In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 50–57. IEEE, 2009.

[28] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[29] R. Ranjan, L. Zhao, X. Wu, A. Liu, A. Quiroz, and

M. Parashar. Peer-to-peer cloud provisioning: Service discovery and load-balancing. *Cloud Computing*, pages 195–217, 2010.

[30] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[31] A. Sch§pbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

[32] S. Shende. Profiling and tracing in linux. In *Proceedings of the Extreme Linux Workshop 2, USENIX*, June 1999.

[33] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with os clustering. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 61–76, New York, NY, USA, 2011. ACM.

[34] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 9–16, New York, NY, USA, 2011. ACM.

[35] A. Talkington and K. Dixit. Scaling-up or out. *International Business*, 2002.

[36] G. Vilutis, L. Daugirdas, R. Kavaliunas, K. Sutiene, and M. Vaidelys. Model of load balancing and scheduling in cloud computing. In *Information Technology Interfaces (ITI), Proceedings of the ITI 2012 34th International Conference on*, pages 117–122. IEEE, 2012.

[37] D. Wentzlaff and A. Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review*, 43(2):76–85, 2009.

[38] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995.

[39] R. M. Yoo, A. Romano, and C. Kozyrakis. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 198–207, Washington, DC, USA, 2009. IEEE Computer Society.

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[41] J. Zhan, L. Zhang, N. Sun, L. Wang, Z. Jia, and C. Luo. High volume throughput computing: Identifying and characterizing throughput oriented workloads in data centers. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1712–1721. IEEE, 2012.

[42] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.