Evaluating High-Performance Computing on Google App Engine

Radu Prodan, Michael Sperk, and Simon Ostermann,
University of Innsbruck

// Google App Engine offers relatively low resource-provisioning overhead and an inexpensive pricing model for jobs shorter than one hour. //

FUNDING AGENCIES AND institutions must purchase and provision expensive parallel computing hardware to support high-performance computing (HPC) simulations. In many cases, the physical hosting costs, as well as the operation, maintenance, and depreciation costs, exceed the acquisition price, making the overall investment nontransparent and unprofitable.

Through a new business model of renting resources only in exact amounts for precise durations, cloud computing promises to be a cheaper alternative to parallel computers and more reliable than grids. Nevertheless, it remains dominated by commercial and industrial applications; its suitability for parallel computing remains largely unexplored.

Until now, research on scientific cloud computing concentrated almost exclusively on infrastructure as a service (IaaS)—infrastructures on which you can easily deploy legacy applications and benchmarks encapsulated in virtual machines. We present an approach to evaluate a cloud platform for HPC that's based on platform as a service (PaaS): Google App Engine (GAE).[1] GAE is a simple parallel computing framework that supports development of computationally intensive HPC algorithms and applications. The underlying Google infrastructure transparently schedules and executes the applications and produces detailed profiling information for performance and cost analysis. GAE supports development of scalable Web applications for smaller companies—those that can't afford to overprovision a large infrastructure that can handle large traffic peaks at all times.

Google App Engine

GAE hosts Web applications on Google's large-scale server infrastructure. It has three main components: scalable services, a runtime environment, and a data store.

GAE's front-end service handles HTTP requests and maps them to the appropriate application servers. Application servers start, initialize, and reuse application instances for incoming requests. During traffic peaks, GAE automatically allocates additional resources to start new instances. The number of new instances for an application and the distribution of requests depend on traffic and resource use patterns. So, GAE performs load balancing and cache management automatically.

Each application instance executes in a sandbox (a runtime environment abstracted from the underlying operating system). This prevents applications from performing malicious operations and enables GAE to optimize CPU and

0740-7459/12/$31.00 © 2012 IEEE

memory utilization for multiple applications on the same physical machine. Sandboxing also imposes various programmer restrictions:

- Applications have no access to the underlying hardware and only limited access to network facilities.
- Java applications can use only a subset of the standard library functionality.
- Applications can't use threads.
- A request has a maximum of 30 seconds to respond to the client.

GAE applications use resources such as CPU time, I/O bandwidth, and the number of requests within certain quotas associated with each resource type. The CPU time is, in fuzzy terms, equivalent to the number of CPU cycles that a 1.2-GHz Intel x86 processor can perform in the same amount of time. Information on the resource usage can be obtained through the GAE application administration Web interface.

Finally, the data store lets developers enable data to persist beyond requests. The data store can't be shared across different slave applications.

A Parallel Computing Framework

To support the development of parallel applications with GAE, we designed a Java-based generic framework (see Figure 1). Implementing a new application in our framework requires specialization for three abstract interfaces (classes): `JobFactory`, `WorkJob`, and `Result` (see Figure 2).

The master application is a Java program that implements `JobFactory` on the user's local machine. `JobFactory` manages the algorithm's logic and parallelization in several `WorkJobs`. `WorkJob` is an abstract class implemented as part of each slave application—in particular, the `run()` method, which executes the actual computational job. Each slave application deploys as a separate GAE application and, therefore, has a distinct URI. The slave applications provide a simple HTTP interface and accept either data requests or computational job requests.

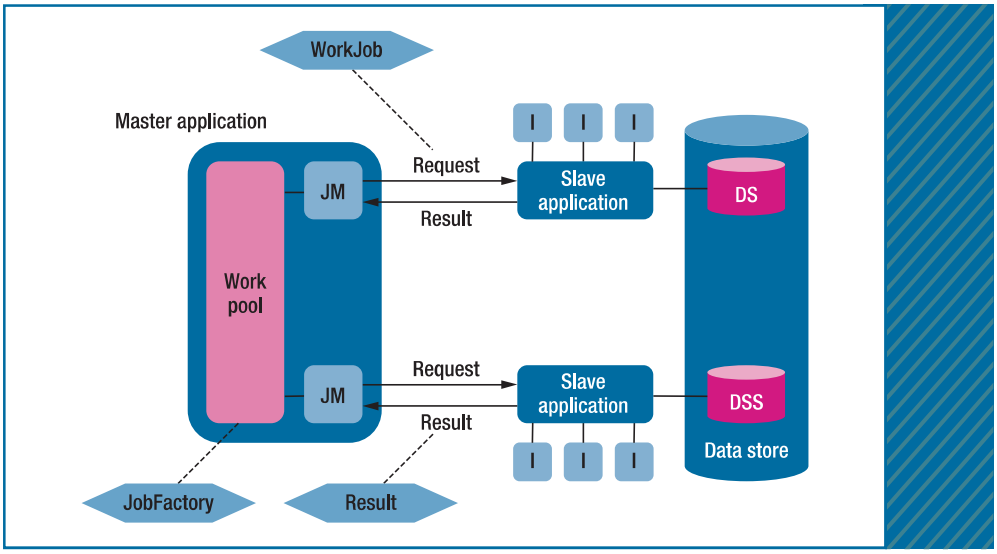

FIGURE 1. Our parallel computing framework architecture. The boxes labeled I denote multiple slave instances. The master application is responsible for generating and distributing the work among parallel slaves implemented as GAE Web applications and responsible for the actual computation.

Requests

The HTTP message header stores the type of request.

A *job request* contains one `WorkJob` that's submitted to a slave application and extended. If multiple requests are submitted to the same slave application, GAE automatically starts and manages multiple instances to handle the current load; the programmer doesn't have control over the instances. (One slave application is, in theory, sufficient; however, our framework can distribute jobs among multiple slave applications to solve larger problems.)

A *data request* transfers data shared by all jobs to the persistent data store (indicated by the `useSharedData` method). It uses multiple parallel HTTP requests to fulfill the GAE's maximum HTTP payload size of 1 Mbyte and improve bandwidth utilization. The `fetchSharedData` method retrieves shared data from the data store as needed.

In a *clear request*, the slave application deletes the entire data store contents. Clear requests typically occur after a run, whether it's successful or failed.

A *ping request* returns instantly and determines whether a slave application is still online. If the slave is offline, the master reclaims the job and submits it to another instance.

WorkJob Execution

Mapping `WorkJobs` to resources follows a dynamic work pool approach that's suitable for slaves running as black boxes on sandboxed resources with unpredictable execution times. Each slave application has an associated job manager in the context of the master application. It requests `WorkJobs` from the global pool, submits them to its slave

```
public interface JobFactory {
  public WorkJob getWorkJob();
  public int remainingJobs();
  public void submitResult(Result);
  public Result getEndResult();
  public boolean useSharedData();
  public Serializable getSharedData();
}

public abstract class WorkJob extends Serializable {
  private int id;
  public int getId();
  public void setId(int);
  public Result run();
  public void fetchSharedData();
}

public abstract class Result implements Serializable {
  private int id;
  private long cpuTime;
  public long getCPUTime();
  public void setCPUTime(long);
  public int getId();
  public void setId(int);
}
```

FIGURE 2. The Java code for our parallel computing framework interface. **JobFactory** instantiates the master application, **WorkJob** instantiates the slave, and **Result** represents the final outcome of a slave computation.

instances for computation (GAE automatically decides which instance is used), and sends back partial results.

We associate a queue size with every slave to indicate the number of parallel jobs it can simultaneously handle. The size should correspond to the number of processing cores available underneath. Finding the optimal size at a certain time is difficult for two reasons. First, GAE doesn't publish its hardware information; second, an application might share the hardware with other competing applications. So, we approximate the queue size at a certain time by conducting a warm-up training phase before each experiment.

The slave application serializes the **WorkJob**s' results and wraps them in an HTTP response, which the master collects and assembles. A **Result** has the same unique identifier as the **WorkJob**. The **calculationTime** field stores the effective computation time spent in **run()** for performance evaluation.

Failures

A GAE environment can have three types of failure: an exceeded quota, offline slave applications, or loss of connectivity. To cope with such failures, the master implements a simple fault-tolerance mechanism to resubmit the failed **WorkJobs** to the corresponding slaves using a corresponding exponential back-off time-out, depending on the failure type.

Benchmarks

We began our evaluation of GAE with a set of benchmarks to provide important information for scheduling parallel applications onto its resources. To help users understand the price of moving from a local parallel computer to a remote cloud with sandboxed resources, we deployed a GAE development server on Karwendel, a local machine with 16 Gbytes of memory and four 2.2-GHz dual-core Opteron processors. Instead of spawning additional sandboxed instances, the development server managed parallel requests in separate threads.

Resource Provisioning

Resource-provisioning overhead is the time between issuing an HTTP request and receiving the HTTP response. Various factors beyond the underlying TCP network influence the overhead (for example, load balancing to assign a request to an application server, which includes the initialization of an instance if none exists).

To measure the overhead, we sent HTTP ping requests with payloads between 0 and 2.7 Mbytes in 300-Kbyte steps, repeated 50 times for each size, and took the average. The overhead didn't increase linearly with the payload (see Figure 3) because TCP achieved higher bandwidth for larger payloads. We measured overhead in seconds; IaaS-based infrastructures, such as Amazon Elastic Compute Cloud (EC2), exhibit latencies measured in minutes.[2]

Just-in-Time Compilation

A Java virtual machine's just-in-time (JIT) compilation converts frequently used parts of byte code to native machine code, notably improving performance. To observe JIT compilation effects, we implemented a simple Fibonacci number generator. We submitted it to GAE 50 times in sequence with a delay of one second, always using the same problem size. We set up the slave application with no instances running and measured the effective computation time in the **run()** of each **WorkJob**. As we described earlier, GAE spawns instances of an application depending on its recent load (the more requests, the more instances). To mark and track instances, we used a **Singleton** class that contained a randomly initialized static identifier field.

Figure 4 shows that seven instances handled the 50 requests. Moreover, the first two requests in each instance took considerably longer than the rest. After

JIT compilation, the code executed over three times faster.

Monte Carlo Simulations

One way to approximate π is through a simple Monte Carlo simulation that inscribes a circle into a square, generates p uniformly distributed random points in the square, and counts m points that lie in the circle. So, we can approximate $\pi = 4 \cdot m/p$. We ran this algorithm on GAE.

Obtaining consistent measurements from GAE is difficult for two reasons. First, the programmer has no control over the slave instances. Second, two identical consecutive requests to the same Web application could execute on completely different hardware in different locations. To minimize the bias, we repeated all experiments 10 times, eliminated outliers, and averaged all runs.

Running the simulations. We conducted a warm-up phase for each application to determine the queue size and eliminate JIT compilation's effects. We executed the π calculation algorithm first sequentially and then with an increasing number of parallel jobs by generating a corresponding number of `WorkJobs` in the `JobFactory` work pool. We chose a problem of 220 million random points, which produced a sequential execution time slightly below the 30-second limit.

For each experiment, we measured and analyzed two metrics. The first was computation time, which represented the average execution time of `run()`. The second was the average overhead, which represented the difference between the total execution time and the computation time (especially due to request latencies).

Results. Figure 5 shows that serial execution on GAE was about two times slower than on Karwendel, owing to a slower random-number-generation routine in GAE's standard math library.[3]

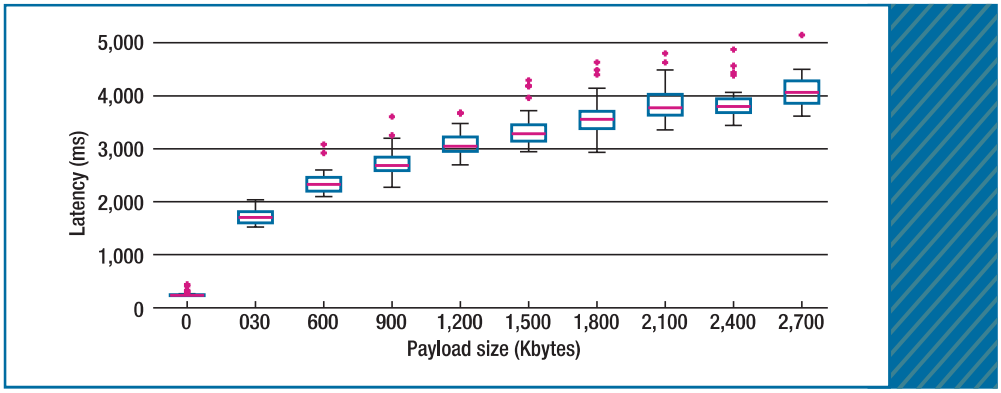

FIGURE 3. Resource-provisioning overhead didn't increase linearly with the payload because TCP achieved higher bandwidth for larger payloads.

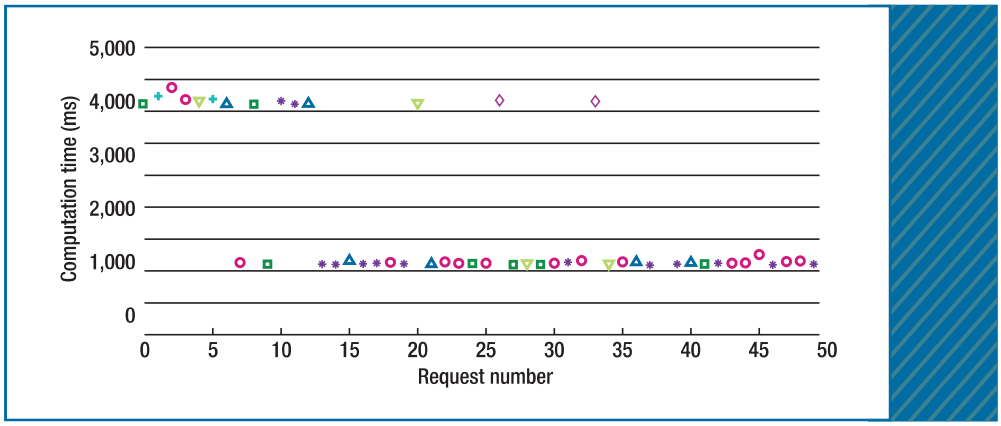

FIGURE 4. Computation time and the mapping of requests to instances. The first two requests in each instance took considerably longer than the rest. After just-in-time compilation, the code executed almost four times faster.

On Karwendel, transferring jobs and results incurred almost no overhead, owing to the fast local network between the master and the slaves. So, the average computation time and total execution time were almost identical until eight parallel jobs (Karwendel has eight cores). Until that point, almost linear speedup occurred. Using more than eight parallel jobs generated a load imbalance that deteriorated speedup because two jobs had to share one physical core.

GAE exhibited a constant data transfer and total overhead of approximately 700 milliseconds in both cases, which explains its lower speedup. The random background load on GAE servers or on the Internet network caused the slight irregularities in execution time for different machine sizes.

This classic scalability analysis method didn't favor GAE because the 30-second limit let us execute only relatively small problems (in which Amdahl's law limits scalability). To eliminate this barrier and evaluate GAE's potential for computing larger problems, we used Gustafson's law[4] to increase the problem size proportionally to the machine size. We observed the impact on the execution time (which should stay constant for an ideal speedup). We distributed the jobs to 10 GAE slave applications instead of one to gain sufficient quotas (in minutes).

In this case, we started with an initial problem of 180 million random

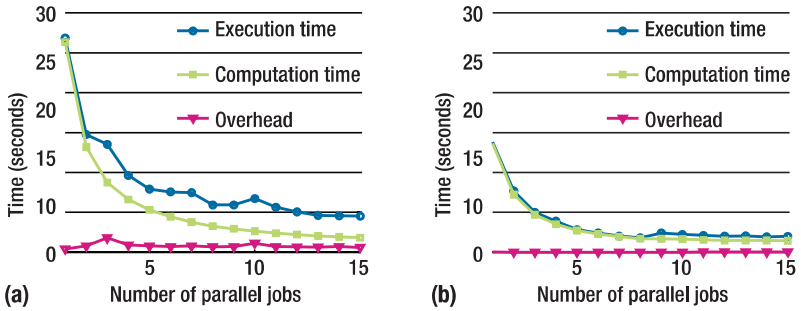

FIGURE 5. Results for calculating π on (a) Google App Engine (GAE) and (b) Karwendel, the local machine. Serial execution on GAE was about two times slower than on Karwendel, owing to a slower random-number-generation routine in GAE's standard math library.[3]

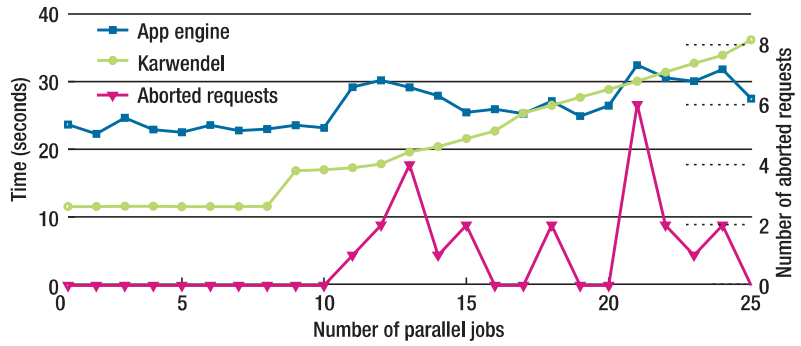

FIGURE 6. Scalability results for GAE and Karwendel for proportionally increasing machine and problem sizes. Karwendel had a constant execution time until eight parallel jobs, demonstrating our framework's good scalability.

points to avoid exceeding the 30-second limit. (For a larger number of jobs, GAE can't provide more resources and starts denying connections.) Again, Karwendel had a constant execution time until eight parallel jobs (see Figure 6), demonstrating our framework's good scalability.

Starting with nine parallel jobs, the execution time steadily increased proportionally to the problem size. GAE showed similarly good scalability until 10 parallel jobs. Starting additional parallel jobs slightly increased the execution time. The overhead of aborted requests (owing to quotas being reached) caused most irregularities.

For more than 17 parallel jobs, GAE had a lower execution time than Karwendel owing to Google's larger hardware infrastructure.

Cost Analysis

Although we conducted all our experiments within the free daily quotas that Google offered, it was still important to estimate cost to understand the price of executing our applications in real life. So, alongside the π approximation, we implemented three algorithms with different computation and communication complexity (see Table 1):

- *matrix multiplication*, based on row-wise distribution of the first matrix and full broadcast of the second;
- *Mandelbrot set generation*, based on the escape time algorithm; and
- *rank sort*, based on each array element's separate rank computation. This could potentially outperform other faster sequential algorithms.

We ran the experiments 100 times in sequence for each problem size and analyzed the cost of the three most limiting resources: CPU time, incoming data, and outgoing data, which we obtained through the Google application administration interface. We used the Google prices as of 10 January 2011: US$0.12 per outgoing Gbyte, $0.10 per incoming Gbyte, and $0.10 per CPU hour. We didn't analyze the data store quota because the overall CPU hours includes its usage.

As we expected, π approximation was the most computationally intensive and had almost no data-to-transfer cost. Surprisingly, rank sort consumed little bandwidth compared to CPU time, even though the full unsorted array had to transfer to the slaves and the rank of each element had to transfer back to the master. The Mandelbrot set generator was clearly dominated by the amount of image data that must transfer to the master. For π approximation, we generally could sample approximately $129 \cdot 10^9$ random points for US$1 because the algorithm has linear computational effort. For the other algorithms, a precise estimation is more difficult because resource consumption doesn't increase linearly with the problem size. Nevertheless, we can use the resource complexity listed in Table 1 to roughly approximate the cost to execute new problem sizes.

Finally, we estimated the cost to run the same experiments on the Amazon EC2 infrastructure using EC2's m1.small instances, which have a computational performance of one EC2 compute unit. This is equivalent to a 1.2-GHz Xeon or Opteron processor, which is similar to GAE and enables a direct comparison. We packaged the implemented algorithms into Xen-based virtual

TABLE 1

Resource consumption and the estimated cost for four algorithms.

Algorithm	Problem size (points)	Outgoing data		Incoming data		CPU time		Cost (US$)		
		Gbytes	Complexity	Gbytes	Complexity	Hrs.	Complexity	Google App Engine (GAE)	New GAE	Amazon Elastic Compute Cloud
π approximation	220,000,000	0	$O(1)$	0	$O(1)$	1.7	$O(n)$	0.170	0.078	0.190
Matrix multiplication	1,500 × 1,500	0.85	$O(n^2)$	0.75	$O(n^2)$	1.15	$O(n^2)$	0.292	0.203	0.440
Mandelbrot set	3,200 × 3,200	0.95	$O(n^2)$	0	$O(1)$	0.15	$O(n^2)$	0.129	0.066	0.440
Rank sort	70,000	0.02	$O(n^2)$	0.01	$O(n)$	1.16	$O(n^2)$	0.119	0.120	0.245

RELATED WORK IN CLOUD PERFORMANCE

Analysis of four commercial infrastructure-as-a-service-based clouds for scientific computing showed that cloud performance is lower than that of traditional scientific computing.[1] However, the analysis indicated that cloud computing might be a viable alternative for scientists who need resources instantly and temporarily.

Alexandru Iosup and his colleagues examined the long-term performance variability of Google App Engine (GAE) and Amazon Elastic Compute Cloud (EC2).[2] The results showed yearly and daily patterns, as well as periods of stable performance. The researchers concluded that GAE's and EC2's performance varied among different large-scale applications.

Christian Vecchiola and his colleagues analyzed different cloud providers from the perspective of high-performance computing applications, emphasizing the Aneka platform-as-a-service (PaaS) framework.[3] Aneka requires a third-party deployment cloud platform and doesn't support GAE.

Windows Azure is a PaaS provider comparable to GAE but better suited for scientific problems. Jie Li and colleagues compared its performance to that of a desktop computer but performed no cost analysis.[4]

MapReduce frameworks offer a different approach to cloud computation.[5,6] MapReduce is an orthogonal application class[5] that targets large-data processing.[7] It's less suited for computationally intensive parallel algorithms[8]—for example, those operating on small datasets. Furthermore, it doesn't support the implementation of more complex applications, such as recursive and nonlinear problems or scientific workflows.

References

1. A. Iosup et al., "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 6, 2011, pp. 931–945.
2. A. Iosup, N. Yigitbasi, and D. Epema, "On the Performance Variability of Production Cloud Services," *Proc. 11th IEEE/ACM Int'l Symp. Cluster, Cloud, and Grid Computing* (CCGrid 11), IEEE CS, pp. 104–113.
3. C. Vecchiola, S. Pandey, and R. Buyya, "High-Performance Cloud Computing: A View of Scientific Applications," *Proc. 10th Int'l Symp. Pervasive Systems, Algorithms, and Networks* (ISPAN 09), IEEE CS, 2009, pp. 4–16.
4. J. Li et al., "eScience in the Cloud: A Modis Satellite Data Reprojection and Reduction Pipeline in the Windows Azure Platform," *Proc. 2010 Int'l Symp. Parallel & Distributed Processing* (IPDPS 10), IEEE CS, 2010, pp. 1–10.
5. C. Bunch, B. Drawert, and M. Norman, "MapScale: A Cloud Environment for Scientific Computing," tech. report, Computer Science Dept., Univ. of California, Santa Barbara, 2009; www.cs.ucsb.edu/~cgb/papers/mapscale.pdf.
6. J. Qiu et al., "Hybrid Cloud and Cluster Computing Paradigms for Life Science Applications," *BMC Bioinformatics*, vol. 11, supplement 12, 2010, S3; www.biomedcentral.com/content/pdf/1471-2105-11-s12-s3.pdf.
7. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, vol. 51, no. 1, 2008, pp. 107–113.
8. J. Ekanayake and G. Fox, "High Performance Parallel Computing with Clouds and Cloud Technologies," *Cloud Computing and Software Services: Theory and Techniques*, S.A. Ahson and M. Ilyas, eds., CRC Press, 2010.

machines deployed and booted on `m1.small` instances. Table 1 shows that the computation costs were lower for GAE, owing mostly to the cycle-based payments as opposed to EC2's hourly billing intervals.

Google recently announced a change in its pricing model that will replace CPU cycles with a new instance-hours unit. The unit is equivalent to one application instance running for one hour and will cost $0.08. In addition, Google

ABOUT THE AUTHORS

RADU PRODAN is an associate professor at the University of Innsbruck's Institute of Computer Science. His research interests include programming methods, compiler technology, performance analysis, and scheduling for parallel and distributed systems. Prodan has a PhD in technical services from the Vienna University of Technology. Contact him at radu@dps.uibk.ac.at.

MICHAEL SPERK is a PhD student at the University of Innsbruck. His research interests include distributed and parallel computing. Sperk has an MSc in computer science from the University of Innsbruck. Contact him at sperk.michael@gmail.com.

SIMON OSTERMANN is a PhD student at the University of Innsbruck's Institute of Computer Science. His research interests include resource management and scheduling for grid and cloud computing. Ostermann has an MSc in computer science from the University of Innsbruck. Contact him at simon@dps.uibk.ac.at.

will charge $9 a month for every application. The new model will primarily hurt Web applications that trigger additional instances upon sparse request peaks and afterward remain idle.

Table 1 gives a rough cost estimation assuming 15 parallel tasks and an instance utilization of 80 percent for useful computation. The results demonstrate that the new pricing model favors CPU-intensive applications that try to fully utilize all available instances. In addition, we can expect free resources to last longer with the new pricing model.

We plan to investigate the suitability of new application classes such as scientific workflow applications to be implemented on top of our generic framework and run on GAE with improved performance. For a look at other research on cloud computing performance, see the "Related Work in Cloud Performance" sidebar.

Acknowledgments

Austrian Science Fund project TRP 72-N23 and the Standortagentur Tirol project RainCloud funded this research.

References

1. D. Sanderson, *Programming Google App Engine*, O'Reilly Media, 2009.
2. A. Iosup et al., "Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing," *IEEE Trans. Parallel and Distributed Systems*, vol. 22, no. 6, 2011, pp. 931–945.
3. M. Sperk, "Scientific Computing in the Cloud with Google App Engine," master's thesis, Faculty of Mathematics, Computer Science, and Physics, Univ. of Innsbruck, 2011; http://dps.uibk.ac.at/~radu/sperk.pdf.
4. J.L. Gustafson, "Reevaluating Amdahl's Law," *Comm. ACM*, vol. 31, no. 5, 1988, pp. 532–533.