# The Utility Coprocessor: Massively Parallel Computation from the Coffee Shop

John R. Douceur, Jeremy Elson, Jon Howell, and Jacob R. Lorch
*Microsoft Research*

## Abstract

UCop, the "utility coprocessor," is middleware that makes it cheap and easy to achieve dramatic speedups of parallelizable, CPU-bound desktop applications using utility computing clusters in the cloud. To make UCop performant, we introduced techniques to overcome the low available bandwidth and high latency typical of the networks that separate users' desktops from a utility computing service. To make UCop economical and easy to use, we devised a scheme that hides the heterogeneity of client configurations, allowing a single cluster to serve virtually everyone: in our Linux-based prototype, the only requirement is that users and the cluster are using the same major kernel version.

This paper presents the design, implementation, and evaluation of UCop, employing 32–64 nodes in Amazon EC2, a popular utility computing service. It achieves 6–11× speedups on CPU-bound desktop applications ranging from video editing and photorealistic rendering to strategy games, with only minor modifications to the original applications. These speedups improve performance from the coffee-break timescale of minutes to the 15–20 second timescale of interactive performance.

## 1 Introduction

The hallmark that separates desktop computing from batch computing is the notion of interactivity: users can see their work in finished form as they go. However, many CPU-intensive applications that are best used interactively, such as video editing, 3D modeling, and strategy games, can be slow enough even on modern desktop hardware that the user experience is disrupted by long wait times. This paper presents the Utility Coprocessor (UCop), a system that dramatically speeds up desktop applications that are CPU-bound and parallelizable by supplementing them with the power of a large datacenter compute cluster. We demonstrate several applications and workloads that are changed in kind by UCop: slow jobs that take several minutes without UCop become interactive (15–20 seconds) with it. Thanks to the recent emergence of utility-computing services like Amazon EC2 [8] and FlexiScale [45], which rent computers by the hour on a moment's notice, anyone with a credit card and $10 can use UCop to speed up his own parallel applications.

One way to describe UCop is that it effectively converts application *software* into a scalable cloud *service* targeted at exactly one user. This goal entails five requirements. *Configuration transparency* means the service matches the user's application, library, and configuration state. *Non-invasive installation* means UCop works with a user's existing file system and application configuration. *Application generality* means a developer can easily apply the system to any of a variety of applications, and *ease of integration* means it can be done with minimal changes to the application. Finally, the system must be *performant*.

UCop achieves these goals. To guarantee that the cluster uses exactly the same inputs as a process running on the client, it exclusively uses clients' data files, application images, and library binaries; the cluster's own file system is not visible to clients. The application extension is a simple user-mode library that can be installed easily and non-invasively. We demonstrate UCop's generality by applying it to the diverse application areas of 3D modeling, strategy games, and video editing; we also describe six other suitable application classes. Finally, UCop is easy to integrate: with our 295-line patch to a video editor, users can exploit a 32-node cluster (at $7/hour), transforming three-minute batch workflow for video compositing into 15-second interactive WYSIWYG display.

The biggest challenge in splitting computation between the desktop and the cloud is achieving good performance despite the high-latency, low-bandwidth network that separates them. It is dealing with this challenge that most distinguishes our work from past "depart-

ment clusters," such as NOW [9], MOSIX [12], and Condor [40], which assume users and compute resources are colocated and connected by a fast network. UCop combines a variety of old and new techniques to address networking issues. To reduce latency penalties, we carefully relax the file consistency contract and use automatic profiling to send cache validation information to the server before it is needed. To reduce bandwidth penalties, we use remote differential compression. A library-level multiplexer on the cluster end of the link scales the effects of these techniques across many servers. This combination reduces UCop's overhead for remotely running a process (assuming most of its dependencies are cached in the cloud) down to just a few seconds, even on links with latencies of several hundred milliseconds.

Of course, it makes little sense to pay a remote-execution overhead of a few seconds for a computation that could be done locally in less time. UCop is also not practical for tasks that are I/O bound, or for multi-threaded applications with fine-grained parallelism. In other words, UCop will not speed up an Emacs session or reduce the wait while Outlook indexes incoming email. However, there is an important class of desktop applications that are both CPU-bound and parallelizable; UCop enhances such applications less invasively and at more interactive timescales than existing systems.

The contributions of this paper are:

- We identify a new cluster computing configuration: remote parallelization for interactive performance, which provides practical benefit to independent, individual users.
- We identify the primary challenges of this new configuration: the latency and bandwidth constraints of the user's access link.
- We introduce *prethrowing* and *task-end-to-start consistency* as techniques for dealing with that link.
- We add remote differential compression, a cluster-side multiplexer, and a shared cache, resulting in a system that can invoke a wide parallel computation using just four round-trip latencies and minimal bandwidth.
- We show our system is performant, easy to deploy, and can readily adapt existing programs into parallel services running in the cloud.

We begin with a review of related work in §2. §3 describes UCop's architecture and implementation, and §4 describes UCop applications. §5 has several evaluations: microbenchmarks (§5.1), end-to-end application benchmarks (§5.2), a decomposition of each optimization's effect (§5.3), a sensitivity analysis to latency and bandwidth (§5.4), and an analysis of the optimized system's time budget (§5.5). Finally, §6 concludes.

## 2 Prior Work

UCop bears similarity to prior research on computational clusters, grids, process migration, network file systems, and parallel programming models.

**Computational clusters** are collections of computers that are typically homogeneously configured, geographically close, and either moderately or very tightly coupled. Sprite [32] is a distributed operating system that provides a network file system, process-migration facilities, and a single system image to a cluster of workstations. MOSIX [12] is a management system that runs on clusters of x86-based Linux computers; it supports high-performance computing for both batch and interactive processes via automatic resource discovery and dynamic workload distribution. Condor [40] is a software framework that runs on Linux, Unix, Mac OS X, FreeBSD, and Windows, and supports the parallel execution of tasks on tightly coupled clusters or idle desktop machines. The Berkeley NOW [9] system is a distributed supercomputer running on a set of extremely tightly coupled workstations interconnected via Myrinet. Cluster systems have been applied to interactive applications, including some of those we consider in §4, such as compilation [28] and graphics rendering [25]. However, for transparent parallelization, clusters require the client to be one of the machines in the cluster, requiring invasive installation. By contrast, in the UCop system architecture, the client is arbitrarily configured, geographically remote, and largely decoupled from the cluster.

**Computational grids** [19] are collections of computers that are loosely coupled, heterogeneously configured, and geographically dispersed. Grid systems comprise a large body of work, encompassing various projects (e.g., the Open Science Grid [20] and EGEE [3]), standards (e.g., WSRF [11]), recommendations (e.g., OGSA [20]), and toolkits (e.g., Globus [18] and gLite [5]). Although the majority of work on grid systems is focused on batch processing, there has been some limited research into adding interactivity to grid systems. IC2D [14] is a graphical environment for monitoring and steering applications that employ the ProActive Java library. I-GASP [13] is a system that provides grid interactivity via a remote shell and desktop. It also includes middleware for matching applications to their required resources, which is considered by some [38] to be a critical problem for satisfying the quality-of-service requirements of interactive applications. The DISCOVER [27] system provides system-integration middleware and an application-control network to support runtime monitoring and steering of batch applications. The Interac-

tive European Grid project [6] provides many services intended to support interactivity, including a migrating desktop, complex visualization services, job scheduling, and security services [34]. However, none of this work supports interactive applications per se, but rather provides mechanisms for interactively monitoring and manipulating a long-running distributed computation.

For a few specific types of applications, there exist massively parallel dedicated services that achieve interactive responsiveness to geographically remote, decoupled client machines. Amazon's Dynamo system employs hundreds of machines to provide real-time response to e-commerce transactions initiated by clients [16]. Google and other search engines perform brief bursts of highly parallel computation to answer clients' search queries [15]. SABRE and other online reservation systems provide clients with near-instant searching and booking for travel options [10]. However, these specialized services do not support arbitrary parallel applications, nor do they support applications whose authoritative state resides on the client machine.

Research on **process migration** is extensive; several surveys of this extensive body of work have been published [29, 31, 37]. To our knowledge, no prior work combines mechanisms and techniques as UCop does, and none of it achieves the same set of benefits. Moreover, the prior systems that are architecturally closest to UCop are not process-migration systems but network file systems. In a sense, UCop is a network file system in which the user's machine is the file server, tuned for a specific usage scenario.

Sun's NFS [36] is a basic **network file system**; in the UCop context, NFS's chatty protocol would make highly inefficient use of the high-latency connection between the client and the datacenter. The Andrew File System (AFS) [23] and Coda [26] avoid chattiness by employing leases [22]. However, leases require the ability to inspect the effect of every file system operation, which would greatly impinge on our goal of non-invasive installation; rather than simply installing a new applications, users would have to start using a new file system. UCop's prethrowing (§3.3) achieves the same performance benefits as leases without modifying the underlying file system.

The Low Bandwidth File System (LBFS) [30] is specifically aimed at improving performance over low-bandwidth WAN links. It employs caching, differential compression, and stream compression, in much the same manner as UCop does to minimize bandwidth usage (§3.4). As a general remote file system, LBFS lacks crucial optimizations for the UCop context, including our task-based consistency model, coalescing of tasks into jobs, cache sharing, and a library interface, all of which we show to be critical to achieving interactive performance (§5.3). In addition, LBFS uses leases instead of prethrowing, so the machine that holds the authoritative files (the server in LBFS terms, but the client in UCop terms) must store its files using the Arla [44] AFS client. This would require invasive installation in our scenario.

Involved **parallel programming models**, such as the Parallel Virtual Machine (PVM) [39] and the Message Passing Interface (MPI) [17], serve more tightly-coupled parallel applications. However, these are mechanisms for writing new applications. UCop's simple model, while less general, offers much easier integration for existing applications, even those not designed to exploit a cluster.

## 3 The Utility Coprocessor

In this section, we describe the design and implementation of the Utility Coprocessor. Sections 3.1 and 3.2 describe the programming model, and outline our implementation of its execution environment. We then describe the optimizations required to achieve good performance over a high-latency, low-bandwidth network link.

### 3.1 Programming model

We had several goals in designing UCop's programming model: simplicity for developers, generality across applications and operating system configurations, and good performance over slow links.

One of the mechanisms UCop uses to achieve these goals is *location independence*: applications can launch remote processes, each of which has the same effect as if it were a local process. Suppose an application's work can be divided among a set of child processes, each of which communicates with the parent through the file system or standard I/O. UCop provides a command-line utility, `remrun`, that looks like a local worker process, but is actually a proxy for a remotely running process. A simple change from `exec("program -arg")` to `exec("remrun program -arg")` provides the same semantics to the application while offloading the compute burden from the client CPU.

The consistency contract is simple: each child process is guaranteed to see any changes committed to the client file system before the child was launched, and any changes it makes will be committed back to the file system before the child terminates. Thus, dependencies among sequential children, or dependencies from child to parent, are correctly preserved. We refer to this contract as *task-end-to-start consistency* semantics. Because this contract applies to the entire file system, remote processes see *all* the same files as local client processes, including the application image, shared library binaries, system-wide configuration files, and user data.

When `remrun` is used to launch a proxy child process, it transmits an `exec` message to the cluster that includes remrun's command line arguments and environment variables. The cluster picks a worker node and launches a worker process with the specified arguments and a replicated set of environment variables, `chroot`ed into a private namespace managed by the UCop daemon (via the FUSE user-space file system framework [4]). On each read access to an existing file, UCop faults the file contents from the client; on each write to a non-existing file name, UCop creates the file in a buffer local to the node's file system. To prevent violations of task-end-to-start semantics from failing silently, UCop disallows writes to existing files. Standard input and output are shuttled between the client proxy process and the cluster worker process. When the worker process exits, UCop sends any surviving created files to the client. It also sends the process exit status; the client proxy process `exit`s with the same status.

An example best illustrates how UCop provides location independence. When compiling a single source file, UCop's `remrun gcc hello.c` produces an output file identical to a locally run `gcc hello.c`, because the remote version

- has the same `$PATH` as the client, and sees the same directories, so uses the same `gcc`;
- sees the same environment, including `$LD_LIBRARY_PATH` (shared library search path) and `$LANG` (localization);
- runs `gcc` in the same working directory, and thus finds the correct `hello.c`;
- finds the same compiler configuration and system include files; and
- writes all its output to the the client file system in the same place as if it had run locally.

Contrast this approach to other remote execution systems. Application-specific clusters such as compile and render clusters [33, 35] must be configured with a version of the compiler or renderer that matches that on the client. Grid and utility computing clusters standardize on a configuration, requiring the client configuration to conform. Process migration systems such as NOW, Condor, and MOSIX assume that user and worker machines have a network-shared `/home` and identical software configurations—for example, so that a dynamically linked executable built on a user's machine can find its shared libraries when it executes on the cluster.

The Utility Coprocessor is meant to be used by disparate independent users. No single configuration is ideal; various users sharing a cluster may have conflicting configurations. The semantics presented here hide these conflicts. Each UCop worker process mimics the client computer, and a single cluster may do so simultaneously across users and applications. As we show in §5.1.2, different Linux distributions can transparently use the same UCop cluster without any explicit preconfiguration. Our UCop cluster, which happens to use GNU libc 2.3.6, never exposes its own libraries to client applications. We have demonstrated applications that expect glibc versions as old as 2.3 and as new as 2.9.

## 3.2 Limitations on location independence

UCop's location-independent compute model does have limits: it extends only to the file system, environment variables, process arguments, and standard I/O pipes. Programmers UCopifying an application need to be aware of these limits. As we will show, these limits are not stumbling blocks in practice; a variety of applications can be UCopified easily.

Because UCop supports no interprocess communication other than standard I/O pipes, it precludes tightly-coupled computations in which concurrent child processes synchronize with each other using shared memory, signals, or named pipes. Some of the applications we adapted to UCop used an unsupported mechanism; our modifications primarily involved rerouting this communication through the file system (see §4).

Another limitation is that the kernel seen by a remote process is that of the cluster's worker machine, not the user's client. This is significant for two reasons. First, the semantics of system calls change slightly between kernel versions. Our application tests have not yet revealed any failures due to such a kernel incompatibility, but they are likely in code that is tightly coupled with the kernel. Second, there will be detectable differences in the machine-local state exposed by the kernel, such as process lists and socket state. UCop hides most of `/dev` and `/proc` from workers, exposing only commonly-used pseudo-devices such as `/dev/null` and `/proc/self`; the latter supports commonly-used idioms for finding loadable modules using a path relative to the currently executing image's path.

Finally, regular files on the client machine appear on the remote machine as symbolic links to files named by a hash of their contents. This is a workaround for a performance problem discussed in §5.1.3. It has little effect on most programs in practice.

## 3.3 Minimizing round trips

At this point, we have a basic system model with semantics suitable for the class of applications we aim to support. However, a straightforward implementation would perform poorly on a high-latency, low-bandwidth

link. We turn now to the problem of using that link efficiently by minimizing round-trip and bandwidth costs, starting with the former.

An obvious requirement for reasonable performance is to cache file contents near the cluster. The classic question is how to ensure that cached content is fresh. Neither frequent validation (à la NFS [36]) nor leases (à la AFS [23] or Coda [26]) are compatible with UCop's requirements, as detailed in §2.

**Prethrow.** Consistency semantics require that, for each path a worker touches during its run, we communicate the mutable binding from path to file attributes and content. A naïve implementation might ask the client for each binding on-demand, requiring one round-trip per file. We observe almost all paths touched by an application are libraries and configuration data touched on *every* run, making them easy to predict. Rather than wait for workers to request path information serially, the client sends a batch of path information likely to be useful before execution starts. We call this a *prethrow*—like a prefetch, but initiated by the sender. A prethrow is a hint: it can improve performance but does not change semantics if the prediction is wrong.

The client maintains sets of accessed paths, indexed by the first argument to `exec`. This way, the set for the 3D modeling program is maintained separately from that for the video editor. UCop prethrows only those paths that have been accessed more than once, to prevent pollution of the prethrow list by temporary files from previous runs. Currently, paths do not expire out of the prethrow list; in future versions, the server will provide a list of useless prethrows to the client after execution, to help the client decide which paths should expire.

One potential limitation of indexing by executable name is that UCop does not distinguish between two different programs invoked via the same interpreter (e.g., Python). UCop may therefore send information about paths that are not relevant. Because prethrows are hints, compact, and cachable (§3.5), this has not been a problem in practice.

## 3.4 Minimizing bandwidth

In a bandwidth-constrained environment, caching is critical. We adopt the well-known approach of caching by immutable hash, so that if a block of data is referred to by multiple names we only have to transmit it once.

**Remote differential compression.** Whole-file caching works well for files that change rarely, such as application binaries. However the user's input often changes slightly between UCop invocations. For example, a video editor's edit decision list (EDL) is a compact representation of the user's proposed manipulations to a set of (unchanging) video input files. The EDL changes slowly, at keyboard and mouse bit rates. *Remote differential compression* (RDC), used by LBFS [30] and rsync [42], is useful in this scenario. RDC detects which parts of a file are already cached and transmits only a small region around each changed part.

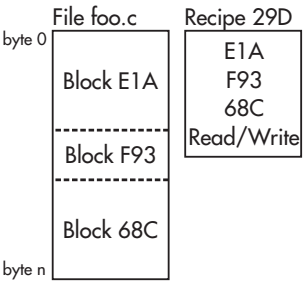


Figure 1: Objects in UCop's file synchronization protocol. To make the illustration compact, 20-byte SHA-1 hashes are represented by three hexadecimal digits.

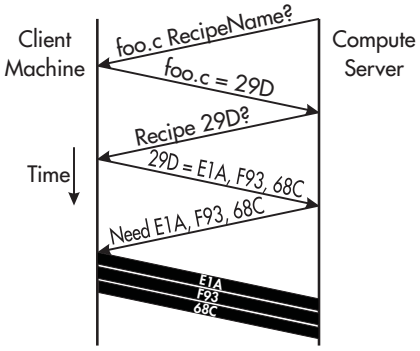


Figure 2: File transfer protocol with cold caches. To make the illustration compact, 20-byte SHA-1 hashes are represented by three hexadecimal digits.

To understand our use of RDC, we first introduce some terminology (Figure 1). UCop uses the rsync fingerprint algorithm to divide all files' contents into blocks with offset-insensitive boundaries. (We plan future support for LBFS, which is more robust to modifications.) It then constructs a *recipe* for each file: a list of its constituent blocks' hashes, plus the file's permissions and ownership fields. This recipe can itself be large, so we often compactly refer to it by its hash, called a *RecipeName*.

UCop rolls whole-file caching and RDC into a single mechanism (Figure 2). Worker nodes resolve each application-requested path to a RecipeName, first by checking prethrows, then making a request to the client. If the worker recognizes the RecipeName, it knows that it already has the whole file cached. Otherwise, it requests the recipe, then any blocks from that recipe it lacks.

**Stream compression.** After RDC, the number of bytes that still must be transmitted can often be reduced using conventional compression. UCop compresses its

channels with `zlib`.

**Cache sharing.** Multiple worker processes virtually always share files, such as the C library. It is wasteful for the client to send this data to each worker over the bottleneck link. Cluster nodes are interconnected with a high bandwidth network, so it is better for the client to send each file once to a cache shared by all workers.

We implemented this scheme by introducing a distributor node, called *remdis*. Remdis has a pass-through interface: it accepts jobs from the client as if it were a (fast) cluster node, and submits jobs to workers as if it were a client. Remdis forwards most messages in both directions without modification. However, it intercepts file system requests, interposing its own cache. Duplicate requests for the same content are suppressed, ensuring that no unique block is sent over the bottleneck link more than once. The Remdis cache does not change consistency semantics because RecipeNames and content block hashes describe immutable content.

In our experiments, remdis began to become a bottleneck at around 64–128 nodes. Because of its simple interface, however, it would be straightforward to build a 32-wide tree of remdis nodes to extend the distribution function to higher scales.

**Job consistency.** Computing and sending a prethrow message requires the client to look up the modification times of hundreds of files and transmit a few tens of KiB across the bottleneck link. Invoking $n$ tasks incurs these costs $n$ times. On the other hand, using the same prethrow message for all tasks, sent once and rebroadcast by remdis, reduces the prethrow cost by a factor of $n$.

Of course, reusing a single prethrow message violates our consistency model. If a file changes between when Task A and Task B are launched, but B uses A's prethrow message, B will see the file used by A, which is now stale. This is not a problem for groups of tasks that have no interdependencies; we call such a collection of tasks a *job*. UCop has support for *job-end-to-start consistency* semantics: each task sees any changes committed to the client file system before its enclosing *job* was launched. Applications that can operate with these semantics group tasks into jobs and generate one prethrow for each job. Other than `make`, all of the applications we deployed bundle their tasks into a job.

## 3.5 Client-side optimizations

The following two optimizations are performed on the client and thus involve no changes to the protocol.

**Recipe caching.** Constructing a recipe on the client is fast. However, some applications require hundreds of recipes, and the client can not generate a prethrow until it has them all. Thus, the client caches recipes, along with the last-modified time (mtime) of the underlying file. When a recipe is needed, the client uses the cached version if the file's mtime has not changed. For one application, this optimization saves the client from hashing 93 MiB of content, saving seconds of computation (see §5.3).

**Thread interface.** The `remrun` command-line utility lets applications divide their work in a natural way, creating what seem to be local worker processes but are actually proxies for remote processes. This elegance makes it trivial to expose remote execution opportunities in systems like `make`. However, simply launching 32 or 64 local processes can take several seconds, particularly on low-end desktop machines. This can consume a significant fraction of our budget for interactive responsiveness.

Thus, we added a `remrun()` library interface. A client that wants $n$ remote processes can spawn $n$ threads and call `remrun()` from each; the semantics are identical to spawning instances of the command-line version. The library obviates the need for extra local processes in exchange for a slightly more invasive change to the application.

## 3.6 Summary

In the common case, UCop incurs four round trips: the necessary one, plus three more to fault in changed user input. (We believe it is possible to eliminate all but a single RTT; see §5.5.2.) UCop also uses bandwidth sparingly. It uploads only one copy of the path attributes required by our consistency model, the per-task parameters with duplication compressed away, and the changed part of the job input. It downloads only the output data and the exit codes of the tasks.

Together, these optimizations compose an algorithm that attaches to application code with a simple interface, yet minimizes both round trips and bandwidth on the high-latency, low-bandwidth link. UCop effectively transforms *software* not originally designed for distributed computation into an efficient, highly parallel application *service* targeted at a single user.

# 4 Applications

In this section, we describe various classes of applications that work well with UCop. We first describe four applications we have already ported (with performance evaluations to come in §5.2). We then describe other suitable application categories.

## 4.1 Make

The process of adapting software to UCop is best explained by starting with a simple example: `make`, the

automatic software build tool. The user's rules in the `Makefile` file tell `make` how to transform input files into output files, e.g., by invoking `gcc`. `make` assumes that when a command completes, the output file has been generated, and it is safe to launch a new command that depends on that output. That is, `make` assumes task-end-to-start consistency. Therefore, one can replace `gcc` with `remrun gcc`—literally, in the `Makefile` definitions—to push each compilation out to UCop. Additionally, `make` has a built-in facility for exploiting parallelism intended to exploit a local multiprocessor: `make -j 20` launches up to 20 concurrent processes that have no mutual ordering constraints. With `remrun`, those concurrent compiles are delegated to the UCop cluster.

Adapting `make` to UCop is trivial since it was designed to expose parallel work as separate processes communicating through the file system. For the monolithic applications we describe next, minor modification is required to expose concurrency as separate processes.

## 4.2 Blender

Blender [1] is a 3D modeling, animation, and rendering program written in C and C++. A common usage mode is to interactively build a model using a real-time wire-frame or shaded model; then, to refine details and lighting, the user requests a ray-traced photorealistic rendering. Since ray tracing is embarrassingly parallel, Blender has a built-in facility for exploiting local multiprocessing. Specifically, it can be easily configured to render different tiles of a scene in different threads, each accessing the current world model via shared memory.

Blender also includes a notion of a render cluster that can batch-process an animation. To use it, the user must configure a cluster with software matching her current version of Blender, with a network-mounted shared file system such as NFS, accessible over a high-bandwidth and low-latency network.

UCop can transform Blender's minutes-long batch-style frame render into an interactive-speed preview. We modified Blender's preview code to write the current 3D model to a temporary file, split the frame into very small (8-pixel-wide) tiles and dispatch a random subset of tiles to each worker node. The randomization reduces the inefficiency introduced by inter-task variance; without it, worker processes responsible for complex portions of the scene don't finish rendering until long after other workers have gone idle. As each worker completes, it writes its JPEG output tiles back to the file system. The parent UI process `wait`s for the children; as each returns, the tiles are read and displayed, generating a preview that is gradually completed. These changes comprise 167 statements.

One unfortunate property of Blender is its unstable model file format: The temporary model file differs substantially from one render to the next, even when the model is unchanged. Therefore, even minor changes to the viewpoint or model require relatively large updates. UCop's remote differential compression (§3.4) is able to eliminate all but 11% of the differences. With warm caches, render requests typically transmit 779 KiB of blocks to express the input delta. With further effort, Blender might be updated to use a stable file format.

## 4.3 Chinese Checkers

Another suitable application class is turn-based strategy games played against a computer opponent. The effective skill of the computer is tightly linked to the amount of processing time available, making players choose between a good artificial opponent or a fast one. Interactivity is key here: it is not fun to play against an opponent that takes a dozen minutes to make each move. Traversal of AI search trees is highly parallelizable, so these games are good candidates for adaptation to UCop.

The application we use to demonstrate this class is `b1cc`, a Java program that plays Chinese Checkers [21]. Its "expert player" mode is based on a 4-deep alpha-beta-pruned minimax move-tree search.

We modified the tree search to emit a snapshot of the game configuration using the save-game function, then dispatch each branch of the first level of the search tree to a separate process. Each process computes an $(n-1)$-level alpha-beta minimax. This sacrifices our ability to prune across trees, but we expect to make up for this with the high parallelism UCop can bring to bear.

This approach was easy, but it exposes varying degrees of parallelism, and its tasks exhibit high variance. A better approach might be to locally evaluate the tree to a greater depth to find low-variance task subsets.

## 4.4 Cinelerra

Cinelerra [2] is an open-source video editing tool. With it, the user creates an *edit decision list (EDL)*, a metadata document that describes how source video is to be clipped, transformed, and composited into the output video. Cinelerra then performs these operations.

To test Cinelerra, we constructed a 45-second video montage composed of 29 MiB of low-resolution clips from a digicam. This montage uses only simple animated transformations, but renders 8.3× slower than realtime. Cinelerra offers many compute-intensive effect plugins that slow down previewing even more.

Like Blender, Cinelerra includes a notion of a render cluster that depends on explicit version configuration and a fast network. Its "background render" function breaks a clip into frames and pre-renders the sequence of

frames so the operator can preview the sequence at full frame rate. We modified Cinelerra to emit a set of control files (in Cinelerra's native job-control language) that divide the preview region temporally into brief snippets. It launches one child process to render each snippet to MPEG, then collects the MPEGs into a render timeline and plays them in order.

The biggest constraint on using Cinelerra with UCop is getting enormous video inputs to the cluster. Our 29-MiB input videos represent amateur video editing; serious editing will use multi-GiB input files. While they are read-only and thus their size does not affect a warm-cache scenario, big inputs produce substantial transmission delay in a cold-cache scenario.

Three techniques may mitigate this constraint. First, UCop might demand-fault individual blocks rather than entire files; this can help if only small portions of the input videos are used in the output. Second, Cinelerra might transcode video at the client into lower-quality drafts to exploit UCop even when transmission delays are dominant. Third, a user might fault in media to a UCop cluster (e.g., by running `remrun md5sum movie.avi` on it) the day before sitting down to edit.

## 4.5 Other applications

Beyond the applications we have modified, many other application classes can exploit UCop. The best applications are those where small changes to input incur CPU-bound and coarsely parallelizable computation. This section has some examples.

One potential class is mathematics software. For instance, numeric modeling packages such as Matlab and Octave parallelize vector math, and symbolic math packages such as Macsyma and Mathematica parallelize manipulation of independent subexpressions. Also, spreadsheet applications have parallelizable data-flow models.

Speech dictation software often performs a great deal of processing to parse a small amount of user speech. A researcher familiar with the area claims desktop access to parallel resources would improve quality and enable new applications [43].

Interactive GIS applications often perform CPU-intensive tasks, such as rendering a large database of vector data into a bitmap or performing convolutions on large bitmaps (e.g., reprojecting maps or aerial photographs). In these applications, small user inputs such as changes in view or layer registration can change the global configuration and necessitate CPU-bound re-rendering.

Photo manipulation software may also be readily adaptable to UCop. Photoshop and GIMP are adopting a nondestructive editing model, i.e., recording a stack of operations rather than just their cumulative effects. This stack is essentially an edit decision list, which UCop could send concisely. Image filters are both coarsely parallelizable and slow, making them a good fit for UCop.

Finally, software analysis tools, such as model checkers, whole-program static analyzers, and theorem provers generate substantial parallel workloads and are often used as part of a developer's interactive workflow.

Note that like Blender and Cinelerra, some of these applications already have support for a single-purpose, locally-administered, tightly-coupled cluster. Some even sell dedicated cluster hardware [7]. UCop, in contrast, is general: a single cluster running a single piece of software that can service all these applications simultaneously.

# 5 Evaluation

Our evaluation of UCop is divided into five parts. We begin with microbenchmarks in §5.1. End-to-end application benchmarks are described in §5.2. In §5.3, we analyze the efficacy of UCop's protocol optimizations, showing how performance suffers as each is disabled. We present a sensitivity analysis to latency and bandwidth in §5.4. Finally, in §5.5, we decompose how a typical UCop task spends its time budget.

All of our experimental clusters were constructed from Amazon's EC2 "Elastic Compute Cloud" service. Each VM is one of Amazon's "high-CPU medium" instances: a Xen virtual machine with 1.7 GB of memory and 2 CPU cores, each of which is approximately equal to a 2.5GHz Opteron or Xeon processor, circa 2007. Within EC2, we measured a typical interconnect bandwidth of 800 Mbit/sec and RTT of 600 $\mu$sec. As we will see in §5.2, most tests used artificial bandwidth and latency restrictions to emulate the typical case of a client separated from the compute cluster by a bottleneck link.

## 5.1 Microbenchmarks

### 5.1.1 Correctness

Our task-close-to-open consistency model and whole-file-system replication scheme were designed to let remote processes produce results identical to those produced by local computation. To verify this property, we used UCop to build GNU Coreutils v7.1, a collection of 102 system utilities. The build process has an intricate dependency structure and invokes hundreds of sub-tasks, many of which redirect `stdin` or `stdout` to other programs or the file system. Errors in UCop's consistency model or its implementation are likely to cause build failures (and did so in early versions of UCop). Adapting the build process to UCop only required typing

| OS Distribution | libc | gcc | kernel |
|---|---|---|---|
| Centos 5.2 | 2.5 | 4.1.2 | 2.6.18 |
| Debian 3.1 | 2.3.2 | 3.3.5 | 2.6.16 |
| Debian 4.0 | 2.3.6 | 4.1.2 | 2.6.21.7 |
| Debian 5.0 | 2.7 | 4.3.2 | 2.6.21.7 |
| Debian 6.0 beta | 2.7 | 4.3.3 | 2.6.21.7 |
| Fedora Core 8 | 2.7 | 4.1.2 | 2.6.21.7 |
| Gentoo 2008.0 | 2.6.1 | 4.1.2 | 2.6.18 |
| Ubuntu 9.04 | 2.9 | 4.3.3 | 2.6.21.7 |

Table 1: Linux distributions used as clients to compile GNU Coreutils with a UCop cluster running Debian 4.0. In each case, UCop generated binaries identical to those compiled locally.

`./configure CC="remrun gcc"`. We also compiled Coreutils locally; all the locally-compiled outputs were identical to their cluster-compiled counterparts.

Further supporting our claim of location independence is our (accidental) discovery that `remrun` is "self-hosting." An author was tinkering with remrun commands when he noticed the system had slowed, and was transferring files that seemed unrelated to his task. He eventually discovered that he'd accidentally edited his command-line to read `remrun remrun gcc hello-world.c`—that is, a recursive call to remrun. The command still ran correctly; the remrun client ran on the server automatically.

We also built this paper using `remrun latex paper.tex`. The emitted .dvi file differed from a locally-built copy in one byte: the minute field of a timestamp.

#### 5.1.2 Configuration Transparency

To test UCop's insensitivity to heterogeneous clients, we repeated the Coreutils build test on various distributions of Linux, only one of which matched the version running on the UCop cluster itself (Debian 4.0). Each distribution generated distinct outputs due to variations in the versions of gcc and libc. Indeed, simply invoking Debian 5.0's `gcc` binary on a Debian 4.0 machine fails due to shared library incompatibility. Using UCop, however, every locally-built binary matched the binary the cluster built on that client's behalf. The distributions we tested are shown in Table 1.

#### 5.1.3 FUSE Performance

We implemented UCop's on-demand file system using FUSE [4], a user-space file system framework. This simplified development, but proxying every file system operation through user-space has a significant performance cost for I/O intensive processes. In this section, we evaluate a technique to mitigate this cost.

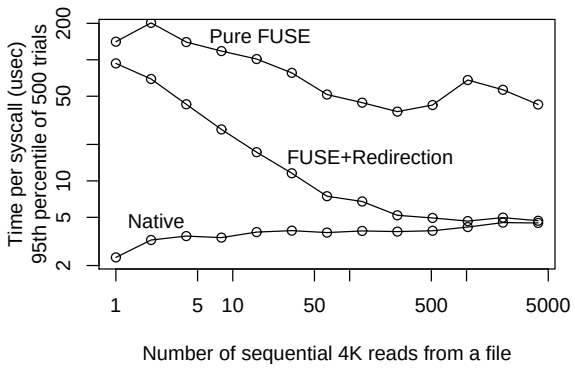


Figure 3: Log-log plot of the amortized time per syscall after one `open`, $n$ `read`s, and one `close` on a file. All reads are 4,096 bytes. The file was resident in the buffer cache.

When a worker process tries to open a path corresponding to an extant file on the client, FUSE instantiates that path not with a local file but with a symbolic link to a file with the appropriate contents. This file is kept on a kernel-managed native file system. Thus, access to extant files is mediated by FUSE only during `open`; other operations like `read` are handled much more efficiently.

We quantified the advantage of our approach by measuring the time required to open, read, and close a file. Figure 3 shows the amortized cost per syscall for sequential 4,096 byte `read`s with a warm buffer cache, plotted for a range of `read`s per `open`.

The top curve shows the performance of a FUSE-managed file system without redirection. The bottom curve shows the performance of Linux's native ext3 file system. The slowdown is significant, and is worst for small numbers of reads, ranging from $10\times$ to $60\times$. The middle curve shows amortized read performance with our symlink scheme in place. The optimization never hurts performance, and after about 40 `read`s, improves amortized performance to within $2\times$ native.

Note that the optimization does slightly hurt transparency: all files seen by the workers are symbolic links.

### 5.2 Application Benchmarks

In this section, we describe end-to-end benchmarks for three applications run on UCop, tested under realistic network conditions. In most tests, the client and cluster were both within Amazon EC2, with latency artificially injected and bandwidth constrained by Linux Traffic Control [24]. The exception is Section 5.2.5, in which we ran experiments from a real coffee shop.

To determine what network conditions should be emulated for our EC2-based experiments, we tested the networks at various locations in Seattle that offer public wireless Internet access. These included two coffee

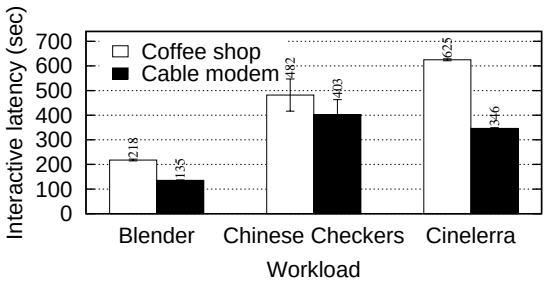


Figure 4: Run times with cold caches and 32 workers. Error bars show 95% confidence interval around displayed mean.

shops, a cell phone company hot-spot, and a restaurant. At each, we characterized the access-link bandwidth and latency to EC2 (which is across the country, in Virginia). Average latency ranged from 121 ms for the cell phone company to 199 ms for one of the coffee shops. Upstream bandwidth occupied a fairly narrow range: from 1300 Kib/s for one of the coffee shops to 1500 Kib/s for the restaurant. Downstream bandwidth also occupied a narrow range: from 1400 Kib/s for the same coffee shop to 1600 Kib/s for the restaurant.

In the experiments that follow, the *coffee-shop* configuration models a round-trip time of 160 ms, an upstream available bandwidth of 1400 Kib/s, and a downstream available bandwidth of 1500 Kib/s. A second *cable-modem* configuration, based on the authors' home offices, models 70 ms RTT, 4 Mib/s upstream, and 16 Mib/s downstream.

In most experiments, the local-computation measurements are run on exactly the same kind of machine as the cluster worker nodes. The exception is §5.2.5, where, out of necessity, the client was a laptop.

#### 5.2.1 Cold cache performance

When caches are cold, UCop is slow. Files are faulted in serially over the bottleneck link, necessitating at least as many RTTs and transfer delays are there are files. Figure 4 shows that warm-up times are from 2 to over 10 minutes. Cinelerra is slowest, with round trips proportional to its 484 paths, and bandwidth costs proportional to its 93 MiB of files.

The evaluations that follow all measure performance once the caches are warm. In each experiment, we first destroy all cached data, then warm the caches by invoking the test application twice. Finally, we collect a data point by timing the application's performance when given a new (uncached) input for the first time: a tweaked Cinelerra edit decision list, a modified Blender model, or a new move in a game of Chinese Checkers. We repeat each experiment 10 times and plot the mean. Around each mean we show, using error bars, the 95% confidence interval for the mean using Student's $t$-distribution.

Cold-cache performance is slow; UCop performs poorly for applications run only once. Of course, the same can be said of client software, whose installation also typically takes several minutes. Though currently unimplemented, caches could be made persistent across cluster instances and even users; clusters would then boot ready-to-use in the common case.

#### 5.2.2 Blender

The first application we test is Blender (§4.2). This test renders a 14.2-MiB model of the Starship Enterprise [41] at HD quality ($1920 \times 1080$). Figure 5 compares the time to run locally with the time to run at various levels of parallelism in UCop.

First, observe that in the local case, rendering takes 137 sec, a duration most users would consider non-interactive and that would cause them to task-switch. Next, observe that in either of our network configurations, a cluster size of two breaks even with the local case; even a small degree of parallelism overcomes the overhead of remote operation. Finally, observe that from the coffee shop, 64 workers render the scene in 20 sec, and, using a cable modem, 64 workers take 18 sec. These results show that even on long-delay, low-bandwidth networks, UCop can perform complex rendering in seconds, turning it from a batch to an interactive operation.

#### 5.2.3 Chinese Checkers

The next experiment measures the time it takes for the Chinese Checkers expert algorithm to make a move. As described earlier, in the local case, Chinese Checkers uses a sequential pruning tree search; for both local and remote tests, we use only one core per machine. We automate the game by driving the expert mode (with and without UCop) against a locally-executed novice opponent. We measure only the time taken to compute the expert's most complex move.

Figure 5 shows the results. Computed locally, the computer's move takes 317 sec, a long enough wait that the game might not be fun. UCop overcomes the remote-processing overhead by degree 3. With a 64-node cluster, the worst-case move time is reduced to 26 sec in the coffee shop, and to 23 sec with a cable modem. This illustrates how UCop can make strategy games enjoyably interactive even at expert levels.

#### 5.2.4 Cinelerra

The third application is the Cinelerra video editor. Because video playback results play out over time, total completion time is not an interesting metric; therefore,

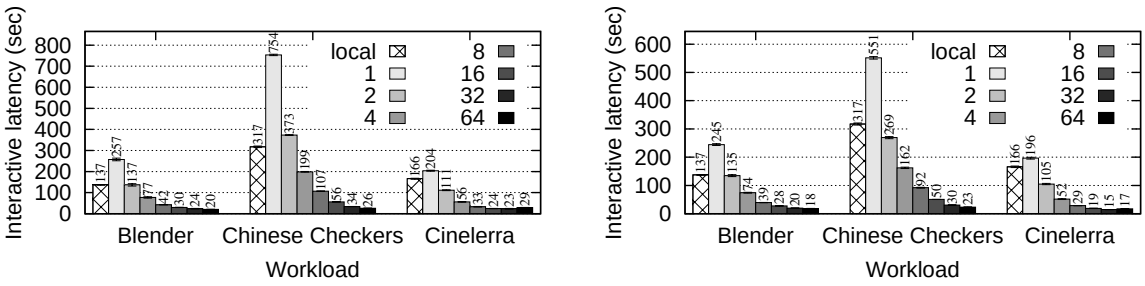


Figure 5: Run times with local computation and varying degrees of UCop parallelism; smaller times are better. Network configurations are *coffee-shop* (left) and *cable-modem* (right). The parallel cluster uses the same class of machine as the local-computation baseline. Error bars show 95% confidence interval around displayed mean.

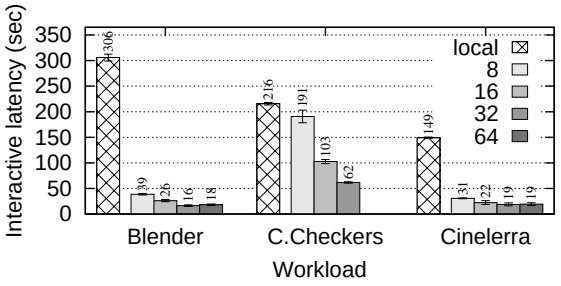


Figure 6: The effect of parallelism when the client is a laptop in a real coffee shop. Note that, unlike in Figure 5, the locally-computed runs execute on a different class of machine than is used in the compute cluster. Error bars show 95% confidence interval around displayed mean.

our measure of interactive latency for Cinelerra is *preroll time*. This is the delay until video playback could theoretically begin and still allow uninterrupted complete playback. The workload is a 20-second clip of the digicam montage described in §4.4.

Figure 5 shows the results. In the local case, the delay is 166 sec, a long time to preview a clip. UCop begins showing benefit at degree 2, readily overcoming the remote overhead. Finally, by degree 32, Cinelerra delivers the same clip in only 23 sec from a coffee shop, or 15 sec using a cable modem.

#### 5.2.5 Tests from a real coffee shop

The previous sections reported experiments done in a controlled environment meant to emulate a coffee shop. We now discuss experiments using an *actual* coffee shop. In these tests, the client is a laptop, a Lenovo z61p with a 2GHz Core Duo running Debian Lenny. The workers are still EC2 cluster machines. Blender is linked against Debian Etch, and thus runs from inside a `kvm` hardware virtual machine, which is limited to one core. Figure 6 shows the results; they are essentially similar to, and thus validate, our earlier emulated-environment results.

#### 5.2.6 Discussion

UCop's goal is to improve interactive performance by achieving low latency; computational *efficiency* is less important. Indeed, speedup per node in these tests peaks at 1–2 nodes (56–85%) and decreases monotonically thereafter, sinking to 12–35% for 32–64 nodes. For this reason, in the following experiments we consider only 32-node clusters, as they provide reasonably low latencies at reasonable cost.

"Reasonable" latency is difficult to define objectively. Results are highly sensitive to workload characteristics; any selection is, to some degree, arbitrary. Had we chosen simpler workloads, the desktop might have rendered them quickly, obviating the need for the remote cluster. More complex workloads favor UCop: Overheads limit UCop's ability to use a larger cluster to reduce wait time, but UCop *can* often use a larger cluster to hold wait time constant as the workload complexity increases dramatically. Similarly, due to the preroll metric, a short Cinelerra clip duration favors local computation, and longer durations favor UCop.

### 5.3 Decomposition

The preceding section shows that UCop is performant. We now break down the contribution of each optimization described in §3. As Figure 7 shows, each optimization is necessary for good performance, although some are more important for certain applications.

**Prethrow.** The first group in both graphs of Figure 7 shows that prethrowing is the most important optimization. Cinelerra, which accesses 484 paths, is most affected by the loss of prethrown attributes. Because files are validated sequentially, the worst slowdown is seen in the highest-latency configuration, *coffee-shop*.

**Remote differential compression.** The next group shows performance when RDC is disabled. That is, every byte of changed files must be uploaded, rather than just the changed bytes. Blender fares worst without this optimization, since its input file is the largest.

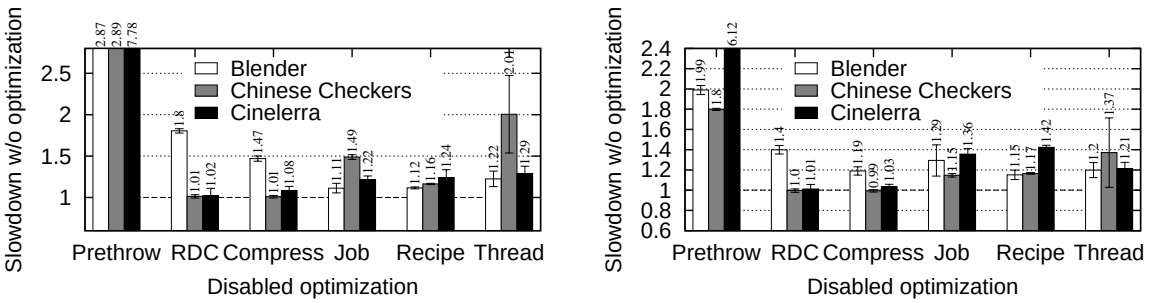


Figure 7: Run times with optimizations disabled. Values are normalized to means in Figure 5, so 1.0 means the optimization has no effect. Network configuration is *coffee-shop* (left) and *cable-modem* (right). Parallelism is 32. Error bars show 95% confidence interval around displayed mean.

**Stream compression.** In the next group, we show the effect of disabling stream compression. Again, this affects Blender the most, since even with RDC, Blender's unstable output requires UCop to transmit 779 KiB of changed blocks; performance suffers in this test because the blocks are compressible.

**Cache sharing.** Without cache sharing within the cluster, all worker nodes must get everything directly from the client via the bottleneck link. This inflates all overheads by a factor equal to the cluster size. Clearly, this is unscalable. Indeed, this experiment ran so slowly that we abandoned collecting statistically-significant data; none appear in the figures.

**Job consistency.** The next group of bars shows the benefit of job-end-to-start consistency. Here, no two tasks share the same job, and hence consistency semantics demand that each task prethrow its own path list. This has no effect on round-trips, but congests the link with duplicate data.

**Recipe caching.** In the next group of bars, the client does not cache recipes as described in §3.5; therefore, the client must compute megabytes of hashes before it can begin the transaction with a prethrow.

**Thread interface.** The last group shows the value of launching UCop tasks from threads rather than processes. Here, the cost is the serialized overhead on the client machine. Chinese Checkers suffers the most because it launches 86 tasks; the other applications launch 32. This optimization will be more important on slow clients (our experimental client is fast), and at higher degrees of parallelism.

## 5.4 Sensitivity analysis

Figures 8 and 9 show the sensitivity of our results to network characteristics. UCop is effective at dramatically reducing sensitivity to poor latency and bandwidth conditions. These experiments also show how UCop achieves this. Without prethrow, performance is highly sensitive to latency. Without compression and RDC, performance is highly sensitive to bandwidth, at least for Blender, which is bandwidth-intensive.

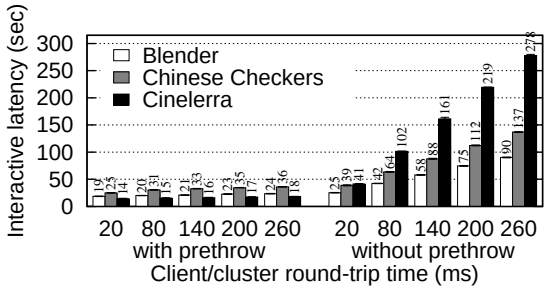


Figure 8: Interactive latencies with varying client/cluster RTT, *cable-modem* available bandwidth, and 32 worker nodes. Smaller times are better. Prethrows make UCop's performance nearly latency-insensitive. Error bars show 95% confidence interval around displayed mean.

## 5.5 Remaining costs

The previous section explained the techniques that achieve interactive-scale performance. This section explores the present limits to performance and how it might be further improved.

### 5.5.1 Latency breakdown

To guide this discussion, Figure 10 provides a rough analysis of how the 20 sec of interactive latency for our Blender workload arose. We estimated the latency components as follows:

**Launch overhead.** We measured `remrun` on EC2 with no artificial latency or bandwidth throttling. It spent about 1 sec launching 32 job requests, including time to `stat` the file system for each prethrow path and fault in missing blocks.

**Process start.** We asked Blender to load the Enterprise model, then exit without rendering anything; this took about 1.5 sec.

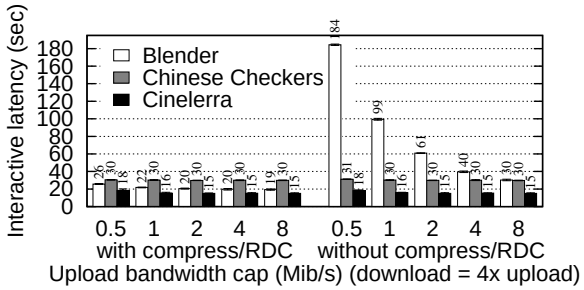


Figure 9: Interactive latencies with varying available bandwidth, *cable-modem* client/cluster RTT, and 32 worker nodes. Smaller times are better. RDC and compression make UCop's performance nearly bandwidth-insensitive. Error bars show 95% confidence interval around displayed mean.

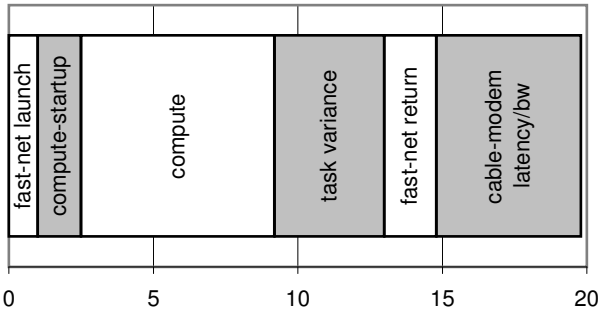


Figure 10: Approximate components of overall latency: Blender on a 32-node cluster.

**Computation.** Rendering one of the 32 tiles takes 6.7 sec on average, with a maximum of 10.5 sec. This variance arises because nodes rendering the blackness of space surrounding the Enterprise become idle long before the most-loaded node puts finishing touches on the glow of the warp nacelles. Thus, we account for the time as 6.7 sec of computation plus 3.8 sec of inter-task variance. Contrary to prior reports [46], we observed no significant effect from inter-*machine* variance; the time required to complete a task correlated with the task's workload, not the machine that ran it.

**Results download.** Our test with an unthrottled EC2 network showed that UCop spends 1.8 sec organizing and returning the result tiles.

**Real network costs.** The costs above account for all but 5 sec of the cable-modem time. We attribute these remaining 5 sec to the network delays due to increased RTT and reduced available bandwidth.

### 5.5.2 Opportunities for improvement

The firmest contributor to latency is the compute time itself. Of course, wider parallelization can help, but the benefit is constrained by inter-task variance and offset by an increase in network launch and return costs.

The four round trips we spend are three more than strictly necessary. Eliminating them as follows could save as much as 0.5 sec in the coffee shop. The prethrow technique already profiles applications to predict paths with stable contents. The same technique could detect repeatedly-used paths with consistently fresh contents, and use that cue to eliminate the RecipeNameRequest round-trip. Reasonably assuming that the RecipeName itself is fresh, the client could pipeline the recipe as well. Finally, by maintaining a shadow index of blocks the cluster already knows, the client could further pipeline the set of new blocks, eliminating a third RTT.

Process startup might be mitigated by checkpointing the client-side process after basic initialization or after parsing stable inputs. In addition, each process's compute time might be predictable in some cases; shorter subtasks might automatically be run locally, or run locally in the case of network failure.

## 6 Conclusions

The Utility Coprocessor is a new use for utility compute clusters: dramatically enhancing the performance of CPU-intensive, parallelizable desktop applications. UCop's non-invasive installation and automatic support for arbitrary client software configurations lets users farm their desktop compute tasks out to the cloud without changing their model of where files are stored or how new software is installed.

The primary challenge in making a system like UCop performant is overcoming the relatively high latency and low bandwidth of the link separating the user's desktop from the compute cloud. We introduce the techniques of task-end-to-start consistency and prethrowing to avoid latency penalties. We avoid bandwidth penalties using a combination of cluster-wide cache sharing, remote differential compression, and the notion of job-end-to-start consistency. We also use a variety of techniques to minimize the client's CPU and I/O load when sending work to a cluster. Taken together, these techniques allow UCop to efficiently execute wide parallel computations in the cloud with low overhead, even though all the canonical state is on the client.

Our evaluation demonstrates speedup with only 2–3 nodes even in a challenging coffee-shop network environment, and 15–20 second interactive performance with 32–64 nodes. We show the necessity of each of UCop's optimizations, and that the optimized system is insensitive to latency and bandwidth variations. We also identify further opportunities for improving performance.

The Utility Coprocessor is a novel and practical system for easily and inexpensively improving the performance of CPU-bound desktop applications. It is general to many applications, and UCop support is easy

for developers to integrate. Thanks to the availability and low cost of utility computing clusters like Amazon EC2, the power of UCopified applications is available to individuals—today.

## References


[1] Blender 3D Modeling Suite. http://blender.org/.

[2] Cinelerra Video Editor. http://cinelerra.org/.

[3] Enabling Grids for E-sciencE (EGEE). http://www.eu-egee.org/.

[4] FUSE: Filesystem in Userspace. http://fuse.sourceforge.net/.

[5] gLite Lightweight Middleware for Grid Computing. http://glite.web.cern.ch/glite/.

[6] Interactive European Grid Project. http://www.i2g.eu/.

[7] ADVANCED CLUSTER SYSTEMS. Math supercomputer-in-a-box. http://www.advancedclustersystems.com/ACS/Products.html.

[8] AMAZON WEB SERVICES. EC2 elastic compute cloud. http://aws.amazon.com/ec2.

[9] ANDERSON, T. E., CULLER, D. E., PATTERSON, D. A., AND THE NOW TEAM. A case for NOW (networks of workstations). *IEEE Micro 15*, 1 (1995), 54–64.

[10] ANTHES, G. Sabre flies to open systems. *Computerworld* (May 2004).

[11] BANKS, T. Web services resource framework (WSRF) primer v1.2. Tech. Rep. wsrf-primer-1.2-primer-cd-02, 2006.

[12] BARAK, A., GUDAY, S., AND WHEELER, R. G. *The MOSIX Distributed Operating System - Load Balancing for UNIX*, vol. 672 of *Lecture Notes in Computer Science*. Springer, 1993.

[13] BASU, S., TALWAR, V., AGARWALLA, B., AND KUMAR, R. Interactive grid architecture for application service providers. In *ICWS* (2003), pp. 365–374.

[14] BAUDE, F., CAROMEL, D., HUET, F., MESTRE, L., AND VAYSSIÈRE, J. Interactive and descriptor-based deployment of object-oriented grid applications. In *HPDC* (2002), pp. 93–102.

[15] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI* (2004), pp. 137–150.

[16] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007), ACM, pp. 205–220.

[17] FORUM, M. P. I. MPI: a message-passing interface standard version 2.1.

[18] FOSTER, I. Globus Toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing* (2006), pp. 2–13.

[19] FOSTER, I., AND KESSELMAN, C. *The Grid - Blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers, 1999.

[20] FOSTER, I. *et al.* The Open Grid Services Architecture, version 1.5. Tech. Rep. GFD-I.080, 2006.

[21] FOUCAUD, F., JOHARY, R., AND TERRAL, J. Bordeaux1 Chinese Checkers. http://sourceforge.net/projects/b1cc.

[22] GRAY, C. G., AND CHERITON, D. R. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP* (1989), pp. 202–210.

[23] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst. 6*, 1 (1988), 51–81.

[24] HUBERT, B. Linux advanced routing & traffic control. http://lartc.org/.

[25] HUMPHREYS, G., HOUSTON, M., NG, R., FRANK, R., AHERN, S., KIRCHNER, P. D., AND KLOSOWSKI, J. T. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH* (2002), pp. 693–702.

[26] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions On Computer Systems 10*, 1 (Feb. 1992), 3–25. http://www.cs.cmu.edu/afs/cs/project/coda/Web/docdir/s13.pdf.

[27] MANN, V., AND PARASHAR, M. DISCOVER: A computational collaboratory for interactive grid applications. In *Grid Computing: Making the Global Infrastructure Reality* (January 2003), John Wiley and Sons, pp. 727–744.

[28] MCCLURE, S., AND WHEELER, R. Mosix: How Linux clusters solve real-world problems. In *USENIX Annual Technical Conference, FREENIX Track* (2000), pp. 49–56.

[29] MILOJICIC, D. S., DOUGLIS, F., PAINDAVEINE, Y., WHEELER, R., AND ZHOU, S. Process migration. *ACM Computing Surveys 32*, 3 (September 2000), 241–299.

[30] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *SOSP* (2001).

[31] NUTTALL, M. Survey of systems providing process or object migration. *Operating Systems Review 28* (1994), 64–80.

[32] OUSTERHOUT, J. K., CHERENSON, A. R., DOUGLIS, F., NELSON, M. N., AND WELCH, B. B. The Sprite network operating system. *Computer 21*, 2 (1988), 23–36.

[33] PETERSEN, D. Loki Blender queue manager. http://sourceforge.net/projects/loki-render/.

[34] PLÓCIENNIK, M., OWSIAK, M., FERNÁNDEZ, E., HEYMANN, E., SENAR, M. A., KENNY, S., COGHLAN, B., STORK, S., HEINZLREITER, P., ROSMANITH, H., PLASENCIA, I. C., AND VALLES, R. Int.eu.grid project approach on supporting interactive applications in grid environment. In *Workshop on Distributed Cooperative Laboratories: Instrumenting the GRID (INGRID)* (April 2007).

[35] POOL, M. distcc, a fast free distributed compiler. In *Linux.conf.au* (2003). http://distcc.samba.org/doc/lca2004/distcc-lca-2004.html.

[36] SANDBERG, S., GOLDBERG, D., KLEIMAN, S., WALSH, D., AND LYON, B. Design and implementation of the Sun network file system. In *USENIX Summer Conference* (June 1985), pp. 119–130.

[37] SMITH, J. M. A survey of process migration mechanisms. *SIGOPS Oper. Syst. Rev. 22*, 3 (1988), 28–40.

[38] STEFANO, A. D., PAPPALARDO, G., SANTORO, C., AND TRAMONTANA, E. Supporting interactive application requirements in a grid environment. In *Workshop on Distributed Cooperative Laboratories: Instrumenting the GRID (INGRID)* (April 2007).

[39] SUNDERAM, V. S. PVM: A framework for parallel distributed computing. In *Concurrency: Practice and Experience* (1990), vol. 2, pp. 315–339. http://www.csm.ornl.gov/pvm/.

[40] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience 17*, 2-4 (2005), 323–356.

[41] THOMAS, W. U.S.S. Enterprise Blender mesh, 2005. http://stblender.iindigo3d.com/meshes_startrek.html.

[42] TRIDGELL, A. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, 1999.

[43] WANG, K. Microsoft Research. Personal communication.

[44] WESTERLUND, A., AND DANIELSSON, J. Arla—a free AFS client. In *Proceedings of the 1998 USENIX, Freenix track* (1998), USENIX.

[45] XCALIBRE. Flexiscale. http://www.flexiscale.com.

[46] ZAHARIA, M., KONWINSKI, A., JOSEPH, A. D., KATZ, R. H., AND STOICA, I. Improving MapReduce performance in heterogeneous environments. In *OSDI* (2008), pp. 29–42.