Exploiting Rich Mobile Environments

Jesse Chang Rajesh Krishna Balan
Mahadev Satyanarayanan

December 2005
CMU-CS-05-199

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was partially supported by the National Science Foundation (NSF) under grant numbers ANI-0081396 and CCR-0205266, and by an equipment grant from the Hewlett-Packard Corporation (HP). Rajesh Balan was supported by an IBM Graduate Fellowship in 2003-2005 and by a USENIX Graduate Fellowship in 2002. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, HP, IBM, USENIX or Carnegie Mellon University.

Keywords: Mobile and Pervasive Computing, Data Decomposition, Rich Mobile Environments

Abstract

Remote execution has commonly been used to allow mobile devices to run large computationally intensive applications such as language translation and optical character recognition. Traditionally, most remote execution systems have concentrated on resource-limited environments. However, with the sharp decrease in the price of computing, it is not inconceivable that environments with an abundance of computing resources will soon become the norm. In this paper, we investigate a solution, known as "data decomposition", that exploits these excess resource to improve application latency. We identify the kinds of applications that might benefit from this solution. We then describe how we extended an existing remote execution system to support data decomposition. We then present results, for three real applications, that show that data decomposition can greatly improve application latency – up to 85% in some cases compared with using just a single remote execution server. We also present results that show the effect of data decomposition in heterogeneous environments. Our results suggest that data decomposition can be used fairly, without global knowledge, by multiple clients.

1 Introduction

In the last few years, there has been a proliferation of small, light mobile devices such as PDAs and cellphones. These devices have become increasingly useful as more and more applications are ported to them. However, the devices themselves only have limited computational power as space, weight, and battery lifetime considerations are more important design considerations. Hence, many useful resource-intensive mobile applications such as language translation and optical character recognition cannot be executed on these devices without performance penalties.

Fortunately, there is a solution. The computational capabilities of these mobile devices can be augmented through the use of remote execution. For example, the resource-intensive application may run completely on the remote server with the mobile device accessing it using ssh, vpn, or similar techniques. Alternatively, the application may run on the mobile device but remote servers may be used to execute the computationally intensive portions of the application.

Previous research on using remote servers has concentrated on cases where there are just enough servers to support the basic requirements of the resource-intensive applications. However, the cost of commodity computing has been decreasing considerably. It is possible, today, to buy a fairly powerful personal computer for just a few hundred dollars – about the same cost as office furniture. It is thus likely that environments with a large number of available servers will become increasingly common. Indeed, such environments already exist in the form of smarts rooms and university environments where server resources are usually abundantly available and underutilized.

In this paper, we investigate whether it is possible to utilize additional servers, over and beyond the basic application requirements, to achieve additional performance improvements – in particular reduced application latency. For example, if the normal application operation requires just two servers, is it possible to achieve lower latency by using four servers?

Specifically, in this paper, we answer the following four questions:

- What kinds of applications can benefit from additional servers?
- How can this benefit be achieved without extensive application modifications?
- What are the performance benefits of using additional servers?
- Can distinct independent mobile clients exploit a common pool of additional servers in a fair manner?

We show that applications can benefit from additional servers as long as they process data that is easy to decompose into independent chunks. This decomposition, known as *data decomposition*, can be performed with relatively little application modification and results in substantial improvements in overall application latency – up to a 85% reduction for some applications. We also present results that show the effect of data decomposition in heterogeneous environments and results that suggest that data decomposition can be used fairly, without global knowledge, by multiple clients.

The rest of this paper is organized as follows: Section 2 present related work and Section 3 describes the kinds of applications that can benefit from data decomposition. In Section 4, we show how data decomposition can be added to an existing remote execution system. We present our validation approach in Section 5 and the actual validation results in Sections 6 to 8.

2 Related Work

The idea of exploiting excess server resources isn't new, but we believe we are the first to apply it to the mobile computing domain (where mobility dictates that the availability of servers will constantly change). In this section, we present a few systems in the parallel and distributed computing domains that have exploited excess resources.

MapReduce [7] is a highly scalable and error resilient production cluster computing system for processing and generating large data sets. It does this by using all available servers to process chunks of work. MapReduce has a very simple programming interface and requires applications to provide only two functions: a Map function that splits data into smaller chunks and a Reduce function that combines the partial results obtained by processing a chunk into full results. This simple API makes it easy to modify applications to to support MapReduce. Many of the ideas presented in this paper are similar in spirit to this work.

River [1] is a data-flow programming environment and I/O substrate for a cluster computing environment. River uses distributed queues, dispersed among the cluster machines, to process application data. River provides a rich programming model and API that application developers could use to specify the composition and location of the distributed queues. A proper specification of the queues allowed River to achieve, by carefully scheduling disk and network transfers, excellent application completion times even in a heterogeneous environment. However, the rich programming interface also made River both harder to use and easier to use incorrectly.

Finally, BAD-FS [5] was designed to orchestrate large, I/O-intensive batch workloads on remote computing clusters distributed across the wide area. It does this by using two distinct components; a storage layer that exposes control of traditionally fixed policies such as caching, consistency, and replication; and a scheduler that exploits this control as necessary for different workloads. Developers are provided with an extended API to precisely specify the control parameters for the storage layer. However, this makes it harder to modify existing applications to use BAD-FS.

3 Applicability of Data Decomposition

In this section, we answer the "What kinds of applications can benefit from additional surrogates?" question posed in the Introduction. Prior systems that exploited additional resources did so by decomposing application data into smaller chunks. This method proves to be suitable for mobile computing applications as well.

Applications that are most suitable for data decomposition have data that can be divided independently. Even if the data cannot be divided independently, data decomposition may still be possible if the contextual information needed to describe each decomposed segment is relatively small and easy to obtain. For example, for a face recognition application, it will be possible to decompose input images into independent pieces only if we can be sure that we have not divided a face in the image between two pieces. On the other hand, it is trivial to decompose the data for an application that converts text into speech. Each distinct word can be converted completely independently from every other word.

In general, the more dependent the data is, the more difficult it is to use data decomposition with that application. The ideal application will have data with easily detectable segment boundaries. To be more precise, the time to inspect the data and find the correct segment boundaries for independently splitting the data into smaller pieces should be much less than the time to execute each smaller piece. The time to recombine partial results into a complete result must also be small. Hence, applications that perform large amounts of computation on large amounts of data (that can be decomposed) are perfectly suited for data decomposition.

One can imagine many types of applications with these characteristics. For example, string matching/manipulation programs like grep and word frequency counter, have input data in text format that can be separated on word boundaries. The partial results from each smaller computation can be combined in a very easy and efficient manner. For grep, the results can simply be concatenated together. For word frequency counter, the frequency counts of each result can be added to obtain the final result.

There are also many computationally-intensive mobile applications that can benefit from data decomposition. In this paper, we show the benefits of data decomposition for three useful mobile applications: an optical character recognition software (OCR), GOCR [9], a text to audio synthesizing tool, Festival-Lite [6], and a language translation tool, Pangloss-Lite [8]. Optical character recognition is highly useful for mobile users. For example, a foreign traveller can take an image of a street sign, perform OCR on it to extract the words, and then translate the words into a known language using a language translator. The image can be broken up into smaller pieces (as long as care is taken to ensure no characters overlap in these pieces) that can be processed independently. Text to audio conversion is incredibly useful as mobile device users may prefer having files read to them instead of having to view them on the devices' small displays. Spoken text is also a viable output modality that can be used in mobile devices designed for the visually impaired. The text to be converted into speech can be broken up into smaller chunks, each containing a smaller number of words, that can be converted independently. Finally, language translation is incredibly useful in allowing mobile users to communicate with locals when travelling in a foreign country. Different sentences or paragraphs can be translated independently.

4 Design of a Data Decomposition System

4.1 Design Considerations

A successful data decomposition system should have the following characteristics:

Provide excellent performance improvements : Data decomposition should result in substantial improvements in application latency.

Easy to use : Application developers should need to do as little as possible to benefit from data decomposition.

Play nice with others : Multiple clients, using data decomposition, in the same overprovisioned environment should avoid affecting the performance of each other.

We address each of these considerations in turn below.

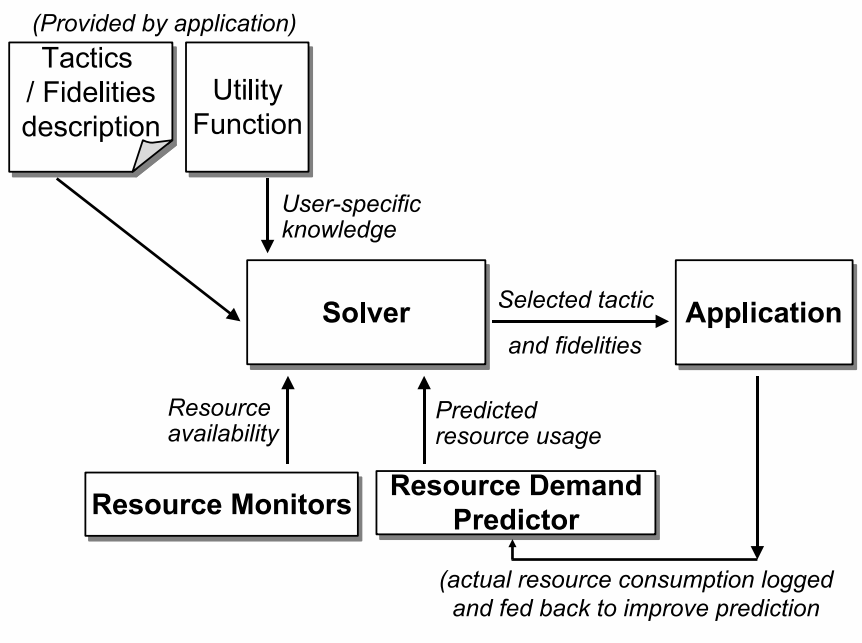


Figure 1: Main Components of Chroma

4.2 Provide Excellent Performance

To be effective, a data decomposition system must be able to dynamically detect available servers and use them to process chunks of application data. These detection and usage steps involve multiple steps: a) detecting a list of possible available servers, b) determining the resource availability (available memory and CPU cycles) on each of these servers, c) determining the expected resource usage for each application chunk, d) selecting a set of servers that can provide adequate performance, e) parallelizing the execution of each application chunk on these selected servers.

Instead of building a new system from scratch, we decided to reuse an existing dynamic remote execution system (that already had most of this functionality) and add data decomposition functionality to it. We used the Chroma [4] runtime as a base as a) it already provides most of the functionality listed above and can provide excellent application performance in mobile environments, b) the Chroma developers have validated that a large number of useful mobile applications can be easily retargeted to use Chroma [3], and c) the source code for Chroma was made available upon our request. We briefly describe Chroma in the next few paragraphs.

4.2.1 Overview of Chroma

Figure 1 shows the main functional components of Chroma. Chroma is an application-aware adaptive remote execution system that dynamically partitions applications among available servers. It can also dynamically decrease the quality or fidelity of applications to reduce resource usage.

To determine the possible application partitions, Chroma uses the notion of *tactics*. Tactics are enumerations of the useful partitions of the applications. In particular, they list the ways of usefully partitioning the computationally intensive portions of the application. Chroma uses a RPC model where applications are partitioned at a modular level. At runtime, Chroma's sophisticated solver

```
OUT INT Resolution; // dynamic fidelity variable that Chroma must set

// definition of RPC procedures that can be remotely executed
RPC a (IN int size, OUT string name1);
RPC b (IN int size, OUT string name2);
RPC c (IN string name1, IN string name2, OUT float result);

TACTIC do_a_and_c = a & c; // Tactic number 1: do RPC c after RPC a
TACTIC do_a_b_and_c = (a, b) & c; // Tactic number 2: do RPCs a and b
                                  // in parallel followed by RPC c
```

Figure 2: Describing an Application's Tactics and Fidelities Using Vivendi

uses predictions of the resource usage of each possible tactic and fidelity combination to determine the optimal tactic, fidelity setting, and server selection that does not exceed the measured available resources (such as the number of available servers, the CPU and memory availability on each of those servers, the available bandwidth, the available battery power, etc.) and which maximizes the target utility function. The utility function precisely captures the preferences of the user. For example, if the user prefers accurate answers over latency, Chroma would run the application at a high fidelity. Conversely, if the user prefers low latency response, Chroma might reduce the fidelity and use a fast remote server (regardless of the energy cost of using such a server).

Chroma also makes it easy for application developers to retarget their applications to use it. Developers first have to describe the tactics and fidelity settings of their application using a simple language called Vivendi. Figure 2 shows an example of such a description. This description is then processed by a smart stub generator that creates most of the code needed to interface the application with Chroma. Finally, the developer inserts a few simple application-specific APIs (created by the stub generator) into the application to complete the process. Balan et al. provides more details and extensive validation of this process [3].

4.2.2 Adding Data Decomposition to Chroma

To successfully use data decomposition, we need to a) determine the available servers that can be used to process chunks of data, b) efficiently split application data into chunks, c) execute each chunk on a different server, and d) efficiently recombine the partial results into a complete result.

We modified the Chroma solver to determine the servers that can be used for processing data chunks. In particular, instead of determining an optimal server selection, the solver was modified to return a list of servers that were "good enough". This set of servers can then be used for data decomposition. We used a user-specified performance threshold to determine if any particular server was good enough.

Because efficient methods to decompose application and to recombine partial results is extremely data-specific, we require application developer to provide a *split* and a *join* function. The split function is used to split application data into smaller chunks. It takes as input the list of server

(determined by the solver) that can be used for decomposition, the group size (explained below), and the input data. It is up to the split function to determine how to split the data and how many servers to use. It returns the split data and the exact number of servers that should be used (this number cannot exceed the number chosen by the solver. The split function may choose to use less servers if there is not enough data to justify using more servers). The join function takes all the partial results and returns a recombined final result. The actual execution of each application chunk, on a different remote server, is performed by Chroma.

4.3 Making Data Decomposition Easy to Use

To make it easy for application developers to use data decomposition, we extended the Vivendi syntax for describing application tactics. For example,

$TACTIC \quad decomp \quad = \quad \%split : ((a,b)\ \&\ c)) : join;$

states that the *decomp* tactic can be decomposed (denoted by the keyword %). The decomposable part of the tactic comprises of executing RPCs *a* and *b* in parallel followed by RPC *c*. The split function is *split* and the join function is *join*. Note that the tactic already contains parallelism. Thus, for this tactic to be most effectively executed, it requires at least 2 servers (the final RPC can be run on the same server as one of the earlier RPCs). The group size for this tactic is thus 2. The number of distinct server groups that can be used to execute application chunks is thus obtained by dividing the number of available servers with the group size. The application data should be split using the number of distinct server groups to avoid server usage collisions.

The stub generator was modified to support data decomposition. In particular, the stub will generate all the intermediate data structures needed to store chunks of data and partial results (the developer will not need to create any data structures). The stub will also ensure that each chunk is correctly executed in parallel. For example, in this case, the stub will generate the code needed to execute the tactic of RPCs *a*, *b*, and *c* in parallel – each with a different chunk of application data. The stub uses the group size to ensure that each chunk is provided enough servers to execute any parallel stages in the tactic without competition from other chunks.

This combination of a simple description language and a stub generator makes it easy for developers to use data decomposition. They will only have to provide the functions to split and join application data. Every other aspect of data decomposition (handling of partial data, buffer management, error recovery, parallel remote execution, etc.) will be automatically stub generated.

4.4 Allowing Multiple Clients to Use Data Decomposition

A key concern with data decomposition is that it easily affect the performance of other clients. For example, if other clients are performing normal execution and a single decomposable client starts using every available server, the performance of the normal clients might suffer. This situation becomes worse if there are multiple decomposable clients all competing for the same set of available servers.

One solution to prevent this is to use strict admission control and resource allocation schemes. However, this scheme requires some sort of central authority for each mobile environment. In this

paper, we investigate whether individual clients can use simple mechanisms that require no global knowledge to achieve fair usage of available resources. In particular, we use simple randomization server selection schemes. We present the effectiveness of these schemes in Section 7.

5 Validation Approach

To validate the usefulness of data decomposition, we used three applications that are representative of the needs of a future mobile user. These applications are all computationally intensive interactive applications that are currently being actively developed for mobile environments. These applications are

- Pangloss-Lite [8] : A natural language translator written in C++ for translating sentences in one language to another. This kind of application is important for the modern mobile user who is moving from country to country. Panlite's data was split at the line level and the translations of each line were combined to form the final result. Panlite translates sentences by using up to 3 translation engines. The output of these engine is processed by a language modeler. The tactic is shown below and has a group size of 3.

  $TACTIC \quad translate \quad = \quad \%split : ((eng\_1, eng\_2, eng\_3) \ \& \ modeler)) : join;$

- GOCR [9] : An optical character recognizer that identifies text in images. This can be used to convert unknown signs into text which can then be translated into a known language. We split the input image into smaller pieces (we used a simple intensity detection algorithm to ensure that we did not cut a possible character into two pieces) and recombined the partial text recognitions to form the final output. GOCR uses a simple tactic with a group size of 1 as shown below.

  $TACTIC \quad recognize \quad = \quad \%split : (ocr) : join;$

- Flite [6] : A program that converts text files into audio. This is very useful as a user interface mechanism and for allowing the visually handicapped access to computers. We split Flite's data at the word level. Flite usually outputs WAV format files. However, as shown in Section 8, these were inefficient to recombine. Hence, we toggled a parameter in Flite such that it output raw format audio files (of slightly larger size than WAV files) that were then easily merged together to form the final audio file. Flite also has a simple tactic with a group size of 1 as shown below.

  $TACTIC \quad convert \quad = \quad \%split : (synthesize) : join;$

5.1 Experimental Platform

We used HP Omnibook 6000 notebooks with 256 MB of memory, a 20 GB hard disk and a 1 GHz Mobile Pentium 3 processor as our main remote servers. We used two different clients that represent the range of computational power available in today's mobile devices. The *fast client* is the above mentioned HP Omnibook 6000 notebook. The *slow client* is an IBM Thinkpad 560X

notebook with 64 MB of memory and a 233 MHz Mobile Pentium MMX CPU. The computational power of the Thinkpad 560X is representative of today's most powerful handheld devices. In some experiments, we also used the slow clients as servers.

The clients and servers ran Linux and were connected via a 100 Mb/s Ethernet network. This was acceptable as the goal was to verify the performance of data decomposition and not the ability of the system to adapt to bandwidth fluctuations. We used a distributed file system to share application code between the clients and servers.

5.2 Success Criteria

To successfully validate the effectiveness of data decomposition, we need to show the following things:

- Data decomposition can greatly improve application performance – in particular application latency.
- Data decomposition can be used in a fair manner without any global knowledge or admission control.

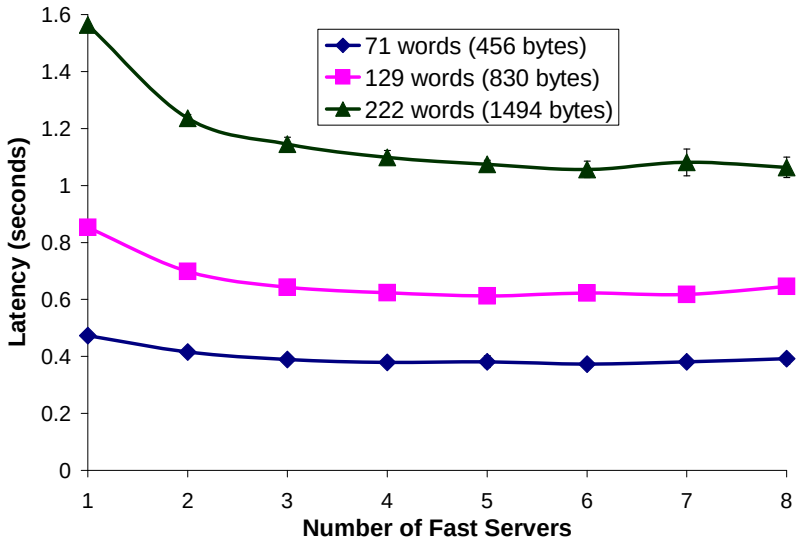
The validation of the first two parts (results shown in Sections 6 and 7 respectively) will justify our claim that data decomposition is valuable and usable. Additionally, we show in Section 8, that the performance of data decomposition depends greatly on the efficiency of the split and join functions used.

6 Results: Data Decomposition is Good

In this section, we show that data decomposition can be effective in reducing application latency. We first show results for the three applications in a homogeneous environments where all the servers are similar. We then repeat the experiment for a realistic heterogeneous environment containing two different types of servers.

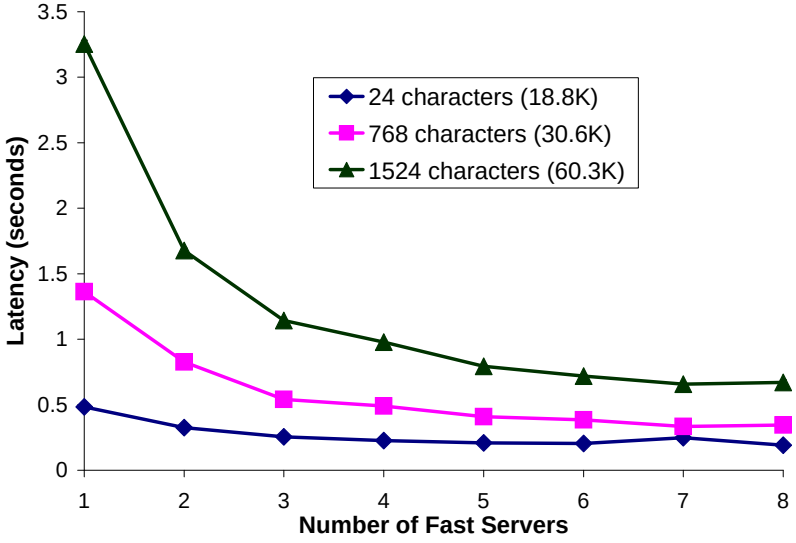
6.1 Homogeneous Environments

Figures 3, 4, and 5 show the results of data decomposition for Flite, GOCR, and Panlite respectively. In all three cases, there were nine available fast servers. We see that data decomposition greatly improved the performance of each application (compared with using just one fast server) – up to a 36% reduction in latency for Flite (when using seven servers and an input size of 222 words) , up to a 75% reduction in latency for GOCR (when using six servers and an input size of 1524 characters), and up to a 66% improvement in latency for Panlite (when using three groups of 3 servers each (since Panlite has a group size of 3) for a total of nine servers and an input size of 603 words).



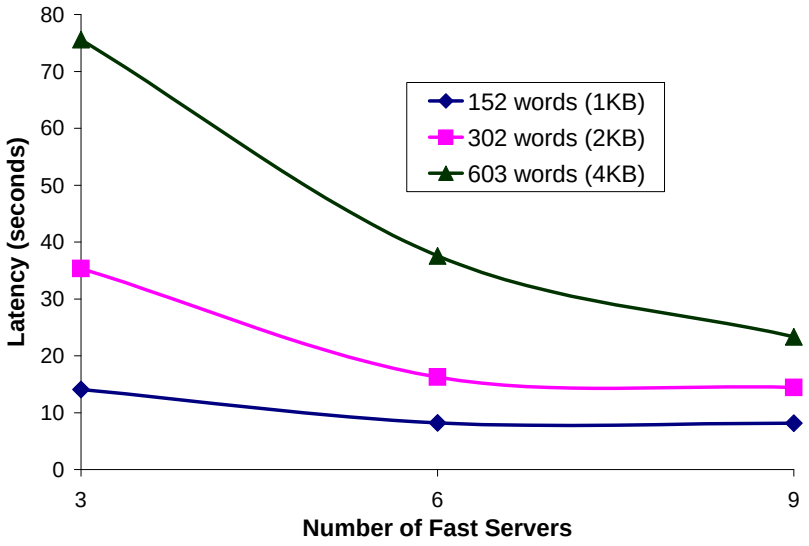
The Figure shows the effect of data decomposition for three different inputs. The x-axis is the number of servers used and the y-axis is the overall application latency.

Figure 3: Performance Improvement for Flite



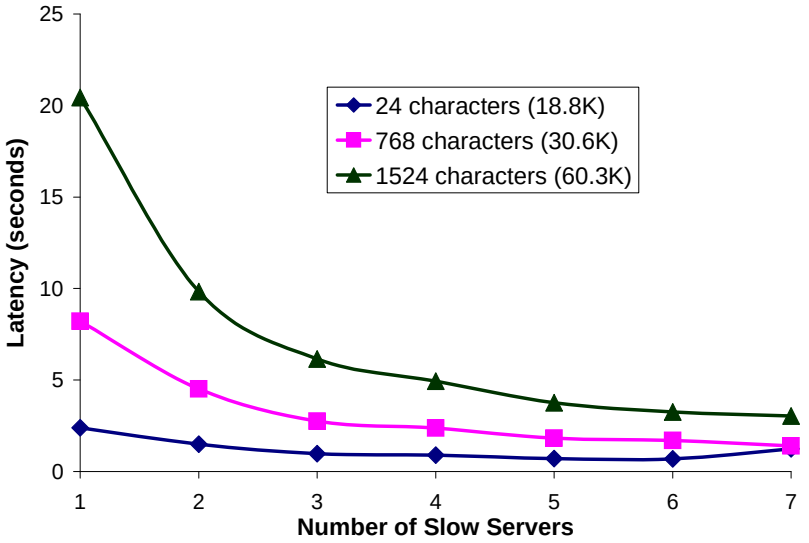
The Figure shows the effect of data decomposition for three different inputs. The x-axis is the number of servers used and the y-axis is the overall application latency.

Figure 4: Performance Improvement for GOCR



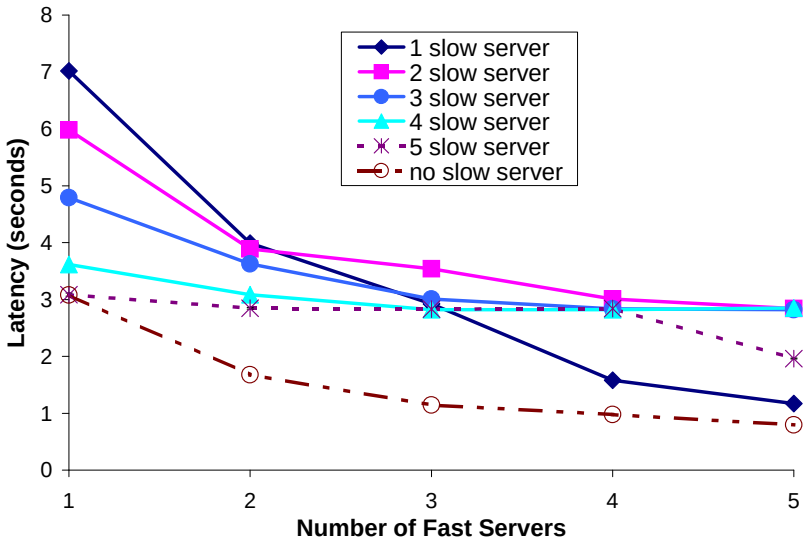
The Figure shows the effect of data decomposition for three different inputs. The x-axis is the number of servers used and the y-axis is the overall application latency. The servers are used in groups of 3 because panlite's tactic has 3 parallel stages.

Figure 5: Performance Improvement for Panlite



The Figure shows the effect of data decomposition for three different inputs. The x-axis is the number of servers used and the y-axis is the overall application latency.

Figure 6: Performance Improvement for GOCR Using Slow Servers



The Figure shows the effect of data decomposition in a heterogeneous environment. The x-axis is the number of fast servers used and the y-axis is the overall application latency. Each line represents the number of slow servers used.

Figure 7: Effect of Naively Using Both Fast and Slow Servers

This latency improvement also occurs even if we just use slow servers. Figure 6 shows the effect of performing data decomposition for GOCR using just slow servers. Compared to using just one slow server, using six slow servers resulted in an 85% improvement in latency for an input size of 1524 characters. Hence, data decomposition is highly effective, when using homogeneous servers, in reducing application latency.

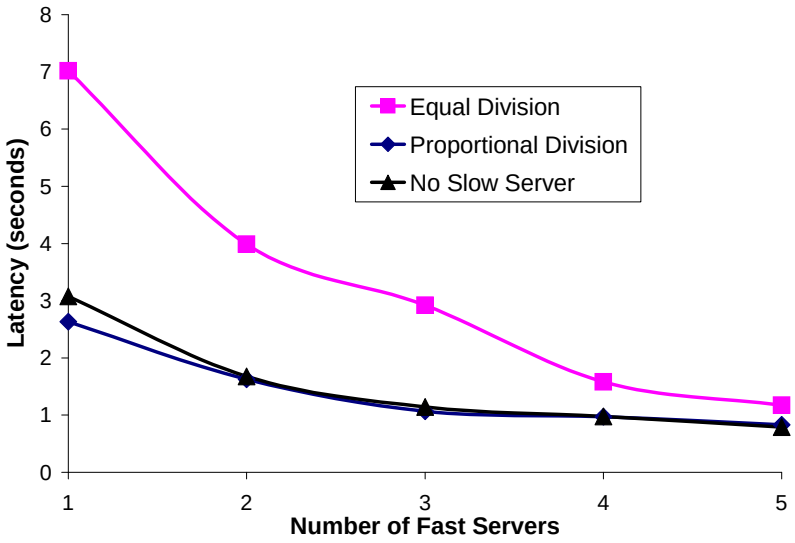
6.2 Heterogeneous Environments

However, the assumption that all servers in the environment are homogeneous may not be realistic. In this section, we investigate the effect of using data decomposition in an environment where there is more than one type of server.

Figure 7 shows the effects of using different servers in a naive fashion. In particular, each server was given the same amount of data to process regardless of its speed. With this naive approach, we see that using slow servers actually hurts the overall latency. This is because the slow server becomes a bottleneck in the system. It is actually better to ignore the computational power of the slow servers and use only a small number of fast servers instead of using a larger number of fast and slow servers.

6.2.1 Solution: Proportional Decomposition

The solution to using a heterogeneous environment is to give each server data in proportion to how powerful it is. Thus, a fast server would get a large chunk of data compared with a slow server. Figure 8 shows the effect of using such a scheme. In this experiment, there was only one slow server being used and up to five fast servers (shown on the x-axis). The bottom line shows the results when the slow server was not used. The other two lines show the naive case where each



The Figure shows the effect of proportional data decomposition in a heterogeneous environment. The x-axis is the number of fast servers used and the y-axis is the overall application latency. The base case is the "No Slow Server" case. The other two lines show the effects of using a single slow server with proportional and equal data division.

Figure 8: Effect of Proportional Data Decomposition

server was given equal amounts of data and the better case where data was proportionally divided between the servers. We see that proportional division allows us to perform slightly better than not using the slow server at all – which is what we wanted as more server resources should improve and not hurt performance.

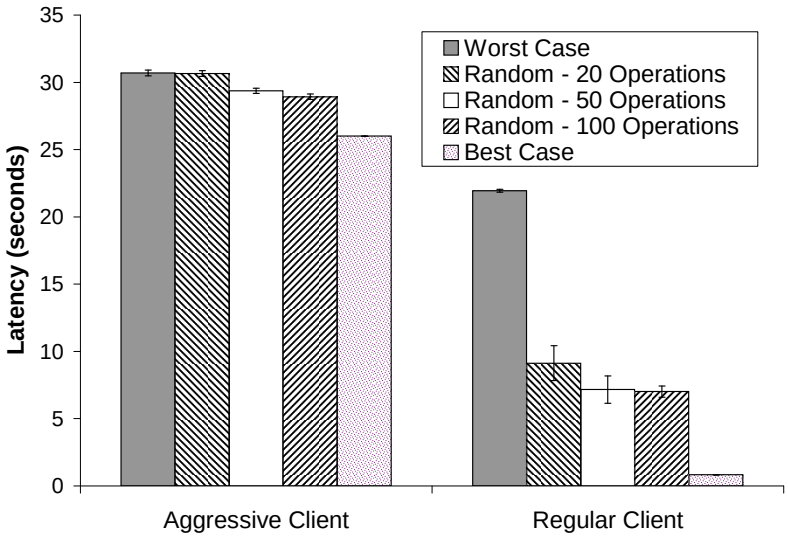
6.3 Summary

In summary, data decomposition can greatly reduce application latency in homogeneous environments. It can also reduce application latency in heterogeneous environments as long as the computing capabilities of the server are considered when deciding how much data to send to that server.

7 Results: Data Decomposition Can be Used Fairly

In situations where multiple clients are present, each client will compete for the same server resources. In this section, we show how using the same servers can affect client performance – in particular for clients that are executing small jobs. We then describe a method that uses purely local decisions (no global coordinator or admission control mechanism) that can prevent clients from using the same servers.

We used GOCR as the application for this experiment. The experiment used two clients. The first was an aggressive client that was continuously recognizing a large image containing 9144 characters. This operation typically takes 120 seconds on a single fast server. The second was a regular client that kept recognizing a small image containing 768 characters every 10 seconds.



The Figure shows the effect of random server selection for two clients (each client selecting 2 servers from a set of 8). The worst case occurs when both clients use the same servers. The next three bars show the effect of random server selection for different numbers of concurrent operations for the regular client. The final bar shows the best case performance where each client uses separate servers.

Figure 9: Effect of Using a Random Server Selection Mechanism

This operation takes 1.36 seconds on average on a single fast server. The environment contained 8 servers and each client was allowed to use up to 2 servers for data decomposition.

7.1 Random Server Selection

In this section, we investigate whether a simple statistical approach is sufficient to ensure that independent clients pick different servers – similar approaches have been used by other systems, such as Ethernet's back-off mechanism, to provide fairness without any central arbiter. Each time a client performs an operation, it randomly selects two servers to use. This makes it very unlikely, when there are abundant server resources, for collisions to occur for every operation. We hope that on average, over a period of time, each client will be able to have a fair share of server usage with minimal collisions.

We ran this experiment using four different cases. In the first case, both clients used exactly the same two servers. This represents the worst case performance. We then ran three different experiments with random selection. For each experiment, we changed the number of operations performed by the regular client. We used 20, 50, and 100 operations. Each experiment was run long enough such that each client made the required number of requests.

Figure 9 shows the results of the experiments. The worst case results shows that using the same servers results in the aggressive client taking an average of 30 seconds (instead of 120 seconds) while the regular client took an average of 23 seconds (instead of 1.36 seconds) to complete. Without competition (results not shown), the average latency for the regular client reduced to 0.82 seconds while the average latency of the aggressive client remained at 30 seconds. Clearly, with competition, only the aggressive client benefited from data decomposition. The regular client actually suffered significant performance degradation.

Random selection helped to reduce the number of collisions and the corresponding performance degradation. In particular, the latency of the regular client decreased significantly with no discernible effect on the aggressive client. As we increased the number of operations, the average latency of the regular client kept decreasing – reducing to 7 seconds when the regular client executed 100 operations. This is still not an ideal result as the regular client can use a single server and complete the operation in only 1.3 seconds (even less with data decomposition and free servers).

This result still provides evidence that it may be possible to use client-side mechanisms, without any global knowledge, to obtain some level of fair resource usage. These mechanisms require a large number of available free servers to be effective. Otherwise, collisions will become too frequent. We believe that improving the statistical methods used by clients to pick servers will result in even better results. It should be noted that a central admission mechanism will be able to achieve much better results (close to the best results). However, such a mechanism may not exist in many environments.

8 Sensitivity Analysis

In this section, we investigate if data decomposition is sensitive to the choice of split and join functions. We used Flite for this experiment. Recall, from Section 3, that Flite was set to return data using a raw audio format instead of the standard WAV file format.

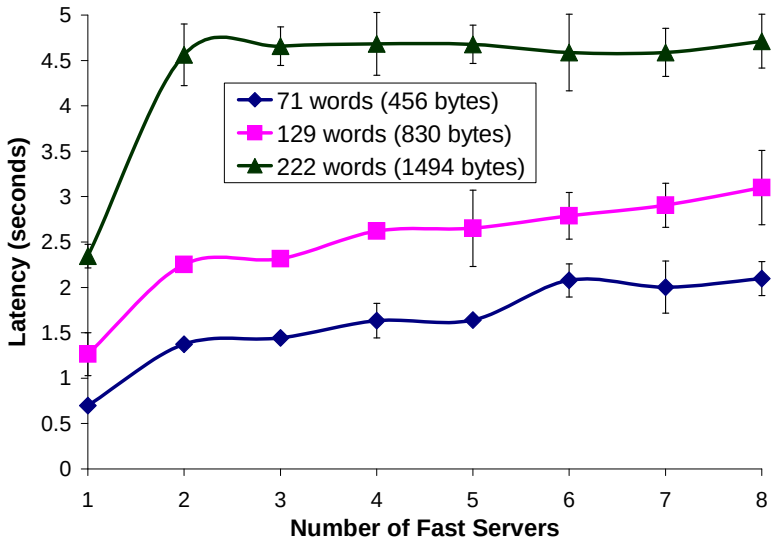
For this experiment, we modified Festival-Lite to return data in WAV format instead. The WAV audio format uses compression and thus requires longer time to recombine partial results compared with recombining raw audio format (where partial files can simple be appended to each other to create the final output). We used a popular audio manipulation tool called SoX [2] to recombine the partial WAV files into the final result.

8.1 Results

We conducted the same experiment for which the results are shown in Figure 3. Figure 10 shows the results when using WAV files instead of raw audio. Comparing the two figures, we notice a huge performance difference. Additionally, the shape of the two graphs even looks completely different. Indeed, the overall latency actually increases as the number of servers increase due to the overhead of recombining WAV files. This result clearly demonstrates that data decomposition is only effective if the time needed to either split application data chunks or join partial results into a final result is small (compared to the execution time of each chunk).

9 Conclusion

This paper was motivated by the observation that even though mobile devices have become increasingly common, they still have limited computational capabilities – weight, power, and size are larger concerns. However, for mobile computing is to become truly useful, these devices have to be able to execute a broad range of computationally-intensive applications. This paper extends



The Figure shows the effect of an inefficient join function for three different inputs. The x-axis is the number of servers used and the y-axis is the overall application latency.

Figure 10: The Effect of Using an Inefficient Join Function

the large body of work in remote execution systems and asked the following question: "Is it possible for mobile devices to use excess server resources (a situation that might become increasingly common due to the low cost of computing hardware) to improve application latency?".

We presented a solution, called data decomposition, that demonstrates that this is possible. We implemented data decomposition into an existing remote execution system and made it easy for applications to use. We then evaluated data decomposition using three real applications – a language translator, an optical character recognizer, and a speech synthesizer. Our evaluation showed that data decomposition can greatly reduce application latency (up to an 85% reduction in some cases). We presented a simple client-side solution that can potentially allow different clients to fairly use a common pool of server resources without any global knowledge. Finally, we also showed that the performance of data decomposition depends greatly on the efficiency of the routines used to split data into chunks and to recombine partial results into the final answer.

References

[1] Arpaci-Dusseau, R. H., Anderson, E., Treuhaft, N., Culler, D. E., Hellerstein, J. M., Patterson, D., and Yelick, K. Cluster i/o with river: Making the fast case common. *Proceedings of the sixth workshop on I/O in parallel and distributed systems (IOPADS)*, Atlanta, GA, May 1999.

[2] Bagwell, C. Sox source code and online documentation. `http://sox.sourceforge.net/`, Feb. 2005. (Version 12.17.5).

[3] Balan, R. K., Gergle, D., Satyanarayanan, M., and Herbsleb, J. Simplifying cyber foraging for mobile devices. Technical Report CMU-CS-05-157R, Carnegie Mellon University, Pittsburgh, PA, Aug. 2005.

[4] Balan, R. K., Satyanarayanan, M., Park, S., and Okoshi, T. Tactics-based remote execution for mobile computing. *Proceedings of the 1st International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 273–286, San Francisco, CA, May 2003.

[5] Bent, J., Thain, D., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H., and Livny, M. Explicit control in the batch-aware distributed file system. *Proceedings of the First USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[6] Black, A. W. and Lenzo, K. A. FLITE: a small fast run-time synthesis engine. `http://www.speech.cs.cmu.edu/flite/`, Feb. 2003. (Version 1.2).

[7] Dean, J. and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[8] Frederking, R. and Brown, R. D. The pangloss-lite machine translation system. *Expanding MT Horizons: Proceedings of the Second Conference of the Association for Machine Translation in the Americas*, pages 268–272, Montreal, Canada, 1996.

[9] Schulenburg, J. GOCR source code and online documentation. `http://jocr.sourceforge.net/`, Feb. 2004. (Version 0.39).