Wanted: Programming Support for Ensuring Responsiveness Despite Resource Variability and Volatility

George H. Forman
Software Technology Laboratory
HPL-98-15
February, 1998

E-mail: gforman@hpl.hp.com

variability in response time, software frameworks, futures, concurrency, multi-threading, multi-resolution, incremental quality of service, dynamic resource allocation, prioritization, task garbage collection, mobile computing, Petra-Flow

Applications running in networked mobile computing environments are prone to a great deal of variability in the response time they experience from system services, such as an (implicitly distributed) file system or (explicit) networking. The range of variability is much greater than in traditional computing environments and changes dynamically. Without special programming considerations, such applications will exhibit unacceptable user responsiveness when resources are slow.

Therefore, applications need to be flexible about their resource demands if they are to remain usable through periods of degraded service. There exist known techniques to cope with resource variability, including incremental (multi-resolution) results, concurrency, and dynamic resource allocation. However, their programming cost is substantial, and infrastructure support is weak. This raises the need for general-purpose software frameworks and mechanisms to support these techniques.

The author briefly describes such a framework. Through the use of annotated futures, it implicitly constructs a data-flow graph of outstanding concurrent tasks, which affords a degree of meta-level reasoning. The experimental implementation capitalizes on this graph by automating some resource allocation activities.

Internal Accession Date Only

1 Introduction

Our purpose here is to discuss the need for programming support to ensure that applications exhibit good response time to the user despite resources that may give variable, and at times, unsatisfactory service. We begin by motivating the problem of variability and the challenge of writing software to cope with it. Later sections give known techniques for coping with variability and suggest a candidate framework that supports these.

1.1 Response Time Variability

It can be extremely aggravating when the programs we use exhibit poor response time, especially if our expectations of responsiveness are dumbfounded by high variability [Nic69,Rus86]. Supposing we download an acclaimed program to our wireless-capable, Java-enabled mobile computer—it is prone to exhibit poor responsiveness in our networked[1] mobile computing environment for a number of reasons:

1. network bandwidth variation between wired and wireless access—to illustrate, there are 5 orders of magnitude difference between 155 Mbps ATM and 9600 bps wireless modem,
2. variable network latency due to sporadic setup delay[2]—mobile computers may have to establish wireless networking on demand to save on connection costs or to recover from lost connections,
3. variable network delays due to wireless interference—intermittent interference may cause reduced or zero bandwidth for brief periods,
4. location-dependent resource variability as we change locations—e.g., in a conference room we may have 2 Mbps wireless network service with a heavy-duty proxy server available, whereas on the road we may resort to a 9600 bps wireless modem and our outdated home machine as proxy server,
5. variations in processing capability—the application might have been written with the expectation of a 266 MHz processor with specialized multimedia processing capabilities in hardware, whereas our mobile device may have no special hardware for multimedia and might (always or at times) process instructions much slower to save battery life and,
6. shared resource variability—shared resources, such as CSMA wireless networks, exhibit varying responsiveness according to current user load, and
7. variable data magnitude—the trend toward sporadic inclusion of multimedia content into some documents can drastically vary their transfer and processing time[3].

Furthermore, applications running on mobile computers are *likely* to be dependent on the performance of the network and remote services. This is because their use is often as a communicator or information utility, and considering their limited computing resources, it is natural to compensate by employing remote services. This dependence on remote services may be transparent to the application, such as with remote file systems [Kis92], or remote virtual memory paging [Sch91].

1.2 Challenge to Programming

There is an inherent tradeoff between response time and the quality of results presented to the user (amount of work done). For example, repainting a window as it is dragged with the mouse in real time gives a higher quality experience than just dragging an outline of the window, but at the cost of less responsive movement if insufficient computing resources are available.

Traditionally, programmers have managed this balance, often unawares, by *static sizing*: they scale the amount of processing their application requires based on its response time on a specific platform, typically their own. Such applications running on less endowed platforms will exhibit poor responsiveness, and on more endowed platforms will forgo opportunities for higher quality. With variable resources, both disadvantages may be experienced at different times.

---

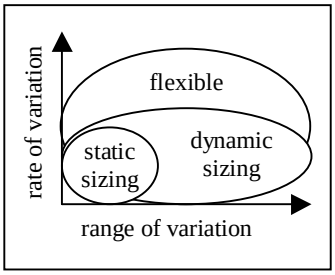
[1] The scope of this paper does not include overcoming delays due to complete network disconnection, an important feature in some mobile computing environments.

[2] CDPD call setup time is a few seconds, whereas analog cellular modems typically require 20-30 seconds.

[3] Some of these issues are more general than mobile computing.

The follow-on is *dynamic sizing*: programmers write their application to size its workload according to the environment. For example, measure network bandwidth and scale the image resolution to be transferred so that it meets a response time target. This lies at the heart of much of the work in quality of service negotiation [e.g., Zha93]. Consider what is required: (1) applications capable of multiple service levels, (2) a predictive model of the response time/workload at each service level, (3) a mechanism to sense the current resources of the environment for selecting a service level, (4) optionally, the capability to make resource reservations, which may have to be revoked later, (5) the capability to monitor an ongoing operation to detect early that it will not meet its objective due to either changes in the environmental resources/reservations or errors in the performance model, and (6) a mechanism to cancel and renegotiate an operation at a different service level, preferably not throwing away work that has already been accomplished.

This is a lot to get working correctly. In addition, the paradigm of quality of service negotiation breaks down in environments where resource variability may be highly dynamic. In a volatile environment, the measurement of current resources has little bearing on the actual resources available when the operation is finally performed. At high volatility, the system may be reduced to thrashing as it constantly renegotiates service levels. The conceptual graph to the right distinguishes variable resource environments along two axes: the range of variation and the rate of variation (*volatility*). Dynamic sizing is appropriate for a wider range of variation than static sizing; however, "flexible" techniques are required to expand into the space of high volatility. The remainder of this paper addresses techniques and support for building applications that can operate effectively in all these regions.



2 Obtaining Responsive Behavior Despite Variability and Volatility

How can applications behave with good responsiveness towards the user, even though the resources they depend on are unpredictable? We forward four principal techniques [For96,Dui90]:

Incremental Results (a.k.a. multi-resolution encoding, progressive transmission/computation): The notion is to do the most important bits first and improve on quality as resources become available, if at all possible, without taking any more time than it would have taken to do the work in the straightforward order. This technique can be applied to a broad variety of domains: images, movies, audio, object graphics, 3-D models, compound documents—even to the order in which portions of a graphical user-interface are drawn.

Concurrency: In traditional, sequential programming, the time to complete each statement depends on the response time of those before it, even if there is no data dependency. This is perilous for response time in an environment where some tasks may unpredictably take a long time. By making tasks independent of each other wherever possible (e.g., all but for data-flow constraints), the opportunity for forward progress is maximized. This is especially important in mobile computing, where the dependence on remote services may be implicit and non-obvious to the programmer, making it difficult to predict the portions of their application that need special attention to ensure non-blocking behavior. We can minimize this risk by exposing as much concurrency as possible.

Dynamic prioritization: As the user's priorities shift, revealed for example by shifting focus between windows, the outstanding tasks associated with each window may be profitably re-prioritized to allocate most resources to what the user is currently attending to, at the cost of delaying tasks that are not of immediate interest.

Cancellation: In an environment where asynchronous tasks may take a long time to finish, newer user actions can cause outstanding tasks to become obsolete. By detecting these situations and terminating obsolete tasks proactively, we can conserve resources, improving responsiveness for the tasks of current interest.

These responsiveness-enhancing techniques come with a significant cost in programming complexity, and so are used only sparingly by today's programmers. Programming support for these techniques by

popular languages and libraries is weak and spotty, at best. Thread interfaces are relatively low level and inconvenient to use for small pieces of concurrent work.

We therefore propose an important area for software research: to develop general purpose frameworks that effectively support programmers in building applications that exhibit good responsiveness in volatile, variable-resource environments, e.g., general support for the techniques listed above.

3 The Petra-Flow Framework

Here we very briefly summarize one such candidate framework, *Petra-Flow* [For96]. The basic notion is to use *futures* to expose concurrency, and extend their capabilities to support the incremental delivery of results to their consumers. The framework supplies implicit data management and synchronization for the incremental results, which conceptually flow down through a data-flow graph.

This bipartite DAG of concurrent tasks and program storage locations is constructed implicitly according to the read/write parameters of annotated asynchronous procedure calls (futures). The graph affords a *global view* of the outstanding work to be done and gives opportunities for standard compiler optimizations. For example, the framework automatically eliminates write-after-write and read-after-write hazards via an analog of *variable renaming*. This avoids having to delay execution of a new task that might overwrite a needed data value. The framework also detects when the old value to be generated is dead, in which case, it performs the equivalent of garbage collection on the graph. *Obsolete* tasks are those on which no program output or variable is dependent. Obsolete tasks might occur, for example, when moving on to a new Web page before the incremental processing of the previous page is complete. Any tasks that become obsolete are signaled to terminate by sending them an exception. Once they have all ended, the anonymous storage locations are recycled.

This global view of the outstanding work to be done and its interdependencies is leveraged in two other important ways: dynamic resource allocation and debugging views. When the relative priority of an output changes, such as when a window is iconified or brought to the foreground, the *Petra-Flow* framework supports propagating this priority change to the upstream tasks, as appropriate to the programmer-selected priority semantics. Such shifts in priority can be propagated to network connections and remote servers, as well.

The main point is that all of these responsiveness-enhancing features are factored out of the applications themselves and implemented just once in a general purpose framework, leveraging software re-use. Details of the *Petra-Flow* framework, a C++ implementation on *pthreads*, and its practical evaluation in three application areas can be found in [For96][4].

4 Conclusion

We believe successful applications must offer the user consistently acceptable response time despite tremendously variable service from the mobile computing environment. Mobility and wireless networking cause much of this variability.

Further, we believe it is both essential and practical to develop programming support for techniques that insulate against service variability. Without this, programmers must expend much more effort to implement such techniques, and so they will be applied only sparingly. This results in software that locks up awkwardly during periods of scarce resources. Which software would you rather have?

Finally, we believe that solutions in this space will find great use beyond mobile computing as well. Stationary computers are facing increased service variability with the ubiquitous use of wide area networking and resource sharing, and the sporadic inclusion of multimedia content. Interestingly, there is a related need in building troubleshooting tools for IT administrators—such tools must operate robustly and with robust performance even when the environment is behaving poorly, unlike typical user applications, which give themselves permission to simply quit or hang if, say, the distributed file system is not working (promptly).

[4] We found that incremental processing of results is particularly prone to lock priority inversion. The thesis also contains a novel locking mechanism that reduces this problem.

Acknowledgements

I am grateful to John Zahorjan for his guidance and clarity of thought in this research. I also wish to thank Keith Moore and Chelliah Muthusamy for their support and valuable feedback on this position paper.

References

[Dui90] D. Duis and J. Johnson. Improving user-interface responsiveness despite performance limitations. In *IEEE COMPCON, Spring '90*, pages 380-386, 1990.

[For96] G. H. Forman. Obtaining responsiveness in resource-variable environments. PhD dissertation, Computer Science & Engineering Dept., Univ. of Washington, 1996. Available at ftp://ftp.cs.washington.edu/tr/1998/01/UW-CSE-98-01-05.PS.Z

[Kis92] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Code file system. *ACM Transactions on Computer Systems*, 10(1):3-25, Feb. 1992.

[Nic69] R. S. Nickerson. Man-computer interaction: a challenge for human factors research. *IEEE Transactions on Man-Machine Systems*, 10:164-180, Dec. 1969.

[Rus86] A. Rushinek and S. F. Rushinek. What makes users happy? *Communications of the ACM*, 29:594-598, 1986.

[Sch91] B. N. Schilit and D. Duchamp. Adaptive remote paging for mobile computers. TR CUCS-004-91, Dept. of Computer Science, Columbia University, Feb. 1991.

[Zha93] L. Zhang, S. Deering, D. Estrin, S. Shenker and D. Zappala. RSVP: A New Resource ReSerVation Protocol. *IEEE Network*, 7(5):8-18, 1993.