CALM, Collected and Disorderly: Distributed Programming in Bloom

Joseph M. Hellerstein
hellerstein@berkeley.edu
University of California, Berkeley

1. INTRODUCTION

Traditional programming models are rooted in the von Neumann architecture: an ordered list of instructions manipulating an ordered array of memory. Given that it is difficult to control or even observe the order of operations across agents in a distributed system, it is no surprise that programming models developed for a von Neumann architecture are a poor match to the disorderly realities of distributed computing.

Research in distributed computing has led to protocols that attempt to impose orderly, von Neumann-style thinking (automata, threads, event loops) on disorderly distributed systems. For example, there is a rich tradition of protocols to manage consistency, including protocols for distributed transactions, atomic broadcast, and consensus. Distributed protocols and their proofs of correctness typically enforce constraints on the order of low-level I/O behavior: reads and writes of data, and processing of messages between nodes.

Despite the power of these protocols, many developers choose to sidestep them, and instead use application-level reasoning to design programs that are robust to disorderly infrastructure. In many cases this design style can lead to leaner systems that perform more predictably and are easier to manage in large-scale deployments [4]. The tradeoff is that application developers must reason about concurrency and make intelligent tradeoffs between consistency, availability and latency. There is a tradition of best practices and design patterns for this approach, but few concrete principles or tools to help programmers ensure the correctness of software they write in this style.

In recent years my group has been pursuing a theoretical framework to address application-level reasoning about distributed consistency, and a practical programming environment that enables programmers to write correct distributed code with judicious use of distributed protocols.

2. THEORY: CALM, CRON, EXPRESSIVITY

A good developer will naturally ask existential questions about the pillars of distributed computing. Why worry about partial orders, group communication or clocks? Is coordination inherently necessary, or is it a design choice? Is it possible to work around these issues and achieve proper program semantics anyhow?

To resolve these kinds of questions, it would be useful to precisely characterize the class of specifications that require expensive distributed mechanisms like coordination, and separate out those specifications that are amenable to efficient workarounds. Model theory and declarative programming provide formalisms for making such characterizations, and we have been able to fairly naturally adapt these formalisms to capture issues in distributed computing [7].

I laid out a set of conjectures about the promise of this approach in an invited talk at PODS 2011 [5]. The initial insight is encapsulated in the CALM Theorem, which knits together Consistency And Logical Monotonicity. CALM begins with the hypothesis that specifications that are purely monotonic (in the sense of monotonic logic) are guaranteed to converge to a consistent deterministic answer independent of execution or message ordering. In addition, non-monotonic programs can be executed consistently by controlling the order of execution only for specific non-monotonic constructs (“points of order”), whose inputs must be sealed before their implications are considered. Two rather different formalizations and proofs of the CALM Theorem appeared in the past year [3, 7].

A corollary to CALM is the CRON conjecture, which questions the underlying motivation for causality. CRON asserts that Causality is Required Only for Non-monotonic constructs. In monotonic programs, eventual program outcomes are not affected by derivations appearing in non-causal orders. As an analogy, consider a child who travels back to the time of his grandparents. This may be surprising, but need not have any effect on the eventual contents of the population. By contrast, consider a child who goes back in time and murders his grandparents before he himself is born; this is paradoxical. Murder, like deletion, is non-monotonic.

It is perhaps surprising that protocols for ensuring distributed consensus and causality are unnecessary for monotonic programs. But surely monotonic programs form only a trivial subset of the tasks we would like to achieve in distributed systems? In fact, a celebrated result of Immerman in descriptive complexity shows that all of PTIME—any polynomial-time algorithm!—can be expressed in monotonic logic [6]. From an asymptotic perspective at least, much of the literature on distributed systems protocols is required only for inherently exponential tasks. All polynomial tasks (i.e. all sensible distributed systems) are somehow amenable to clever coordination-free, correct “workarounds”. This is not to say that such workarounds are easy to find or practically efficient. But the theory should give us pause, and it adds fuel to the fire of developers seeking to avoid paying the penalty of using distributed protocols.

3. PRACTICE: BLOOM

A key goal of our work is to design tools that help programmers provably achieve properties like consistency with minimal use of distributed constructs like coordination. To that end, we designed a language called Bloom, and a CALM-based analysis framework to help Bloom programmers reason about enforcing order in Bloom programs [2]. Bloom is based on a formal logic called Dedalus [1], but we focus here on more practical aspects of the language.

Bloom is a language for programming individual agents that participate in distributed computation. Some of the main features of Bloom include the following:

- A Collected Approach: Taking a cue from successfully parallelized models like SQL, MapReduce, and key-value stores,
Behavior

<table>
<thead>
<tr>
<th>Type</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>table</td>
<td>A collection whose contents persist across timesteps.</td>
</tr>
<tr>
<td>scratch</td>
<td>A collection whose contents persist for only one timestep.</td>
</tr>
<tr>
<td>channel</td>
<td>A scratch collection with one attribute designated as the location specifier.</td>
</tr>
<tr>
<td>periodic</td>
<td>A scratch collection of key-value pairs (id, timestamp).</td>
</tr>
<tr>
<td>interface</td>
<td>A scratch collection specially designated as an interface table.</td>
</tr>
</tbody>
</table>

Meaning

<table>
<thead>
<tr>
<th>Op</th>
<th>Valid lhs types</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>table, scratch</td>
<td>instantaneous merge: lhs includes the content of the rhs in the current timestep.</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>table, scratch</td>
<td>deferred merge: lhs will include the content of the rhs in the next timestep.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>table</td>
<td>deferred delete: items in the rhs will be absent from the lhs at the start of the next timestep.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>channel</td>
<td>asynchronous merge: items in the rhs will appear in the (remote) lhs at some non-deterministic future time.</td>
</tr>
</tbody>
</table>

Figure 1 presents a brief overview of the basic types of collections and merge operators in Bloom.

The standard data structures in Bloom are disorderly collections like sets and relations, rather than scalar variables and ordered structures like lists, trees and arrays. Bloom provides simple, familiar syntax for manipulating these structures. In our current Bloom embedding in Ruby, much of this syntax is borrowed from Ruby, with a taste of MapReduce and SQL.

- Disorderly Programming: Bloom is “disorderly by default”: programmers are led to think about their statements being executed in an arbitrary order. It is possible to enforce order within and across agents in Bloom, but ordering does not arise as a side-effect of the structure of program syntax. It requires clear programmer intent.
- CALM analysis: Using CALM as a guide, a Bloom compiler can usher programmers through the task of judiciously deciding where to enforce order. Bloom program analysis can point out precise points of order in a program: lines of (non-monotonic) code that depend upon ordering constraints to ensure consistency. Programmers can either employ coordination protocols to achieve this order, or tolerate potential inconsistencies and optionally trace their “taint” through the rest of a program.
- Concise yet familiar code: Bloom is a very high-level language with semantic roots in logic programming. As a result, Bloom programs tend to be extremely compact, small enough to easily conceptualize and adapt. But Bloom’s main syntactic constructs for manipulating collections are based on comprehension constructs popularized in languages like Ruby, Python and LINQ. Modern programmers can correctly interpret Bloom program semantics using intuitions from these more operational languages.

Bloom statements are declarative expressions that define the contents of derived collections. They can be viewed operationally as specifying the merge or accumulation of expression results into collections. The syntax is:

<collection-variable> <op> <collection-expression>

A Bloom statement can only reference data that is local to the agent executing the statement. Bloom statements are defined with respect to atomic “timesteps,” which can be implemented in successive rounds of evaluation. At the start of each timestep, the items that populate a collection owe their existence to one of three sources: persistence semantics of tables, local deferred derivations from the previous timestep, or the arrival of messages from external sources (e.g., network channels or system-clock periodic).

The statements in a Bloom program specify the derivation of additional items, which can be declared to exist instantaneously in the current timestep, deferred to the very next timestep, or at some non-deterministic time in the future at a remote Bloom agent.

An example Bloom program is shown in Figure 2. It implements a bare-bones, centralized “chat server”, a la IRC. The state block declares a persistent key-value collection called participants, and two communication channels called msgs (for broadcasting chat messages) and connect (for clients to request a connection to the server) that transport key-value pairs. The Bloom block specifies the two statements that make up the server logic. The first (line 8) unpacks messages from the connect channel and instantaneously merges them (via the <> operator) into the collection of participants. The second statement (line 9) specifies the basic behavior of a chat server: for each client message m that arrives at the server and each participant p in the server’s participants collection, the content of m is asynchronously merged (via the <> operator) into the receiving end of the msgs channels at p.

As a prototype, we have embedded Bloom as a domain-specific language in Ruby that we call Bud. The Bud prototype includes a library of core distributed systems functionality, made up of traditional protocols for message delivery and coordination, as well as application-level examples like distributed key-value stores and chunk storage systems. Figure 3 shows how the chat code above can be embedded in a simple executable Ruby program using Bud.

4. DISCUSSION

The CALM Theorem provides a formal framework for addressing coordination requirements, and points the way to further understanding of when and why classical distributed protocols are required. Languages like Bloom can help in practice to make monotonic constructs more natural for programmers, and to provide semantic cues that enable richer compiler analysis. But many questions remain on both the formal and practical fronts.

Traditional distributed computing centers on models and mechanisms for controlling general-purpose I/O behaviors. In the work we describe here, we shift the focus to languages and compilers that can reason about distributed systems properties in higher-level, application-specific ways. Having made some progress on that front,
it is intriguing to try and circle back to ask whether the lessons learned in the language context can inform the general-purpose systems (and systems theory) viewpoint. For example, one could ask whether the CALM Theorem points to general-purpose protocols that achieve consensus without distributed coordination. Immerman’s construction cited above was not intended to be used in practice (even on a centralized computing model), but it may provide direction toward a more efficient and practical method for our purposes.

Our program analysis contributions to date focus on identifying monotonicity. Like many program properties, monotonicity is undecidable in general, so tests for it are inherently conservative. We would like to enrich Bloom with programming constructs that expose monotonicity even better, enabling more powerful automated analysis. Design patterns like escrow transactions [9] and commutative replicated data types [10] provide some practical grounding here. Using these examples as inspiration, in ongoing work we are generalizing from relational logic to algebras of lattices and commutative monoids that can provide more tools for reasoning about monotonicity.

On the programming front, we hope to exploit our declarative, model-theoretic framework to develop new tools for software development tasks like tracing, debugging and testing. For example, to improve coordination efficiency we would like to measure, given a set of traces, the costs of coordination for a particular specification. We could then ask whether a different specification would guarantee acceptable outcomes with less coordination for that trace. We want to minimize the tracing overhead to achieve this understanding by logging only the events and orderings that are relevant to coordination decisions. In terms of debugging, we would like to improve the ability to fix specifications, not just implementations. For example, we might ask “why” a particular message arrived at a certain location in a certain trace. We might also ask “why not” if expected messages did not arrive. Another interesting opportunity made possible by declarative specification is the ability to synthesize a minimal set of tests with inputs and timings that cover all meaningfully distinct cases for a specification. Recent work in database theory provides a lens on many of these problems that appears promising [8].

More practically, we can ask whether high-level languages like Bloom can be compiled to highly efficient code, for settings like single-rack computation or multicore where performance per node can be as important as network messaging and high-level program properties. Although we see no fundamental obstacles here, the question deserves proof by demonstration. To that end, we are evolving our initial Ruby-based Bloom prototype into a code-generating compiler that can convert Bloom into high-performance, low-latency dataflow code in a variety of target languages.

Finally, it is important in many practical settings to reason about programs that are designed to behave non-deterministically under stress—e.g. during failures or network partitions. In deploying such programs, it is typical to employ additional program logic or extra-computational means (coupons, lawsuits) to compensate for bad outcomes. The overall behavior of such systems merits further formal study, as well as practical design tools.

Acknowledgments

Much of this material developed in the context of joint work in the BOOM project, including Peter Alvaro, Peter Bailis, Ras Bodik, Neil Conway, David Maier, and Bill Marczak. Particular thanks to Alvaro, Conway and Bailis for comments on drafts of this paper. Neil Immerman’s input shaped our thinking regarding the expressivity of monotonic programs.

This work was supported by NSF grants 0917349, 0803690, 0722077, 0713661 and 0435496, Air Force Office of Scientific Research award 22178970-41070-F, the Natural Sciences and Engineering Research Council of Canada, and gifts from Microsoft Research and NTT Multimedia Communications Laboratories.

Figure 3: Ruby driver program for Bloom chat server above.

5. REFERENCES


