BIGDATA:Small:Data Collection and Management: Widespread CALM: Coordination Analysis and Big Data

PI: Joseph M. Hellerstein, U.C. Berkeley

Most architects of modern Big Data infrastructure believe that perfect distributed consistency is too expensive to guarantee in the large [3]. This issue is often discussed in the context of data storage and retrieval, but it is also a major consideration in distributed Big Data computational infrastructure as well—be it application-level business rules [4] or machine learning algorithms [12]. Instead, loosely consistent approaches are often considered to be a better choice, since temporary inconsistencies across machines can often be made to work out at application level.By this reasoning, coordination mechanisms (transactions, locks, quorums, barriers, etc.) should be reserved for infrequent, mission-critical tasks. The cost of this design pattern comes in software complexity: with loosely consistent infrastructure, application programmers must reason explicitly about distributed consistency issues for each high-level application feature.

Like many well-intentioned design maxims, this one is not so easy to translate into practice; all kinds of unavoidable tactical questions pop up. Exactly where in a multi-component system is eventual consistency good enough, and when is coordination truly necessary? How can a developer know that their mission-critical software is not tainted by others' "best-effort" components? How does an architect maintain proper design maxims as software evolves? For example, how can the junior programmer in year n of a project reason about whether their piece of the code maintains the systems overall consistency requirements?

We made some foundational breakthroughs into these thorny questions in recent years. The *CALM Theorem* proves that Consistency can be guaranteed without coordination for programs that are Logically Monotonic. CALM analysis also leads to judicious use of coordination logic: coordination serves exclusively as a guard for non-monotonic reasoning. Following from these ideas, we developed the *Bloom* programming language. Bloom is a "disorderly", data-centric language that discourages programmers from dictating the order of data and operations, since such orderings are expensive to guarantee in distributed systems. Bloom's roots in logic programming enable simple compiler checks for monotonicity, which can be used to inform programmers about when and why to use expensive coordination logic to control order.

The goal of this proposal is to **bring CALM theory to bear on the practices of Big Data developers in the Cloud**, making it easy for distributed programmers to relax their use of coordination and embrace distributed disorder in a natural yet principled fashion. This goal requires significant research to extend the instantiation of CALM theory into the practice of building distributed systems.

Intellectual Merit. Initial versions of CALM and Bloom have been steeped in a "closed world" of logic programming and low-level, fine-grained communication. The work here proposes to extend this research to address challenges in a number of directions: *modeling massive distributed storage as a software component*, including a wide variety of possible consistency semantics; *analyzing monotonicity in familiar data types* that go beyond simple logic predicates and relations; *composing encapsulated, service-oriented APIs using Bloom as a principled orchestration language* that can reason about the composition of API contracts; and *providing CALM-aware software engineering tools* to back up these ideas with efficient debugging and testing frameworks that can help achieve and maintain software correctness over time.

Broader Impacts. Beyond the scientific and engineering impact of the Bloom languages and CALM analysis, this work is intended to affect the way that Big Data software is implemented in the field. To promote understanding of these concepts and adoption of techniques, we will run a Massive Open Online Course (MOOC), to reach out not only to Berkeley students and fellow researchers, but also to developers around the world. We will also continue to engage aggressively in industry, as we have been doing in recent years. Given our history of releasing useful software, we plan to focus a significant fraction of this effort on releasing new versions of Bloom and its compiler analyses that can be used in academia and industry, along with the course materials and videos.

1 Introduction

The ubiquity of both Big Data and Cloud Computing are having profound effects on software development. Scalable data management and distributed computing used to be esoteric topics handled by low-level systems experts. But these issues have now surfaced into the mainstream of software development. The challenges of large distributed systems—concurrency and asynchrony, performance variability, and partial failure—often translate into tricky data management challenges regarding task coordination and data consistency. Given the growing need to wrestle with these challenges, there is increasing pressure to find solutions to the difficulty of distributed programming.

There are two main bodies of work to guide programmers through these issues. The first is the "ACID" foundation of distributed transactions, grounded in the theory of serializable read/write schedules and consensus protocols like Paxos and Two-Phase Commit [6]. These techniques provide strong consistency guarantees, and can help shield programmers from much of the complexity of distributed programming. However, there is a widespread belief that the costs of these mechanisms are too high in many important scenarios where availability and/or low-latency response is critical. As a result, there is a great deal of interest in building distributed software that avoids using these mechanisms.

The second point of reference is a long tradition of research and system development that uses application-specific reasoning to tolerate "loose" consistency arising from flexible ordering of reads, writes and messages (e.g., [5, 7, 8, 13, 14]). This approach enables machines to continue operating in the face of temporary delays, message reordering, and component failures. The challenge with this design style is to ensure that the resulting software tolerates the inconsistencies in a meaningful way, producing acceptable results in all cases. Although there is a body of wisdom and best practices that informs this approach, there are few concrete software development tools that codify these ideas. Hence it is typically unclear what guarantees are provided by systems built in this style, and the resulting code is hard to test and hard to trust.

1.1 Background: CALM and Bloom

Ideally, we would like to have a theoretical frame of reference for high-level application-specific guarantees of consistency, much as serializability gives (inefficiently) to low-level read/write reasoning.In addition, we would like to be able to transfer this frame of reference into practical tools that could radically simplify life for developers. In recent years, we have made significant breakthroughs on both these fronts, which represent a powerful starting point to the broader research agenda in this proposal.

First, we developed the CALM Theorem, which provides a formal connection between monotonic program logic and eventually-consistent outcomes [9]. The CALM Theorem has been formalized in both operational [2, 1] and model-theoretic [11] terms. Intuitively, a monotonic program makes forward progress over time: it never retracts an earlier conclusion in the face of new information. The CALM Theorem demonstrates that monotonic programs are *confluent*—they produce deterministic outputs regardless of non-determinism in network delays and message ordering. As a result, monotonic programs have the property of "eventual consistency": in the fullness of time, once all messages are delivered, the final state of all program replicas will be consistent with the deterministic meaning of the program. This frees programmers from worrying about orders of operation, one of the trickiest issues to reason about and control in distributed systems. CALM also provides insights into managing non-monotonic logic: in particular, if distributed programs *coordinate* (only) before non-monotonic operations, the resulting program will be eventually consistent as well. In short, CALM provides insights into *when* and *why* programmers should use expensive coordination protocols.

Second, following from CALM, we developed the Bloom language, and programmer-facing tools for verifying eventual consistency of user programs based on CALM analysis. Like many other verification tasks, consistency analysis is much more effective in a sufficiently high-level language—in this case, one where the constructs of the language can be translated into an underlying logic. Bloom is our language along these lines.

Given a Bloom program, we can "bless" software modules as monotonic, and hence safe to run without coordination. We can also identify non-monotonic *points of order* in Bloom programs, which can be resolved by either (a) adding packaged coordination logic (e.g. quorum consensus) to enforce the ordering, or (b) augmenting the program to tag downstream data as tainted with potential inconsistency. In addition, we can visualize a module and its points of order in a graphical debugger, to help programmers reason about restructuring their

code for more efficient consistency enforcement.

1.2 Moving Forward: Widespread CALM

Bloom and its CALM analysis have been the subject of growing attention not only in the research community [2, **?**, **?**, **?**], but also within industry: in addition to invited presentations at various industrial workshops [**?**, **?**, **?**], and extensive discussion on social media and blogs [], Bloom was the winner of the TR10 award for "ten technologies most likely to change our world" in 2010 [**?**]. More practically, Bloom and its CALM analysis have enabled us to write and analyze a variety of complex distributed programs compactly and correctly [**?**].

However, in our initial research on applying the CALM foundations in practice, we purposely kept a narrow focus on fine-grained, interactive distributed systems written from scratch in Bloom. This allowed us to validate the utility of our theoretical framework for building practical software in a carefully-designed language. But it restricted our impact to developers willing to adopt Bloom at a low level of the software stack. The goal of this proposal is to aggressively expand the impact of CALM foundations, by exploring a fundamental technical agenda driven by contemporary practices in Big Data. We focus on four key directions:

- **CALM Beyond Sets.** The original formulation of CALM and Bloom only verifies the consistency of programs that compute sets of facts that grow over time (set monotonicity); that is, forward progress is defined according to set containment. As a practical matter, this is overly conservative: it precludes the use of common monotonically-increasing constructs such as timestamps and sequence numbers. This presents a dilemma to developers: either they can translate traditional designs to a (potentially unnatural) set representation and benefit from fine-grained CALM analysis of monotonicity, or they can have natural, inherently monotonic code like increasing timestamps be marked conservatively as a point of order by CALM analysis. We seek to move past this dilemma by generalizing Bloom and CALM analysis beyond sets to work on any *join semi-lattice*. (Section **??**).

- **CALM Service Composition.** Basic CALM analysis works within the scope of a whole program written in Bloom. However, modern software is typically written by composing multiple reusable services—typically exposed via opaque APIs. While CALM analysis provides powerful new insights within a Bloom "sandbox", it offers no help to the practical scenario of building contemporary service-oriented architectures. We therefore hope to modularize CALM analysis in a generic fashion, to reason about the composition of widely-used services written in traditional languages, with Bloom used as a thin layer of "orchestration logic" for service composition. (Section **??**).

- **Staying CALM with Distributed Storage.** The early versions of Bloom represented state exclusively in local collections of data; distributed data sets were to be explicitly implemented by programmers. This has made Bloom unattractive for all but the lowest-level developers of Big Data systems. We intend to expose distributed storage—and its spectrum of consistency guarantees—as a first-class construct in our development and program analysis tools. Because the consistency semantics of distributed storage can vary widely, we would like our program analysis to capture these semantics meaningfully, and enable us to compose correct programs over these various storage semantics, relaxing consistency when acceptable, and demanding it when not. (Section 4).

- **CALM Software Engineering for Distributed Programs.** Even with powerful tools like CALM analysis, programmers sometimes choose to forgo static consistency guarantees of the form provided by monotonic code and distributed coordination. In short, they choose to "cut corners" with respect to formal guarantees, and rely on dynamic tests (exception handling) to deal with any problems. This raises challenges in Quality Assurance (QA) and Software Engineering issues involving the testing and dynamic analysis of complex software. QA over Big Data can be very difficult: there is an enormous space of possible executions to exercise, arises from (a) the non-determinism inherent in asynchronous messaging between unreliable machines, and (b) the incredible diversity of execution scenarios that arise at massive scale, where small-probability events occur on surprisingly small timescales. We believe CALM

analysis can help here to radically prune the space of scenarios to test, and provide succinct, targeted explanations of software hazards for programmers to reason about. (Section 5).

2 CALM Beyond Sets

Intro paragraph. – jmh

- Crib motivation from VLDB submission, shortening the CRDT and Bloom background and moving straight to the idea of solving the type dilemma (as discussed in Intro) by merging them. (Cite Ross/Sagiv along the way.)
- Discuss the idea of extending Bloom logic programming and CALM analysis to disorderly programs over arbitrary lattices. Say we've got an initial prototype of BloomL – can cite TR if we like.
- Give an example of a Bloom rule over sets and a corresponding rule over lmaxes.
- Discuss mappings, commutative functions and homomorphisms. Impact on CALM and delta computation.
- Show the vector clock example in a box.
- Challenge: proving lattice properties. Possible conservative analysis in a nice imperative language, or a subsetted DSL within such a language.,
- Challenge: Lattices and efficiency: garbage collection and lower bounds
- Challenge: Lattices and non-monotonicity: e.g. "odometer" compositions, etc.

2.1 Summary of Tasks and Goals

- **Efficient Evaluation of Lattice programs**. Sentence or 2 here on Lower bounds, zero-copy, etc. – jmh
- **Tools for guaranteed lattice properties**. Sentence or two here on a possible DSL agenda, perhaps subsetting some existing language like Scala. Remind that scope can be small because so much can be done outside the DSL in BloomL via lattice composition (e.g. data structures). – jmh
- **Extend the Bloom prototype to support rich set of built-in for composition.** Sentence or two here on what's involved, including language design and evaluation. Say we've got a first prototype, and highlight remaining challenges. – jmh.
- **Evaluations: KVSs and Collaborative Editors.** Sentence or two here pitching the challenge here, and sketching some milestones/metrics for success – jmh.

3 CALM Service Composition

Goal here is about 2 pages that are cogent at a high level, illustrating focus and ambition. Detailed clarity is less important – jmh

3.1 Summary of Tasks and Goals

Introductory paragraph here. This structure is not dictated by NSF, it's just a way for us to appear organized and focused. It also distills any vagueness above into recognizable deliverables that are worthy of research, even if their details are vague. – jmh

- **Thing One**. We will investigate the previously-sketched challenge of Thing One, produce a prototype implementation, and evaluate it by...
- **Thing Two**. We will investigate the previously-sketched challenge of Thing Two, produce a prototype implementation, and evaluate it by...
- ...
- ...

4 CALM over Distributed Storage

Goal here is about 2 pages that are cogent at a high level, illustrating focus and ambition. Detailed clarity is less important – jmh

4.1 Summary of Tasks and Goals

Introductory paragraph here. This structure is not dictated by NSF, it's just a way for us to appear organized and focused. It also distills any vagueness above into recognizable deliverables that are worthy of research, even if their details are vague. – jmh

- **Thing One**. We will investigate the previously-sketched challenge of Thing One, produce a prototype implementation, and evaluate it by...
- **Thing Two**. We will investigate the previously-sketched challenge of Thing Two, produce a prototype implementation, and evaluate it by...
- ...
- ...

5 CALM Software Quality

Goal here is about 2 pages that are cogent at a high level, illustrating focus and ambition. Detailed clarity is less important – jmh

5.1 Summary of Tasks and Goals

Introductory paragraph here. This structure is not dictated by NSF, it's just a way for us to appear organized and focused. It also distills any vagueness above into recognizable deliverables that are worthy of research, even if their details are vague. – jmh

- **Thing One**. We will investigate the previously-sketched challenge of Thing One, produce a prototype implementation, and evaluate it by...
- **Thing Two**. We will investigate the previously-sketched challenge of Thing Two, produce a prototype implementation, and evaluate it by...
- ...
- ...

6 Summary

The difficulty of distributed programming presents critical challenges to the development of robust and scalable systems for Big Data. Chief among these challenges is the desire to sidestep coordination overheads while guaranteeing acceptable program outcomes. The CALM theorem provided a breakthrough framework for reasoning about these problems. The work we propose here, if successful, would extend the CALM theorem well beyond its initial restricted setting, and use it to define a powerful, principled new framework for robust distributed programming that is consistent with common practice.

7 Evaluation Plan and Metrics

For aspects of the research that involve **mathematical work** (theorems, algorithms, etc.), some of the evaluation will necessarily be mathematical in nature, and judged by peer review in publications.

For aspects of the work that involve **engineering artifacts**, evaluation will entail performance studies using large public data sets, with the software running on multiple computers hosted in commercial cloud services like Amazon EC2. Microbenchmarks may focus on small numbers of computers and modest amounts of data, but there will be a focus on macrobenchmarks that use hundreds or thousands of machines and massive datasets (Terabytes to Petabytes). Although there are no languages with equivalent analysis support, we intend to compare the performance of Bloom programs against other systems when appropriate to validate our approach. Quantitative results from these evaluations will be written up in scholarly papers and judged by peer review in publications.

8 Broader impact and education plan

Beyond the scientific and engineering impact of our proposed system, the impact of this project will be realized by a new educational curriculum and by the ongoing release of open-source code.

8.1 Course Development: Offline and Online

In our early work, we were eager to validate the Bloom language and the CALM coordination tools that go with it. So in Fall 2011, PI Hellerstein and one of his graduate students, Peter Alvaro, taught an undergraduate course on distributed systems at Berkeley called Programming the Cloud (`http://programthecloud.github.com/`). We taught a variety of conceptual issues in distributed systems including distributed clocks and ordering, concurrency control, data replication, data partitioning, commit and consensus protocols, distributed hash tables, and parallel dataflow processing. The students were given assignments to implement many of these features in Bloom using the Bud prototype, including FIFO delivery, two-phase locking, quorum replication, distributed deadlock detection, and two-phase commit. In addition, they broke up into teams to implement a number of more advanced features out of these components including: an alarm server, distributed counters, distributed membership and leader election, multicast, distributed queues, distributed votes, Paxos, and MapReduce. Through this process we identified weaknesses in the Bloom syntax and runtime, and we also found that the language—and its framework for thinking about distributed computing—was relatively natural and powerful for enabling these talented undergraduates to engage in a tangible way with serious issues in distributed systems.

In the coming years we wish to expand the course along the lines described in this proposal: increase the focus on large data sets, expand the language discussion to encompass new monotonic constructs, focus on orchestrating Service-Oriented Architectures in the cloud, and reason about debugging distributed systems in a more organized and better-tooled fashion.

Moreover, we plan to scale the class itself to the Cloud, by hosting it online as a Massive Open Online Course (MOOC) at Coursera.com or a similar facility. The plan is to offer a version of the course in person each year at Berkeley to a moderate audience (50-100 students), and then post lectures and homeworks to the MOOC after a lag of a month or two; the delay will facilitate polishing the material and ensuring that homeworks can be self-managed.

8.2 Capacity Building Activities and Goals

Capacity-building activities will focus on "infrastructure for data storage, access and shared services," and "training and communication strategies." The PI has a track record of delivering significant open source software systems and supporting them to achieve broad usage. Examples include the MADlib in-database analytics library [10] which was adopted by EMC and is used by a variety of their customers, the Telegraph adaptive dataflow system [**?**] which led to a startup company called Truviso (recently acquired by Cisco), and the Data Wrangler system [**?**] which is in wide use in open source and frequently mentioned online as a leading tool for data cleaning and transformation.

In terms of dissemination and training, the PI established the open-source MADlib collaboration between industry and academia, convincing EMC to contribute dedicated engineers [10]. He also has served as an aca-

demic speaker in industry discussions of Big Data at a wide range of venues: EMC Data Science Summit (2011 and 2012), Accel Partners Big Data Conference at Stanford, O'Reilly's Strata conference on Big Data (2011), and talks in industry (Google, LinkedIn). Maintains a widely-read blog on research matters in Data and Computation (databeta.wordpress.com), and served as a guest blogger at popular industry blogs including O'Reilly Radar and GigaOm. Advises a number of Big Data companies (EMC, SurveyMonkey, Platfora, Captricity). Advises a venture fund called Data Collective that focuses on Big Data startups. Participates actively in discussions among Data Scientists, engineers, entrepreneurs and industry watchers on Twitter (@joe_hellerstein).

It is an explicit goal of this proposal to develop the open-source FOOBAR system so it can reach such a wide audience. We will also continue to hold workshops and tutorials to evangelize and support our open source projects, as we have done with GraphLab and D3, in addition to interacting at industry events on Big Data.

8.3 Results from Prior NSF Funded Collaborations Between PIs

Over the last 5 years, PI Hellerstein has received six NSF grants, four of which have been completed or will be completed this year. The most relevant of the completed grants is fill in the MUNDO grant. – jmh Of the remaining two, grant 1016924 focuses on system recovery in cloud computing; it completes in 2013 and has no substantive overlap with this proposal. Grant 0963922 focuses on *Social Data Analysis*, in which groups of people collaborate to analyze data. Hellerstein's prior NSF grants have supported dozens of top-tier publications, successful PhD students, and open source software including the Bloom programming language, which in 2010 was recognized by MIT Technology Review as one of 10 technologies "most likely to change our world".

References

[1] Serge Abiteboul, Meghyn Bienvenu, Alban Galland, and Émilien Antoine. A rule-based language for web data management. In *PODS*, 2011.

[2] Tom J. Ameloot, Frank Neven, and Jan Van den Bussche. Relational transducers for declarative networking. In *PODS*, 2011.

[3] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[4] Shel Finkelstein, Thomas Heinzel, Rainer Brendle, Ike Nassi, and Heinz Ulrich Roggenkemper. Transactional intent. In *CIDR*, 2011.

[5] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD*, 1987.

[6] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[7] Pat Helland. Life beyond Distributed Transactions: an Apostate's Opinion. In *CIDR*, 2007.

[8] Pat Helland and David Campbell. Building on Quicksand. In *CIDR*, 2009.

[9] Joseph M. Hellerstein. The Declarative Imperative: Experiences and Conjectures in Distributed Logic. In *PODS '10: Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems of data*, 2010.

[10] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng ad Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. The MADlib Analytics Library or MAD Skills, the SQL. In *VLDB*, 2012.

[11] William Marczak, Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Confluence analysis for distributed programs: A model-theoretic approach. In *Datalog 2.0*, 2012.

[12] Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[13] Dan Pritchett. BASE: An Acid Alternative. *ACM Queue*, 6(3):48–55, 2008.

[14] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

Data Management Plan

This proposal focuses on the development of programming languages for Big Data, program analysis tools, and distributed systems infrastructure implemented in such languages. There are a variety of use cases for the work that will benefit from experimentation with large data sets. Fortunately, there is a plethora of free, open data sets available in the public domain. We do not intend to produce or curate data during the course of our work.

As discussed in the Project Description, the PI has a strong track record of producing open-source research software that is widely disseminated. We expect to do the same here.

Types of data, samples, physical collections, software, curriculum materials, and other materials to be produced in the course of the project: One of the key goals of this proposal is the development of new versions of Bloom, which will be released to the public under the BSD open source license. The software will include documentation and comments to facilitate reuse. Datasets used for evaluation will be those available in the public domain, from public sources such as Amazon Web Services' Public Data Sets (http://aws.amazon.com/publicdatasets/) and InfoChimps.com.

Standards to be used for data and metadata format and content: The work of this proposal is focused on software development issues surrounding large-scale distributed systems for managing Big Data. Most such systems can be explored in a manner that is indendent of data or metadata formats, and hence we do not expect to develop canonical formats for data.

Policies for access and sharing including provisions for appropriate protection of privacy, confidentiality, security, intellectual property, or other rights or requirements: The software will be available in a public code repository like Github, under an open source license. No privacy concerns are apparent at this point.

Policies and provisions for re-use, re-distribution, and the production of derivatives: Our proposed language will be made available at our group website.

Plans for archiving data, samples, and other research products, and for preservation of access to them: We plan to maintain the primary copy of our software at an online repository such as Github, where it will be permanently archived in their repository.

Software Sharing Plan

As with the PI's previous projects, new versions of the Bloom language and its analysis tools will be released open-source, under the common, liberal BSD license. We will follow the "release early, release often" strategy, where we will early on connect with potential users, and use their feedback to correct direction if needed.

In terms of capabilities, we expect that the releases of Bloom will develop as follows:

- **Year 1**: Extension of Bloom to incorporate arbitrary lattices, not only sets. Extension of CALM analysis to monotonic lattice programs.
- **Year 2**: Support for the use of Bloom as an orchestration langage for composing services and analyzing composite properties. Analysis of the use of wide-area distributed storage as a service within the context of whole-program analysis.
- **Year 3**: Efficient debug and test suites for Bloom that reflect both CALM theory and programmer practicalities.