STATEMENT OF RESEARCH INTERESTS

WILLIAM CHRISTIAN BENTON

> I would therefore like to posit that computing's central challenge, "How not to make a mess of it," has *not* been met. On the contrary, most of our systems are much more complicated than can be considered healthy, and are too messy and chaotic to be used in comfort and confidence.
>
> – *E.W. Dijkstra*, 2001

Developing correct, high-performance software has always been difficult. Two contemporary challenges exacerbate matters further: the ubiquity of multi-core processors, which demand parallel software, and the increasing size and intricacy of software projects.

My research program is motivated by my first-hand experience of these challenges as a developer and researcher. I seek to (1) raise the abstraction level of next-generation software by developing simple yet powerful models and systems; (2) improve legacy software by developing analyses that bring research advances to mainstream languages; and (3) turn novel techniques into tools to evaluate their advantages in the real world.

I have worked most often on problems related to *program analysis*, *type systems*, and *concurrency*, but I am also interested in *declarative* programming systems and *runtime systems and virtual machines* for high-level languages. In the future, I will draw on my experience to design abstractions for concurrency, develop type-based tools for program understanding, and improve the performance of high-level and declarative languages.

Research Experience

Performance, Scalability, & Concurrency

Early in my graduate career, I worked on the Paradyn performance analysis tool and gained a broad range of experience in performance analysis, binary instrumentation, and the low-level details of real systems. I analyzed the scalability of the Paradyn tool's front-end; as a consequence of my findings, I converted the front-end to take advantage of multiprocessor machines and eliminated several bottlenecks. My work increased the responsiveness of the Paradyn front-end and fixed a sixteen-fold slowdown on Paradyn's critical path that had been caused by code in a seemingly innocuous module.

This experience – developing high-performance threadsafe data structures, identifying subtle concurrency bugs, and finding performance problems in a large program – reinforced two insights that have informed my subsequent work: the defects of large programs may be obscure, even to experienced programmers, and managing concurrency can dramatically increase the complexity of software projects.

Program Analysis Frameworks

Effective domain-specific languages (dsls) can eliminate unnecessary software complexity for certain problems. *Program analysis frameworks* simplify the process of designing, implementing, and evaluating program analyses by providing dsl or library support for high-level formalisms like constraint solving or logic programming. Most

extant tools, however, are unsuitable for rapid prototyping; the feedback loop necessary for finding and fixing problems in an analysis specification is simply too large.

I developed the DIMPLE bytecode analysis framework for Java [1] to address this limitation. DIMPLE represents programs as databases of facts and provides a DSL for developing programs that operate on other programs, like analyses, preprocessors, and interpreters. DIMPLE allows researchers to design every aspect of a program analysis – preprocessors, semantics, and inference rules – with *declarative* specifications that are very close to the logical definitions that might appear in a technical paper. DIMPLE is efficient enough to evaluate real analyses on programs with hundreds of thousands of statements, but flexible enough to admit *interactive* refinement of any part of an analysis specification.

Exploiting Object-Level Parallelism

My DISSERTATION WORK focuses on managing the challenge of concurrency. My work is motivated by trends in computer architecture: aggressive superscalar designs are being replaced by *multicore processors*. This trend shifts the responsibility for future performance gains from speculative hardware to parallel software. Some programs, like regular numeric codes, are amenable to known parallelization techniques. However, typical end-user applications enjoy neither a known path to automated extraction of parallel tasks nor the economic incentives to justify finding such tasks manually. Application programmers face a dilemma: developing explicitly parallel code may be infeasible, but sequential programs will underperform on contemporary processors.

Sequential programs are written and executed in some order. Some features of this ordering are *essential* and represent data or control dependencies; changing an essential ordering constraint changes the meaning of the program. *Accidental* features of the ordering simply arise from the way the program is expressed and do not affect its meaning. Most sequential programs exhibit some degree of *implicit parallelism*, meaning that we can execute some statements simultaneously in a way that maintains all essential constraints. The major contribution of my dissertation research is a framework for automatically *finding* implicit parallelism in sequential object-oriented programs and exploiting this parallelism *safely* and *profitably*.

My work proposes *object-level parallelism* (OLP), which enables method invocations on distinct objects to execute concurrently while preserving essential ordering constraints. The intuition behind OLP is that the same properties that make software modular often create implicit parallelism; distinct modules often treat distinct sets of data. Similar models have been proposed for manual construction of concurrent programs [3, 6, 7], but OLP generalizes previous proposals and is better-suited to finding implicit parallelism.

I have developed a *type-based analysis* that discovers dependencies between methods, finds noninterfering program fragments, and identifies methods that are good candidates for *safe* concurrent execution. Safe executions have the same observable behavior as sequential executions. My analysis encodes dependence and modularity properties in an extended type system and infers types that provide a basis for OLP safety guarantees.

Safe, structured parallelism is not sufficient if we are to take advantage of parallel hardware. A parallel decomposition must also be *profitable* and present better performance than the serial version. My framework incorporates a *runtime scheduler* that models method invocation costs and determines whether or not a concurrent invocation is likely to be profitable. I have also investigated *lightweight synchronization mechanisms* for OLP.

Future Agenda

Types, Semantics, & Concurrency

My work has shown how to use a *type-based* analysis to automatically find implicit parallelism in Java programs. Type-based analysis has also been applied to a wide range of problem domains [8]. In fact, the very techniques I have developed are applicable to other problems. For example, both secure information flow and several classical optimizations for object-oriented programs can be specified in terms of modularity analysis. In the future, I will investigate new applications for types and type-based analysis that improve both the performance and the reliability of software.

I will also use what I have learned about type-based analyses, concurrency, and OLP while designing user-visible type systems and abstractions to form the basis of novel programming models for tomorrow's concurrent software. This goal demands a deliberate and careful treatment of competing ideals: limited programmer burden versus tractable inference algorithms, simplicity versus expressivity.

Program Analysis Frameworks

DIMPLE addresses a major problem for researchers by reducing analysis implementation time, but some questions remain: how can we be sure that an analysis is sound with respect to the semantics of the language? What time and space complexity does a new analysis exhibit? In the future, I aim to investigate novel program analysis frameworks that integrate expressive formalisms with *automated reasoning systems* and *operational semantics*, in an effort to provide frameworks that answer these questions.

Declarative & High-Level Languages

DIMPLE has also exposed new avenues for research in declarative programming, since its input databases are substantially larger and more complex in structure than the datasets used by most logic programs. Other researchers used an early prototype of the DIMPLE system to evaluate advances in dynamic indexing [5] and Prolog performance on large databases [4]. I am currently investigating the memory performance of tabled Prolog in an effort to better meet the competing goals of interactive and scalable specification of whole-program analyses. Meeting these goals will improve the state of declarative programming and will thus impact a wide range of application areas beyond analysis.

Research and Technology Transfer

While developing software for drug discovery at SmithKline Beecham in 1999, I saw the need for better concurrency abstractions. I developed a class library that supported *service combinators*, which is a programming model developed by Cardelli and Davies [2]. This model explicitly treats failure and redundancy and facilitates developing concurrent and distributed software. In bringing an idea from a technical paper to my workplace, I saw the potential for research to impact real software.

This ideal has motivated me throughout my career. I expect that my research results will be novel, substantial, and embodied in usable tools for other researchers and for software developers. Software tools bring research advances to working programmers and allow greater confidence in published results. To better publicize available language tools and compiler infrastructures, I created *compiler-tools.org* in 2004.

Interdisciplinary Technology Transfer

I have also developed research ideas that apply core computing research to serve interdisciplinary problems. As a consultant for Thumtronics Ltd in 2005, I participated in

developing and refining a novel audio synthesis algorithm. I was able to find practical application both for my experience in performance evaluation and parallel processing and for several signal processing techniques that I had investigated with Prof. William Sethares as part of my Ph.D. minor at Wisconsin. I took a high-level idea, developed sound extensions, produced a prototype, and made research results accessible to nonspecialist developers in a detailed specification. (A product based on this algorithm will see commercial release alongside Thumtronics' electronic musical instrument.)

Summary

My research program has treated a broad range of questions related to software and systems. I expect to continue by developing novel techniques, tools, and systems to help manage the complexity of contemporary software. I plan to conduct research that considers both the ideal world and the real world: both the future of software and current mainstream systems. I am committed to turning research into technology, and I look forward to cross-disciplinary collaborations with my future colleagues.

References

1. William C. Benton and Charles N. Fischer, *Interactive, scalable, declarative program analysis: from prototype to implementation*, Proceedings of the 9th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (Wrocław, Poland) (Michael Leuschel and Andreas Podelski, eds.), ACM, 2007, pp. 13–24.
2. Luca Cardelli and Rowan Davies, *Service combinators for web computing*, IEEE Transactions on Software Engineering **25** (1999), no. 3, 309–316.
3. Denis Caromel, *Toward a method of object-oriented concurrent programming*, Communications of the ACM **36** (1993), no. 9, 90–102.
4. Vítor Santos Costa, *Prolog performance on larger datasets*, Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages (Michael Hanus, ed.), Lecture Notes in Computer Science, vol. 4354, Springer, 2007, pp. 185–199.
5. Vítor Santos Costa, Konstantinos F. Sagonas, and Ricardo Lopes, *Demand-driven indexing of prolog clauses*, Proceedings of the 23rd International Conference Logic Programming (Verónica Dahl and Ilkka Niemelä, eds.), Lecture Notes in Computer Science, vol. 4670, Springer, 2007, pp. 395–409.
6. Robert H. Halstead, Jr., *Multilisp: a language for concurrent symbolic computation*, ACM Transactions on Programming Languages and Systems (TOPLAS) **7** (1985), no. 4, 501–538.
7. Bertrand Meyer, *Systematic concurrent object-oriented programming*, Communications of the ACM **36** (1993), 56–80.
8. Jens Palsberg, *Type-based analysis and applications*, 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01) (Snowbird, UT), 2001, pp. 20–27.