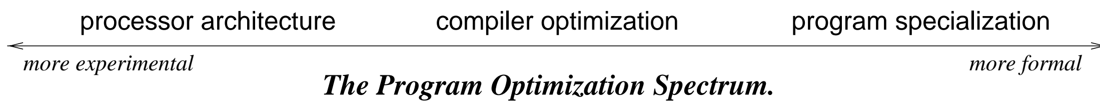
The Program Optimization Spectrum

A Research Statement

Rastislav Bodik

My current interests span programming languages, compilers and computer systems, with emphasis on *program optimization*. Contrary to accepted views, the area of program optimization is broad, multifaceted, almost interdisciplinary. Its spectrum extends from *compiler optimization* to the more experimental *processor architecture* and to the more formally treated *program specialization*. While my thesis focused on compiler optimization, I have been exploring relationships among all three fields.

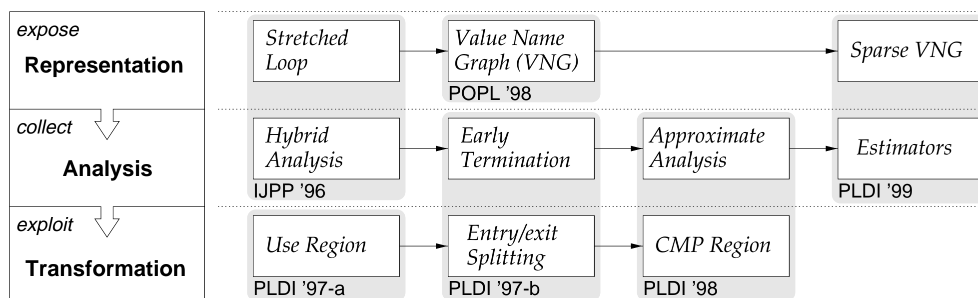


The Program Optimization Spectrum.

The breadth of program optimization offers two ingredients that fuel my research motivation. First, by learning from both empirical and formal methods, I am able to be a balanced systems researcher, one that closely ties theory with practice. This balance is evident in my thesis, where several practical problems previously treated only heuristically received a simple, clean theory. Second, by attacking a common goal—program execution speedup, these three fields weave together insight into a fundamental computer science question: how to efficiently execute programs.

Dissertation Thesis. I developed a compiler optimization framework for removing instructions that redundantly recompute values previously computed by the program. This paradigm of redundancy removal is common to a large class of important optimizations. My framework unifies them and improves their *power* and *practicality*. The improvement results from a *path-sensitive* approach: my algorithms focus closely on each possible program execution path (power) without paying the price of treating each path individually (practicality). To remain practical, previous techniques merged paths, diluting their individual optimization opportunities and sacrificing some power.

The first contribution of my thesis was distilling the independent issues of a redundancy optimizer into three stages: *representation*, *analysis*, and *transformation*. First, the representation *exposes* the redundancies in the program. The analysis then traverses the graph



representation and *collects* the exposed redundancies. Finally, the transformation *exploits* collected optimization opportunities by transforming the program.

While these stages seem traditional, previous research in redundancy optimization blended them, hindering a formulation of a path-sensitive framework. For my thesis, the separation was consequential. With clear independent goals, I was able to deliver for each stage state-of-the-art algorithms, often optimal under these goals. Each of these algorithms is a contribution per se, as outlined below. As a whole, these algorithms make up a framework that is *parameterizable* and *powerful*: the framework was used to formulate and implement the removal of arithmetic instructions [7], conditional branches [5], and loads [8]. The power of the framework subsumes existing methods, and for the important problem of redundant load removal, is close to ideal [8].

1. Representation. The first stage represents the program as a graph that exposes redundant recomputations. An old instance of this general compiler problem is how to expose repeated array accesses in scientific loops. For this instance, I developed *Stretched Loop*, a representation that works as a domain transformer. It converts loops with arrays (vectors) into acyclic code with ordinary variables (scalars). The simpler domain (acyclic and scalar) offers optimal algorithms for redundancy removal. By applying them on the stretched loop, the array redundancy optimization became *efficient* and *optimal*; the previous best algorithm was complex and heuristic [9].

The stretched loop is a successful domain transformer but is restricted to very regular loops. The goal of a general framework is to accept arbitrary programs and expose redundancy of all instructions: arithmetic, loads, conditional branches, etc. My generalized representation, *Value Name Graph* (VNG), has gone beyond that goal. It fused three orthogonal techniques that each exposed a different class of redundancies. Until VNG was developed, three separate algorithms were needed to capture them all. My integration preserves their individual advantages in a mutually beneficial way, within a single algorithm.

The VNG is powerful but does not scale well. To handle large programs, I developed a *Sparse VNG*, which is up to 30-times smaller, without any loss of precision. Furthermore, being a derivative of the popular SSA program form, the sparse version enables retrofitting the extra power of the VNG into existing SSA-based implementations.

2. Analysis. The data-flow analysis stage traverses the representation and groups instructions that compute the same value. One research challenge is to reduce the cost of analyzing a large representation. For the stretched loop, I developed a *Hybrid Analysis* that combines *elimination* and *iteration* analyzers: elimination partitions the representation; iteration analyzes the smaller parts in parallel.

Early Termination is another novel cost-reducing method. It terminates the analysis prematurely, before all redundancies are collected. Ideally, we want to stop the analysis when additional information cannot benefit the subsequent transformation stage. I developed a simple heuristic that provided an order of magnitude analysis speedup, at a small loss of optimization opportunities. *Approximate Analysis* replaces the heuristic with a discipline that steers the analysis effort towards frequently executed paths. Based on run-time program profile, the technique guarantees that the imprecision due to early termination is below a predetermined degree.

In addition to collecting the redundancies, the analysis estimates their optimization benefit, by computing their run-time amount for a given program profile. Unfortunately, such benefit estimates are impaired by profiles' inherent inability to precisely reconstruct frequencies of program paths. While existing profile-directed optimizations disregarded the profiling error, my *Estimators* compute its bounds. The estimators form a hierarchy: the more complex, the tighter its bounds. In practice, even the intermediate ones provide profile-directed optimization with high level of confidence.

3. Transformation. The final goal is to *completely* remove redundancies collected by the analysis. However, for several classes of redundancies, completeness was beyond prior techniques. One such class were instructions whose results may never be used in the remainder of the program, depending on subsequent program decisions. Their optimization thus depends on the unknown future of the program, rather than on its known past. Based on a new abstraction called *Use Region*, my algorithm is able to reorder statements so that the future program path is sufficiently known before the potentially redundant statement is executed, thus enabling its complete optimization.

The next class were redundancies that cross program procedures, for example error checks in library procedures. These checks are performed in each call, often redundantly. To reuse computed values across procedures, the *Entry/exit Splitting* transformation creates additional procedure *entry* and *exit* points. Through each such entry, the procedure obtains a known value which can be reused, rather than recomputed. Exits have similar function. Previously, complete interprocedural removal had to use procedure inlining. My transformation is applicable even when inlining fails, enabling optimizations of object-oriented virtual calls, which may often be uninlinable.

Unfortunately, complete removal involves *code duplication*, which may inflict harmful code-size bloat. To achieve practical completeness, I developed a transformation method that resorts to code duplication merely to enable the more economical *code motion*, minimizing code growth. The extent of duplication is indicated by *code-motion-preventing (CMP) region*, an abstraction for integrating transformations. When an execution profile is available, the CMP region can fuse duplication and motion with *speculation*, optimally balancing their orthogonal characteristics. The result is a very successful profile-directed redundancy elimination: a near-complete removal at near-zero code growth.

4. Experimental Evaluation. I formulated and implemented several instances of the framework. For redundant loads, I performed a simulation-based limit study which showed that as much as 55% of executed loads are redundant. My framework removed more than 80% of this ideal amount.

Future Research Plans. My plans emerge from the observation that *compiler optimization* is just one technology for program optimization. Other related technologies are *processor architecture* and *program specialization*. For redundancy removal, the three offer complementary solutions. My goal is to explore relationships among these solutions and develop new ones, building on contributions of my thesis and architecture background. As outlined below, such new solutions are both enabled and demanded by emerging computer technologies.

These three methods differ in how they divide optimization work between compile time (*static* work) and run time (*dynamic* work). The division has a profound influence: static methods cannot exploit the program input available only at run time, but can be more complex and slower than dynamic ones. Compiler optimizations are static: the program is analyzed and rebuilt. The architecture is dynamic: on the fly, the processor "learns" about the program and predicts future results of its instructions, effectively removing predictable (redundant) ones [12, 13]. Dynamic versions of program specialization are hybrids: static analysis finds values unchanging during the execution; once the concrete values are known at run time, they are hardcoded into the program, specializing it for its current input [11].

Each method has unique strengths, none can subsume the others. Therefore, their integration is mutually beneficial [1]. This observation is not widely recognized, and the respective communities do not cooperate enough. Yet, fueled by technology changes, the integration will eventually take place. My goals is to complement the impact of technology on shaping the integration with a careful consideration of fundamental optimization principles. Specifically, my goal is to understand the static-dynamic nature of optimization and, next, exploit it with properly balanced techniques. My ultimate quest is a suitable paradigm for efficiently executing computer programs. Below are projects leading towards these goals, listed from more static to more dynamic approaches.

1. Redundancy removal of loops and procedures. My thesis focused on redundancy of individual statements. Redundant loops or procedure calls were not recognized. Extending the optimization to such larger program constructs will benefit programs written using object-oriented technology, where large-grain redundancy may occur frequently.

2. New paradigms for dynamic optimizations. The advent of mobile Java code necessitates optimizing programs as they are running. While up to a ten-fold speedup can be gained with dynamic program specialization [11], the same holds for instruction-level parallelism methods (ILP), which are static [10]. These two approaches are orthogonal and should both be exploited. Unfortunately, ILP methods are too costly for run time. A careful combination of compiler optimizations and dynamic program specialization may help by planting into run time only optimizations that are uniquely dynamic.

3. Observational analysis. Static compiler analysis examines the program abstractly, without executing it. Current dynamic optimizers analyze the same way, only faster. To prove program properties, they examine only the code, not the values it computes. This is a waste of run-time possibilities: besides examining an *abstracted* execution, they could also observe the *concrete* one. My goal is to design such a dynamic analysis. Based on observation of computed values, it may find opportunities invisible in the program code alone and also be cheaper than pure abstract analysis.

4. Hybrid hardware-software optimizations. Hardware prediction of values is efficient for simple redundancies. To find correlations between instructions, much hardware is needed [12]. A hybrid with compiler technology may help. A static analysis will find correlated pairs and the hardware will carry out the transformation, by remembering the generated value sequence.

Thanks to embedded computing, hybrid optimizations can be brought to life and to the market. Through the emerging hardware-software co-design technology, we can smuggle onto the chip non-traditional features to support the optimization. As a result, the low cost embedded processors might enjoy some of the server-class power.

5. Redundancy-centric processors. The dependence of successful modern processors on hardware prediction indicates huge amounts of redundancy in programs: what can be predicted is redundant! Future processors should perhaps be redundancy-centric. Instead of learning and predicting, they could analyze the program, avoiding the penalty paid at each misprediction.

The Program Optimization Spectrum *by M.C. Escher.*

The picture shows program optimization, its various characters, and their metamorphosis. Optimization techniques have contrasting characters but no borders. They morph into one another, allowing a fusion that is mutually supportive. Exploring the spectrum between and beyond the existing technologies may produce a spectrum of new techniques whose implementation may be biased either to hardware or software, resulting in different properties.
Left: dynamic (run time) optimizations. Performed by the processor, they adapt on the fly to changes in program behavior. By observing computed values, they also exploit the input.
Right: static (compile time) optimizations. They have zero run-time cost but no run-time adaptability. Furthermore, at compile time only the program code (but not the input) is exploited.

EXISTING TECHNOLOGIES

Dynamic: learn and predict values ***on the fly***. Instructions (***birds***) quickly adapt to changing program patterns.

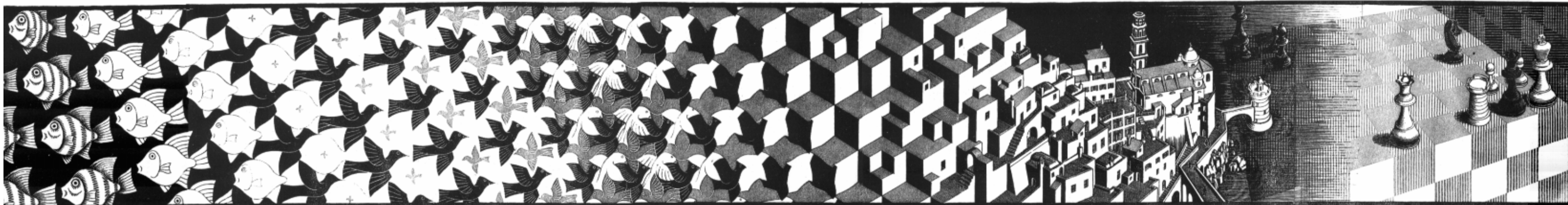
processor architecture ↓

Mixed: static analysis, dynamic rebuild shaped (specialized) by the input. The rebuild is ***simpler***, ***cheaper***.

dynamic program specialization ↓

Static: analyze and re***build*** the program before it runs. During execution, its structure remains ***fixed, static***.

compiler optimization ↓



↑ **redundancy-centric processors**

Processor paradigm shifts: instructions aren't ***birds***, but ***a different kind of animal***, hopefully smarter.

↑ **hybrid hw-sw optimizations**

Hardware remembers repeating value sequences, compiler directs their exploitation.

⟷ **observational analysis**

Analyze running program by observing its values, not only by examining its code. Cheaper and input-sensitive.

↑ **new dynamic optimizations**

Maximize static work, leaving only uniquely dynamic optimizations to run time.

↑ **redundant loop elimination**

Extend the rebuild. Besides statements, achieve removal of redundant loops—a ***holy grail*** of compiler optimization.

↑ **alias analysis** done by others

Loop removal must carefully ***consider each move***, calling for a good alias analysis.

MY FUTURE PLANS

M.C.Escher's "Metamorphosis II" (detail) © 1999 Cordon Art B.V. - Baarn - Holland. All rights reserved.

References

[1] Sarita V. Adve, Doug Burger, Rudolf Eigenmann, Alasdair Rawsthorne, Michael D. Smith, Catherine H. Gebotys, Mahmut T. Kandemir, David J. Lilja, Alok N. Choudhary, Jesse Z. Fang, and Pen-Chung Yew. Theme feature: Changing interaction of compiler and architecture. *Computer*, 30(12):51–58, December 1997.

[2] Rastislav Bodik and Sadun Anik. **POPL'98**, Path-sensitive value-flow analysis. In *Conference Record of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 237–251, January 1998.

[3] Rastislav Bodik and Rajiv Gupta. **IJPP'96**, Array data flow analysis for load-store optimizations in fine-grain architectures. *International Journal of Parallel Programming*, 24(6):481–512, December 1996.

[4] Rastislav Bodik and Rajiv Gupta. **PLDI'97-a**, Partial dead code elimination using slicing transformations. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 159–170, June 1997.

[5] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. **PLDI'97-b**, Interprocedural conditional branch elimination. In *Proceedings of the ACM SIGPLAN '97 Conf. on Prog. Language Design and Impl.*, pages 146–158, June 1997.

[6] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Refining data flow information using infeasible paths. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*. LNCS Nr. 1301, Springer–Verlag, September 1997.

[7] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. **PLDI'98**, Complete removal of redundant expressions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 1–14, June 1998.

[8] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. **PLDI'99**, Load-reuse analysis: Design and evaluation. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (to appear)*, May 1998.

[9] Steve Carr and Ken Kennedy. Scalar replacement in the presence of conditional control flow. *Software Practice and Experience*, 24(1):51–77, January 1994.

[10] W.W. Hwu *et al*. Compiler technology for future microprocessors. *IEEE*, 83:1625–1640, 1995.

[11] B. Grant, M. Mock, M. Philipose, C. Chambers, and S.J. Eggers. The UW Dynamic Compilation Project. Technical Report `http://www.cs.washington.edu/research/projects/-unisw/DynComp/www`, University of Washington, 1998.

[12] Y. Sazeides and J. E. Smith. The predictability of data values. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 248–258, 1997.

[13] Avinash Sodani and Gurindar S. Sohi. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31th Annual International Symposium on Microarchitecture*, pages 205–215, Dallas, TX, 1998.