Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-Scale Object-Oriented Applications

Guoqing Xu
Ohio State University
xug@cse.ohio-state.edu

Nick Mitchell
IBM T. J. Watson Research
nickm@us.ibm.com

Matthew Arnold
IBM T. J. Watson Research
marnold@us.ibm.com

Atanas Rountev
Ohio State University
rountev@cse.ohio-state.edu

Gary Sevitsky
IBM T. J. Watson Research
sevitsky@us.ibm.com

ABSTRACT

Generally believed to be a problem belonging to the compiler and architecture communities, performance optimization has rarely gained attention in mainstream software engineering research. However, due to the proliferation of large-scale object-oriented software designed to solve increasingly complex problems, performance issues stand out, preventing applications from meeting their performance requirements. Many such issues result from design principles adopted widely in the software research community, such as the idea of software reuse and design patterns. We argue that, in the modern era when Moore's dividend becomes less obvious, performance optimization is more of a software engineering problem than ever and should receive much more attention in the future. We explain why this is the case, review what has been achieved in software bloat analysis, present challenges, and provide a road map for future work.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.2.8 [**Software Engineering**]: Metrics—*Performance measures*; D.3.4 [**Programming Languages**]: Processors—*Memory management, optimization*

General Terms

Performance

Keywords

Runtime bloat, performance analysis, performance optimization

1. INTRODUCTION

Over the course of 17 years from 1986 to 2002, the performance of microprocessors improved at the rate of 52% per year. These astounding technology advances provided "free lunch" to the software community, which, for most of the time, enjoyed the performance improvement by simply upgrading the hardware on which an application runs. Software performance scales well with Moore's law, providing an opportunity for the software engineering community to focus on high-level properties of a program, such as reliability, reusability, ease of understanding, etc., assuming performance has been appropriately taken care of by the underlying runtime system and architecture.

What grows even faster than the clock speed is the software functionality and size (a.k.a., Myhrvold's Laws). For example, the number of lines of code for the Windows Vista operating system in January 2008 is almost 150 times as large as that for Windows 3.1 in January 1992, and this growing speed is about 15 times greater than that for the number of transistors per chip unit area predicted by Moore's law [18]. Myhrvold's premise that "software is a gas" describes precisely the phenomenon that no matter how much improvement has been achieved on hardware, developers always have the tendency to add functionality to make their software push the performance boundaries. There is an ever-increasing demand for performance optimization in modern software despite the employment of faster CPUs and larger memory systems.

Runtime bloat In many cases, performance problems in a large application arise not from the lack of sufficiently fast hardware, but instead from the *runtime bloat* from which it suffers. The term bloat refers to a general situation where redundancy exists toward finishing a task, which could have been achieved more efficiently. Two types of runtime bloat are often seen in modern object-oriented applications: *memory bloat* that refers to space inefficiencies and *execution bloat* that occurs due to the execution of unnecessary operations. A typical example of memory bloat in Java applications is a memory leak, which is caused by unnecessarily holding references that are no longer used. While a memory leak does not affect the execution of a program, the accumulation of unused objects can quickly exhaust the heap space and crash the program. Examples of execution bloat are inappropriate design/algorithm choices: running an $O(n^2)$ bubble sort algorithm on a large database of items for which a faster quicksort algorithm may be more suitable, or using a SOAP protocol to transmit simple format data that can be sent directly from a client to a server[1].

In many cases, memory bloat and execution bloat exist simul-

FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.

[1]According to [19], the conversion of a single date field from a SOAP data source to a Java object can require as many as 268 method calls and the generation of 70 objects.

taneously to cause a performance problem. As an example, using a `java.util.HashSet` that always contains a single element involves both types of bloat: the internal implementation of `HashSet` creates and maintains a `HashMap` and delegates all the work to this `HashMap`. The big cost of creating and maintaining the data structure cannot be amortized if it contains a small number of elements. In this case, memory bloat exists as storing this single object has large space overhead, while execution bloat manifests when time is wasted on creating and maintaining the data structure that is not necessary for the forward progress.

Chronic runtime bloat can have significant impact on application running time, throughput, and scalability. Server instances that suffer from bloat can miss their scability goals by orders of magnitude (e.g., a server application designed to support millions of users could only accommodate a few hundreds due to inefficiencies [21]). As another example, by avoiding the generation of strings that are used only for debugging purposes, we have managed to reduce the running time of DaCapo/bloat [3] by 35% [38]. While a few redundant objects and operations may seem insignificant, their effects can quickly get exacerbated due to nesting and layering. This is especially the case in large-scale object-oriented applications that are built on top of a sea of abstractions and libraries.

Has it been solved by multicore? Over the past few years, the effect of Moore's law has transitioned from the growth of single-processor performance to the increase of the number of independent processors on a chip, leading to the blooming of multicore computers, parallel algorithms, and user-level tool support that help programmers exploit the parallelism of the underlying system [18, 2]. While performance improvement can usually be seen by parallelizing sequential programs, for large-scale enterprise applications that are already multi-threaded, the problem of excessive bloat will become increasingly painful as the number of cores grows, because memory bandwidth per core goes down and clock speed could not ameliorate ever-increasing levels of inefficiency. By fixing general memory and execution inefficiencies, IBM researchers have managed to increase the throughput of a large document management server more than twice as much as an alternative that tackles the problem by improving both the quality and quantity of the hardware. While this case study was designed to investigate multicore scalability, it concludes that general inefficiency is the most important performance bottleneck in large-scale enterprise-level Java applications [1].

Why a software engineering problem? Approaches for tackling the bloat problem can cross almost all stages of development, and can thus involve researchers from different SE fields. Our guiding principles in the community encourage certain forms of excess. Programmers are taught to pay more attention to abstractions and patterns to favor reuse and code readability, leaving performance to the compiler and runtime system, such as the Just-In-Time (JIT) compiler and garbage collector (GC). However, when the abstractions become deeply layered and nested, the optimizer can no longer clean up the runtime mess [21, 38]. As a community, we may need to refine our long-held principles to take into account performance. For example, we may teach programmers to be more considerate during the creation of APIs. While a general interface is important for reuse, some specialized versions may also be needed for clients that only have simple requests which can be processed with higher efficiency.

It is hard to have definitive evidences to decide whether an application has runtime bloat, because the definition of bloat is relative and based entirely on the comparison between the current implementation and a more optimal one, if that exists. This nature of bloat can also compromise the effectiveness of any automatic optimization technique provided by even the state-of-art compilers and architectures, as the decision is rather subjective and requires much involvement of human insight including both domain knowledge and programming experience. Finding such inefficiencies can be naturally thought of as a testing problem, where special test strategies (e.g., worst-case complexity testing [7]) may need to be created to evaluate how efficient a specific implementation is. It is also possible to employ sophisticated program analysis techniques, such as those developed in the SPEED project [11, 15, 14, 12, 13], to compute symbolic resource bounds for program components, which can provide useful insights into how they perform as a function of their inputs and can produce early warnings about potential performance problems.

A natural step after bloat is found is to remove it. Program analyses may play a key role in this stage of the work. For instance, dynamic analyses and profiling techniques could be used to identify frequently-occurring bloat patterns, and next, semantics-preserving static analyses could be designed to transform a bloated program into a bloat-free one, based on such patterns.

In a long-term research plan, bloat analysis can become a critical task in dealing with programs that have resource constraints as such programs will be the mainstream of the next-generation mobile computing. We envision a chain of new methods across the entire software life cycle that are designed to alleviate bloat, such as new language features, type systems, performance requirement specifications, and testing and analysis techniques. This chain of methods should take a performance-centric view, and can have conspicuous impact on the mobile software industry.

Our work We have already started research projects on software bloat analysis [19, 24, 40, 38, 22, 39, 41]. Over the past few years, we have analyzed dozens of real-world Java applications, found many severe bloat cases, and achieved considerably large performance improvements after removing the detected bloat. We realized that this work only scratches the surface and more collaborative research should be conducted to tackle this problem. Significant research opportunities are possible, if we, as a community, understand the problem, and start paying attention to it.

2. THE STATE OF THE ART

This section briefly discusses some existing work on bloat analysis, and presents a set of research challenges that we face.

Memory leak detection in managed languages As a type of memory bloat, a memory leak in a managed language occurs when object references that are no longer needed are unnecessarily maintained. Static analyses can be used to attempt the detection of such leaks. However, this detection is limited by the lack of scalable and precise reference/heap modeling (a well-known deficiency of static analyses), reflection, multiple threads, scalability for large programs, etc. Thus, in practice, identification of memory leaks is more often attempted with dynamic analyses [23, 4, 16, 17, 40, 5, 36, 6, 25].

Most existing work follows a "from-symptom-to-cause" diagnosis approach that looks for suspicious objects and then attempts to locate the leak cause from these detected objects. The detections of both suspicious objects and root cause are heuristics-based. For example, work from [23, 17] uses growing numbers of instances of certain types as an indicator of leaking objects while work from [16, 4, 25] identifies stale objects that are not used for a while. Next, the root cause of a memory leak is located by traversing backward the object graph from suspicious objects. However, the major limitation of a heuristics-based approach is the imprecision of the heuristics in the presence of a great number of objects and extremely complex referencing relationships. For example, neither growing

instances nor staleness can precisely capture leaking objects, and it is also difficult for the graph traversal algorithm to locate exactly the cause of the leak.

To overcome this problem, the work from [40] takes a container-centric view and detects leaks by modeling and profiling container behaviors. This approach is based on an important observation that many memory leaks in Java are caused by misuse of containers. The limitation of this work is that leaks that are not container-induced will be missed. Future work may extend this approach to handle more general memory leak cases.

Research related to Java memory leaks also includes the development of run-time techniques that can *survive* memory leaks by either swapping stale objects to disk [5, 36], or simply reclaiming objects that are highly unlikely to be used again [6]. These techniques can maintain stable performance and survive failures for long-running applications, thus extending executions in the presence of memory leaks.

Memory bloat detection The work from [24] introduces *health signatures* to evaluate if an application has achieved a good balance between necessary memory consumption and data structure memory overhead. A focus of this work is Java collection objects. For example, this analysis can show that a HashMap of 2-character Strings devotes 29% of its space to pointer overhead, and a HashSet object can have even higher overhead. Aggregating health measurements for individual collections can reveal memory bloat for an entire application. The report shows a real application that devotes 74% of its memory to collection fixed and per-element costs, a percentage surprising enough to motivate us to take actions to avoid bloat.

Execution bloat detection Similarly to heuristics-based memory leak detection techniques, most of the work that detects execution bloat is based on side effects (i.e., suspicious behaviors) that bloat exhibits. The categories of side effects that have been considered include large volumes of temporary (short-lived) objects, pure heap value copies, and inappropriate container behaviors.

(1) *Temporary objects*. Dufour *et al.* [9] propose a blended problem analysis technique, which applies static analysis to a region of dynamically collected calling structure with observed performance problem. By approximating object effective lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic program region. Shankar *et al.* develop a JVM-based tool called *Jolt* [27] that identifies regions that make heavy use of temporary objects. The tool forces the JIT compiler in the JVM to perform aggressive method inlining in such regions so that the JIT may find more optimization opportunities (e.g., stack allocating more temporary objects).

(2) *Pure copies*. A long sequence of pure heap copies without computations is very likely to indicate redundant operations. A typical example is the existence of multiple representations of the same data, which keeps being wrapped and unwrapped among components that use these different representations. Work from [38] proposes a copy profiling technique that detects bloat by identifying copy-heavy regions. It has been shown to be helpful in pinpointing heap data structures that have their stored data directly copied from other data structures. The unused string problem in DaCapo/bloat described earlier was found using this approach.

(3) *Container bloat*. An important source of bloat is the inappropriate use of containers. Different container types and implementations are designed for different usage scenarios, so choosing containers without understanding their costs and benefits could lead to significant memory and execution bloat. For example, HashSet is suitable for storing a large number of elements and providing quick membership test, and should not be used if the number of elements is relatively small. Work from [26] proposes a dynamic technique that profiles programs to make container choice recommendations. This set of recommendatations can be applied automatically by transforming the program or presented to the user for diagnosis purposes.

Recent work from [41] identifies two specific types of container inefficiencies, namely, underutilized container and overpopulated container. This work proposes the first static analysis to identify bloat: this analysis automatically extracts container semantics and does not require user annotations. Container functionality is abstracted into two basic operations *ADD* and *GET*, and the analysis detects stores and loads that concretize them based on the context-free-language reachability formulation of points-to analysis [34, 42]. The second step of the analysis is to approximate the frequencies of the identified stores/loads to/from heap locations based on the nesting relationships among the loops where they are located: if the number of *ADD* operations is very small, the container is underutilized; if the number of *ADD*s is significantly larger than the number of *#GET*s, it is overpopulated because many elements are not retrieved at all.

Different from these side-effect-based bloat detection techniques, work from [39] proposes a dynamic cost and benefit analysis that detects bloat by capturing data structures that are expensive to construct but have low benefit for the forward execution. Such high-cost-low-benefit data structures are common in many types of inefficiencies despite the different observable symptoms they may exhibit. The cost of a heap value is considered as the number of instructions executed to produce this value from other existing heap values; The benefit of a heap value is considered as the number of instructions executed to transform this value to other heap values. By ranking data structures based on their cost-benefit rates, the tool has been shown to be helpful in quickly understanding the cause of the bloat.

Bloat removal It could be extremely costly to construct and initialize data structures that are invariant across loop iterations. We are currently working on a static analysis that attempts to hoist such data structures out of loops. The analysis works on two dimensions: it first identifies a loop-invariant logical data structure, and next checks whether it can be hoisted by attempting to hoist statements that access it. For data structures that are not safe to hoist, it computes hoistability measurements that are presented to the user for further inspections.

Despite all advances made by existing work, a number of significant challenges remain. Here we describe three most significant challenges that we have faced in our work.

Challenge 1: improving dynamic analysis precision In most cases, dynamic analysis is the "weapon of choice" for bloat detection. The most significant limitation of such postmortem dynamic bloat detectors is that the generated reports usually have a large number of false positives, pointing to program entities that are all "suspicious" according to certain rules used by the tool but are not really problematic. The problem with the memory leak detectors described earlier is a typical example. We found such a problem exists for almost all dynamic detectors that we have developed, primarily because it is extremely hard to define selection rules that can describe *unique characteristics* for the problematic program entities. A possible solution (adopted by container profiling [40]) is to reduce our expectation from detecting a general class of problems to a very specific class, for which a more precise symptom definition is possible and is more closely related to the root cause. This small class of problems has to appear frequently so that the sacrifice of ignoring other problems (not in this class) can be justified. It remains to be seen, however, how such a solution can be

applied to a more general setting of bloat detection, for example, to distinguish copies that are useful and those that are not.

Challenge 2: static vs. dynamic analysis The container inefficiency detection work from [41] uses static analysis to reduce dynamic analysis false positive rates, because the static analysis can exploit source code properties that represent programmers' intentions or mistakes that are inherent in the program. This comparison between static and dynamic analysis used in bloat detection is quite interesting and is completely opposite to the traditional belief that dynamic analysis is more precise, for example, for testing and debugging. This difference results from the two different uses of the analysis, namely for inferring and for checking. Alternatively, dynamic analysis can be used to prune the code entities that are considered by a subsequent static analysis, as done in [8, 9]. It remains to be investigated if there are other types of bloat whose detection can take advantage of a static analysis or a hybrid technique.

Challenge 3: specification-based bloat detection It is clear that the primary reason why the state-of-the-art tools are not precise enough is the lack of precise specifications that define what bloat is. The problem of bloat detection could be made much easier if there existed (non-functional) specifications that a programmer can use during development. Such specifications may bridge the gap between the non-functional performance analysis and the huge body of existing work on checking and testing functional properties of programs, thus making it possible to completely automate the performance tuning process.

3. FUTURE DIRECTIONS

In this section, we describe future research opportunities, with a focus on what the SE community can do to address the ever-increasing levels of inefficiency in object-oriented applications.

Thin patterns While design patterns [10] have been extremely successful in helping programmers write highly-manageable code, they are the root causes of many types of runtime bloat. In many large applications, for instance, in order to implement a visitor pattern, the programmer uses inner classes to represent different kinds of visitors, which contain nothing but a `visit` method. Such a visitor class can be instantiated an extremely large number of times (e.g., the allocation sites are located in loops with many layers of nesting), and all objects created are identical: they have no data and are used only for dynamic dispatch. It is *not* free to create and deallocate these objects, and significant overhead reduction can be seen when we use only the method without creating objects.

Future research on patterns may consider the creation of a few specialized versions for each existing pattern (i.e., *thin patterns*), which provides different tradeoffs between inefficiency and modularity. On the compiler side, pattern-aware optimization techniques could be expected to remove inefficiencies and generate higher-quality code.

Performance-conscious modeling languages and tools While performance-aware design has been extensively studied in the field of software performance engineering (e.g., [28, 37]), this research focuses primarily on high-level architectures and processes, rather than low-level program inefficiencies. Hence, the problem is worth re-considering in the future, and additional efforts should be focused on explicit bloat avoidance in the state-of-art modeling languages (e.g., UML) and tools (e.g., EMF and Rational Software Modeler).

Careless design can lead to significant runtime bloat, especially when modeling tools are used to generate code skeletons automatically from the design. As an example, we found that one cause of bloat are carelessly chosen associations. Consider several classes X_1,X_2, ... together with the associations between them (e.g., as defined in object-oriented design and captured in UML class diagrams). The associations typically include directionality (uni- vs. bi-directional) and multiplicity (e.g., one-to-many). There are often many semantically-equivalent ways to implement them in the code. The programmer may choose one of these possibilities without truly understanding the implications of her/his choice on the memory footprint of the application. Even worse, in many cases, the programmer does not make this choice at all— the default data model defined in the modeling tool is applied automatically behind the scenes. A performance-conscious design model will take performance requirements as an explicit parameter, and this will result in extended modeling languages and tools that incorporate various resource constraints.

Unit testing/checking bloat It is important to avoid inefficiencies early during development before they accumulate and become observable. This calls for novel program analysis and testing techniques that can work for incomplete programs. While there exists a body of work on unit testing and component-level analysis, it is unclear how to adapt them to verify non-functional properties. For example, it may not be easy to write assertions (i.e., test oracles) for unit testing, as redundancy at the unit level may not be obvious and thus the assertions are likely to be insensitive to explicit performance checks (e.g., running time and space).

This difficulty actually points to the more general non-functional specification problem. What can we assert about performance other than running time and space? Can any functional properties of a program be employed to specify performance requirements? Good specifications must be closely related to a certain bloat problem, and not simply describe the symptom that the problem exhibits. Significant improvements could be achieved in the research of performance analysis if such specifications were designed and evaluated.

Autonomous system and program synthesis Looking a bit far into the future, the feedback-directed compilation techniques in a JVM may be powerful enough to unpile the "big pileup" [20] during the execution. For example, dynamic object inlining may be a effective approach to reduce pointer overhead. In order to remove container inefficiency, the runtime system could automatically shrink the space allocated for the container if it observes that much of the space is not used. These technique of course require sophisticated profiling techniques that are semantics-aware and incur sufficiently low overhead. Recent advances in program synthesis [32, 29, 33, 30, 31, 35] shed a new light on solving the execution bloat problem. Given a user-defined specification, a program synthesis tool can automatically choose from a space of algorithms the most efficient one. This can apply naturally in the research of bloat detection to find efficient implementations, and may further be used to generate implementations for performance-critical tasks that are guaranteed to meet performance requirements.

4. CONCLUSIONS

In this position paper, we describe software bloat, an emerging problem that has increasingly negative impact on large-scale object-oriented applications. We argue that it is essentially a software engineering problem, and it is time for the SE community to start contributing new solutions for it. We survey some of the existing work on bloat analysis, describe challenges, and outline some promising future directions. We believe there are larger opportunities than ever before for the SE community to make software more efficient, and this can happen entirely at the application level, without the help of ever-increasing hardware capabilities.

5. REFERENCES

[1] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *OOPSLA*, 2010.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, 2009.

[3] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, pages 169–190, 2006.

[4] M. D. Bond and K. S. McKinley. Bell: Bit-encoding online memory leak detection. In *ASPLOS*, pages 61–72, 2006.

[5] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In *OOPSLA*, pages 109–126, 2008.

[6] M. D. Bond and K. S. McKinley. Leak pruning. In *ASPLOS*, pages 277–288, 2009.

[7] J. Burnim, S. Juvekar, and K. Sen. WISE: Automated test generation for worst-case complexity. In *ICSE*, pages 463–473, 2009.

[8] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA*, pages 118–128, 2007.

[9] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *FSE*, pages 59–70, 2008.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] B. Gulavani and S. Gulwani. A numerical abstract domain based on "Expression Abstraction" and "Max Operator" with application in timing analysis. In *CAV*, pages 370–384, 2008.

[12] S. Gulwani. Speed: Symbolic complexity bound analysis. In *CAV*, pages 51–62, 2009.

[13] S. Gulwani. The reachability bound problem. In *PLDI*, pages 292–304, 2010.

[14] S. Gulwani, S. Jain, and E. Koskinen. Control-flow refinement and progress invariants for bound analysis. In *PLDI*, pages 375–385, 2009.

[15] S. Gulwani, K. Mehra, and T. Chilimbi. SPEED: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, 2009.

[16] M. Hauswirth and T. M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *ASPLOS*, pages 156–164, 2004.

[17] M. Jump and K. S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *POPL*, pages 31–38, 2007.

[18] J. Larus. Spending Moore's dividend. *Commun. ACM*, 52(5):62–69, 2009.

[19] N. Mitchell. The runtime structure of object ownership. In *ECOOP*, pages 74–98, 2006.

[20] N. Mitchell. The big pileup. In *ISPASS*, pages 1–1, 2010.

[21] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[22] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *ECOOP*, pages 77–97, 2009.

[23] N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *ECOOP*, pages 151–172, 2003.

[24] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *OOPSLA*, pages 245–260, 2007.

[25] G. Novark, E. D. Berger, and B. G. Zorn. Efficiently and precisely locating memory leaks and bloat. In *PLDI*, pages 397–407, 2009.

[26] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *PLDI*, pages 408–418, 2009.

[27] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *OOPSLA*, pages 127–142, 2008.

[28] C. Smith and L. Williams. Software performance engineering: A case study including performance comparison with design alternatives. *IEEE Trans. Soft. Eng.*, 19:720–741, 1993.

[29] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[30] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[31] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching concurrent datastructures. In *PLDI*, pages 136–148, 2008.

[32] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, pages 281–294, 2005.

[33] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.

[34] M. Sridharan and R. Bodik. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, 2006.

[35] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[36] Y. Tang, Q. Gao, and F. Qin. Leaksurvivor: Towards safely tolerating memory leaks for garbage-collected languages. pages 307–320, 2008.

[37] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *FOSE*, pages 171–187, 2007.

[38] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *PLDI*, pages 419–430, 2009.

[39] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *PLDI*, pages 174–186, 2010.

[40] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *ICSE*, pages 151–160, 2008.

[41] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *PLDI*, pages 160–173, 2010.

[42] G. Xu, A. Rountev, and M. Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *ECOOP*, 2009.