FILIP PIZLO

Research Statement

pizlo@mac.com • filpizlo.com • 7744 Preservation Dr. • Indianapolis, IN 46278 • 317.966.9592

My research focuses on enabling the use of high-level computer programming languages in deeply embedded, power constrained, and hard real-time environments. In my early work, I explored a number of programming model extensions to Java [21, 20, 19, 14, 5] that sought to improve performance and predictability. More recently, I have invented improvements to the runtime systems underlying high-level languages [15, 11, 16, 13, 12, 9, 8, 3, 6, 2, 1], which enable high-level programs to achieve performance and predictabiltiy that was previously only possible with low-level languages such as C or assembly. The ultimate aim of my research is to eliminate the need for low-level programming languages altogether. This has broad implications for software engineering, security, and computer science education. Some of the most often encountered software bugs – such as buffer overruns and uninitialized state – can be completely eliminated through the use of type-safe languages like Java. My work demonstrates that such languages can be used even in low-level tasks, including operating systems [6, 2] and hard real-time applications [2, 1, 18, 17]. Notable achievements of this research include the first open-source real-time Java virtual machine [18, 17], the first open-source real-time garbage collector [16, 3], the first successful Java flight test as part of the DARPA/PCES program in conjuction with Boeing [18, 17], the first high-performance concurrent copying garbage collector [12], the first wait-free fragmentation tolerant garbage collector for multiprocessors [1], as well as most recently, the world's fastest real-time Java virtual machine that has demonstrated performance and predictability on par with kernel code written in C even when running on a small, highly resource-constrained and radiation-hardened SPARC architecture [2, 1].

I'm also involved in interdisciplinary work in very-high-energy astrophysics as part of the VERITAS [22] collaboration. I am the designer of the main data format including its novel real-time compression scheme, the Quicklook real-time data analysis, and the back-end data acquisition system responsible for handling the 20 MB/sec data stream. I continue to be a member of VERITAS, performing a range of duties including shifts as the nightly telescope operator, routine data analysis, and analysis optimization.

1 Programming Models

Slight changes to existing programming languages, as well as slight changes in programming style, can enable increased performance and reliability. As part of the broader OpenVM effort, I was one of the first to investigate the usability of the RTSJ programming model. The Real Time Specification for Java (RTSJ) was the earliest attempt to extend Java to include real-time support. Because it was introduced prior to the development of high-performance real-time garbage collectors, the RTSJ was based around a type-safe memory management discipline that side-stepped garbage collection while guaranteeing bounds on allocation times. This discipline, known as scoped memory, is based on notions of region-based memory management that have long been used in C and C++ programs. I was one of the first to investigate the usability of RTSJ scoped memory in real programs. This work led to the development of the first benchmark for real-time Java [7], the first design patters for scoped memory [21], an extension to scoped memory that allows for first-in, first-out communication patterns [19], and a number of empiric as well as qualitative comparisons of scoped memory versus real-time garbage collection [16, 10]. I also worked to unify the notions of garbage collected heap memory and scoped memory in my work on Hierarchical Real-time Garbage collection, which showed that slight changes to programming style can lead to large improvements in mutator utilization and predictability [13]. I developed the OpenVM runtime-side support for Reflexes, an extension to Java based on confined types that allows for garbage collection and scoped memory to be side-stepped altogether [14, 5]. Most recently, I developed the dynamic and static analyses that led to the development of a type-system and programming model for thread locality in Java [4].

2   High-assurance Managed Runtime Systems

Languages such as Java and C# typically require a large runtime system which leads to unpredictable performance, large memory overheads, and a lack of portability. These languages are rarely considered as serious candidates for high-assurance, high-performance, high-predictability operating system and application development. My research aims to reverse this trend by developing managed runtime systems that exhibit performance, predictability, and portability that meets or exceeds that of C or C++ code.

2.1   Java on a Plane with OpenVM

As part of the broader OpenVM effort, of which I am a core developer, I developed novel I/O scheduling [17], synchronization [20], compilation [15, 11], and memory management [19, 16, 13, 3] techniques. These techniques were instrumental to the first real-time Java flight test, as part of the DARPA/PCES program [18, 17]. My work on the OpenVM Minuteman real-time garbage collector led to the first (and so far only) open-source implementation of the famous Metronome algorithm, which is used in IBM's WebSphere SRT and WebSphere WRT products. Minuteman has been used to validate OpenVM's accurate stack scanning infrastructure [15], perform the first RTGC-versus-scopes comparison [16], and perform the first direct empirical comparison of the two leading RTGC styles [3].

2.2   Concurrent Lock-free Defragmentation

While an intern at Microsoft Research, I continued my work on real-time garbage collection, this time focusing on the Microsoft Bartok compiler and runtime for .NET and multiprocessor systems. Real-time and concurrent garbage collectors are reasonably well understood if the fragmentation problem is ignored. The typical technique for dealing with fragmentation – moving objects to create a contiguous unfragmented free region of memory – becomes difficult if the garbage collector is running concurrently to the program on a separate processor. Previous techniques either sacrifice object access coherence leading to the risk of the program observing "time travel" of values, or sometimes require pausing program threads while the collector completes copying. To solve this problem I implemented Stopless, the world's first lock-free concurrent copying garbage collector [12]. Because Stopless is able to copy objects concurrently to program execution, it is able to create contiguous free regions of memory without requiring programs to pause. I then invented two improved concurrent copying collectors: Clover, which achieves improved throughput and predictability through the use of random numbers, and Chicken, which is based on opportunistic guarantees [9]. Clover was the first, and is so far the only, probabilistic garbage collector. To make all three collectors fast, I invented a compiler optimization known as Path Specialization [8] that can be used to improve the performance of every known real-time garbage collector. Path Specialization exploits the observation that garbage collectors cause programs to uncur overheads, but only during some phases of the collector's execution. I modified the Bartok optimizing compiler to leverage this observation and produce specialized versions of code for different compiler phases. Through the use of aggressive flow analyses and dead code elimination, this optimization leads to minimal code bloat while improving the performance of every RTGC style that I tested.

2.3   The Fiji VM Experience

I decided to develop my own Java virtual machine from scratch [6]. Having a complete compiler, static analysis, and runtime infrastructure that I control is instrumental to my current research. This clean-room effort, dubbed Fiji VM, is intended to create a system that can be used as an ideal test-bed for runtime extensions that make heavy use of complicated barrier insertion, such as real-time garbage collectors, software transactional memory systems, profilers, and debuggers. To ensure maximum performance, I designed the Fiji VM compiler around state-of-the-art techniques such as 0CFA (monovariant control-flow analysis) and SSA (static single assignment form). To my knowledge, Fiji VM is the first production system to use whole-program static analyses such as 0CFA to drive system-wide optimization. Additional optimizations include global value numbering, sparse conditional constant propagation, loop unrolling, and loop peeling, among many others. I also leveraged my prior work on compilation strategies for accurate garbage collection [15, 11] to allow high-performance stack-scanning with minimal mutator overheads. The initial production garbage collector, dubbed CMR (concurrent mark-region), is a concurrent and real-time variant of the Immix mark-region garbage collector found in Jikes RVM. Thanks to these optimizations, Fiji VM achieves throughput that exceeds that of Jikes RVM, performs on par with Oracle HotSpot, yet

is able to run in platforms as small as a radiation-hardened SPARCv7 ERC32 with 4MB of memory. Fiji VM led both to further research as well as a commercialization opportunity (two SBIR grants from the National Science Foundation and the Air Force). The first research outcome of Fiji VM was the static analysis behind the Loci thread-locality programming model [4]. The Fiji VM compiler transforms Java bytecode into a highly ergonomic SSA-based internal representation that is ideal for rapidly developing novel program analyses. In the Loci effort, I used this infrastructure to statically detect Java classes that do not cause their own instances to escape, thus making them candidates for the Loci `@Thread` annotation. Fiji VM's biggest accomplishment to date is that it allows pure garbage-collected Java to be used for writing operating system kernel code while achieving exactly the same predictability characteristics as C [6, 2, 1]. I accomplished this by aggressively optimizing away all of Java's overheads via null-check elimination, array bounds check elimination, loop unrolling to reduce safepoint overhead, and aggressive type refinement to eliminate type checks. Additionally, Fiji VM's runtime support only requires threads and locks – I make no use of page protection schemes or any sophisticated OS services. This is instrumental in allowing the VM to be used for the development of kernel code and real-time applications: kernels will internally provide threads and locks but little else.

2.4   Schism: a simpler solution to concurrent defragmentation

My initial work on Fiji VM proved that high-level languages can perform low-level tasks. But one of the biggest difficulties in the development of low-level and hard real-time software is not the language but rather the memory management. Even in low-level languages, memory management is complicated due to the fear of fragmentation. Because of fragmentation, the use of dynamic memory allocators is generally disallowed in safety-critical software. In previous work [12, 9], I had invented collectors that copy objects concurrently to program execution in an attempt to solve this problem. However, these collectors have some undesirable trade-offs: Stopless [12] can occasionally cause slow-downs in program execution, the gaurantees provided by Clover [9] are probabilistic in nature, while Chicken [9] is not guaranteed to always succeed in defragmenting the heap. All three algorithms are also quite complex and have sizeable through-put penalties, which jeapordizes their practicality. This led me to conclude that the problem of concurrent defragmentation was not yet solved. Therefore I returned to the concurrent defragmentation problem and designed a new real-time garbage collector, called Schism [1]. Schism eliminates the fragmentation problem for good while maintaining excellent bounds on execution times and space usage. This new collector is wait-free, pause-free, and unlike previous real-time collectors, completely fragmentation-tolerant; i.e. fragmentation is allowed to occur but the collector ensures that it never prevents programs from being able to allocate memory. Schism is fully concurrent, can leverage both multiprocessor and uniprocessor platforms, achieves better throughput and time-predictability than the best non-defragmenting collectors (including the IBM WebSphere implementation of the Metronome), and has proven bounds on both execution times and space usage. This is achieved by exploiting a previously unknown artifact of concurrent replicating semi-space garbage collectors and arraylets. Collectors such as Metronome use Arraylets to fight fragmentation by splitting objects into fixed-size fragments, with each object's fragments being indexed by a single contiguous spine. With fragment sizes of 32 bytes or more, the spine is an order of magnitude smaller than the array, thus reducing fragmentation. Schism takes this notion further and eliminates the problem of fragmentation completely, while ensuring that all object accesses require no more than two indirections. Schism exploits the fact that the spine itself is immutable, making it an ideal candidate for replication-based concurrent copying, which was previously thought to be only suitable for pure functional languages. The Schism algorithm segregates the heap into a fixed-size concurrent replication-copying semi-space for spines, and a fixed-size concurrent mark-sweep space for managing fixed-size object fragments. The mark-sweep space is impervious to external fragmentation due to the use of fixed object sizes, while the spine semi-space eliminates external fragmentation for spines by constantly copying them. Spine copying is wait-free and fully concurrent to application execution, which is possible because spines never change: their only content is pointers to non-moving object fragments. This allows for a wait-free real-time garbage collector with never-before seen time and space guarantees for languages in which object mutability is required, and where coherence of object accesses must be preserved.

To validate these claims, I assembled the largest set of benchmarks, platforms, baseline systems, and configurations to create the most thorough comparison of real-time garbage collection technologies to date [1]. On systems ranging from a 40MHz single-core LEON3 SPARC to an 8-way Intel Xeon machine, Schism soundly outperformed all other real-time garbage collectors (including IBM's and Oracle's production RTGCs) while completely eliminating external fragmentation. In this work I also produced a

comprehensive proof of internal fragmentation bounds. A notable aspect of this performance evaluation is that it was the first, and is so far the only, garbage collector evaluation in which a worst-case simulator is included as a configuration in every benchmark. The simulator gives an immediate sense of how Schism would perform if all collector heuristics failed. The comparison in [1] shows that even in the worst case, Schism's performance is only 20% worse than its average case, with the average case being superior to any other real-time collector.

2.5 Jikes RVM

In addition to my work on real-time Java runtime systems, I also continue to be involved with broader Java efforts. As a member of the Jikes RVM team, I was responsible for the overhaul of the threading subsystem and for the introduction of a novel biased locking scheme which led to the highest ever recorded performance improvement across all available benchmarks.

3 Interdisciplinary Work

While computer science research is the core of what I do, I have also been involved in very-high-energy gamma-ray astrophysics for the past eight years as part of the VERITAS [22] collaboration. VERITAS is the world's most sensitive gamma-ray observatory in the very-high-energy spetrum. I am the designer and maintainer of the back-end data acquisition system, the real-time Quicklook data analysis, and the main data format. In addition to software implementation, I was responsible for designing a compression scheme that is optimized for the probability distribution of night-sky background as well as cosmic ray signals and optimizing the data analysis to maximize signal-to-noise ratios. I continue to be involved in VERITAS, and look forward to using my combined experience in physics and computer science to perform interdisciplinary research. The latest physics experiments increasingly require more computing power than is available from individual processing cores – and yet most of the software engineering methodologies used in physics have yet to adapt to concurrent and parallel systems. Having a deep understanding of the scientific questions being asked in astrophysics gives me a unique view of how the design and implementation of programming languages should be pushed forward to cope with parallelism.

4 Current and Future Work

In my current work I am continuing to bridge the gap between high-level programming and embedded real-time systems, as well as finding ways in which the techniques I have developed can be used to improve non-real-time metrics such as throughput. To this end, I have been focusing on concurrency, compilation and execution strategies, as well as memory management:

High-throughput Concurrent Garbage Collection. Traditional approaches to high-throughput garbage collection require stopping the world. But the very act of stopping the world does not scale – it represents serial work, and implies that managed languages will hit Amdahl's wall sooner than unmanaged ones. Concurrent garbage collection allows this impediment to be side-stepped, provided that concurrent GCs can be optimized for throughput instead of just responsiveness. Fiji VM already has a concurrent collector (the Fiji concurrent mark-region, or CMR) that outperforms some of the best stop-the-world throughput garbage collectors. I am currently working on extending this framework to further exploit the throughput benefits of Fiji CMR.

Sleazy Threading. Styles of threading used for concurrency and parallelism present a trade-off. Using operating system threading facilities maximizes throughput if the number of runnable threads is small, but deteriorates for large numbers of threads. Implementing threading at the user level maximizes throughput for large numbers of threads, but displays poor performance if threads block on operating system resources. What if this trade-off could be eliminated? I am currently working on a new threading model for Java called Sleazy Threads, where each thread can automatically transform itself from user-level to native based on its workload. The goal of this work is to enable Java programmers (as well as programmers of any language that targets the Java virtual machine) to spawn as many threads as they want, without fear of hitting operating system thread limits or losing throughput due to user-level scheduling overhead.

Garbage Collection as Constraint Solving. Garbage collection can be viewed as solving boolean constraints where the only operator is "OR": if object A or object B refer to object C, then object C is live. What if the language allowed for the construction of "AND" constraints as well? This could serve as a generalization of both ephemerons and weak references – both could be implemented at the language level instead of requiring cumbersome libraries. I am currently exploring ways of adding this notion to the Fiji VM, and demonstrating its usefulness for solving well-known memory leaks arising from the use of Java thread locals and weak hash-maps.

High-performance Real Time Locking. Concurrency is at the center of modern real-time programming models. Locks, in turn, are at the center of those models. Traditional approaches to locking in real-time systems require direct operating system support – thus different operating systems provide radically different semantics, and performance generally suffers due to frequent use of system calls. Fiji VM already has locks that are faster than what traditional real-time APIs provide [2]. I am continuing to explore ways of accelerating lock performance, thus reducing the overheads associated with lock acquisition, lock contention, and preemption.

Concurrent On-the-fly Class Loading and Optimization. State-of-the-art techniques for class loading in a Java virtual machine require either giving up on the constant-time bytecode execution guarantees that Fiji VM and Ovm currently enjoy [2, 17, 18], or else stopping the world to make changes to type metadata. I am currently designing and implementing an entirely new algorithm for class loading that will allow Fiji VM to guarantee (a) constant-time wait-free performance of type inclusion tests, (b) constant-time wait-free performance of interface dispatch, and (c) concurrent and on-the-fly class loading without having to stop any threads. This is accomplished by borrowing ideas from real-time concurrent on-the-fly copying garbage collection, and applying them to Java class loading, linking, and compilation algorithms. After the initial implementation is complete I intend to apply the same ideas to make adaptive JIT optimization fully on-the-fly and pause-free.

References

[1] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, J. Vitek. Schism: Fragmentation-Tolerant Real-Time Garbage Collection. *To appear in the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI 2010).*

[2] F. Pizlo, L. Ziarek, E. Blanton, P. Maj, J. Vitek. High-level Programming of Embedded Hard Real-Time Devices. *ACM SIGOPS 2010 EuroSys Conference.*

[3] T. Kalibera, F. Pizlo, A. L. Hosking, J. Vitek. Scheduling Hard Real-Time Garbage Collection. *IEEE Real-Time Systems Symposium (RTSS) 2009: 81-92.*

[4] T. Wrigstad, F. Pizlo, F. Meawad, L. Zhao, J. Vitek. Loci: Simple Thread-Locality for Java. *2009 European Conference on Object-Oriented Programming (ECOOP 2009).*

[5] J. Spring, F. Pizlo, J. Privat, R. Guerraoui, J. Vitek. Reflexes: Abstractions for Integrating Highly Responsive Tasks into Java Applications. *ACM Transactions in Embedded Computing Systems (TECS), 2009.*

[6] F. Pizlo, L. Ziarek, J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. *JTRES 2009: 110-119.*

[7] T. Kalibera, J. Hagelberg, F. Pizlo, A. Plsek, B. Titzer, J. Vitek. CDx: a family of real-time Java benchmarks. *JTRES 2009: 41-50.*

[8] F. Pizlo, E. Petrank, B. Steensgaard. Path Specialization: Reducing Phased Execution Overheads. *2008 International Symposium on Memory Management (ISMM 2008).*

[9] F. Pizlo, E. Petrank, B. Steensgaard. A Study of Concurrent Real-time Garbage Collectors. *ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 2008).*

[10] F. Pizlo, J. Vitek. Memory Management for Real-time Java: State of the Art. *IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC),* May 2008.

[11] J. Baker, A. Cunei, T. Kalibera, F. Pizlo, J. Vitek. Accurate Garbage Collection in Uncooperative Environments. *Concurrency and Computation: Practice and Experience,* 2008.

[12] F. Pizlo, D. Frampton, E. Petrank, B. Steensgaard. Stopless: A Real-Time Garbage Collector for Modern Platforms. *2007 International Symposium on Memory Management (ISMM 2007)*

[13] F. Pizlo, A. Hosking, J. Vitek. Hierarchical Real-time Garbage Collection. *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'07).*

[14] J. Spring, F. Pizlo, R. Guerraoui, J. Vitek. Reflexes: Programming Abstractions for Highly Responsive Systems. *2007 ACM International Conference on Virtual Execution Environments (VEE'07).*

[15] J. Baker, A. Cunei, F. Pizlo, J. Vitek. Accurate Garbage Collection in Uncooperative Environments with Lazy Pointer Stacks. *Compiler Construction, 16th International Conference, CC 2007.*

[16] F. Pizlo, J. Vitek. An Empirical Evaluation of Memory Management Alternatives for Real-time Java. *27th IEEE Real-Time Systems Symposium (RTSS 2006).*

[17] A. Armbuster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, J.Vitek. A Real-Time Java Virtual Machine with Applications in Avionics. *ACM Transactions in Embedded Computing Systems (TECS)*, 2006.

[18] J. Baker, A. Cunei, C. Flack, F. Pizlo, M. Prochazka, J. Vitek, A. Armbuster, E. Pla, D. Holmes. Real-time Java in Avionics Applications. *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006).*

[19] F. Pizlo. First in, first out memory. *Presented at the 3rd Workshop on Java Technologies for Real-time and Embedded Systems (JTRes 2005).*

[20] F. Pizlo, M. Prochazka, S. Jaggannathan, J. Vitek. Transactional lock-free data structure for Real Time Java. *Workshop on Concurrency and Synchronization in Java Programs 2004.*

[21] F. Pizlo, J. Fox, D. Holmes, J. Vitek. Real-time Java scoped memory: design patterns, semantics. *IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC'04).*

[22] VERITAS collaboration. http://veritas.sao.arizona.edu/