mization for inlining and the selective optimization in our prototype. Figure 1 illustrates the combined effect of these annotation-based optimizations on compilation delay. The left graph shows the resulting transfer delay with and without annotation and the right graph shows the impact of annotation on startup time. The benchmarks used for the results are Java applications from the SpecJVM [40] benchmark suite and those commonly used in similar studies.

In the left graph, we show compilation delay for optimized compilation (O3), annotation-guided optimization (Annot), and fast, unoptimized compilation (O1). Since the optimizations are profile-guided, we present realistic results: different inputs are used for profile and result generation. When the same input is used for both profile and result generation (equating to having perfect information about program behavior), an additional 0.1% improvement is achieved. On average, annotation-guided optimization is approximately 250 milliseconds slower than unoptimized (O1) compilation and 2 seconds faster than optimized compilation. In addition, the latter is achieved without affecting total time (execution plus compile time). On average, these annotation optimizations reduce compilation delay by almost 80% and enable speedups of 4% on average (these results, not shown here, can be found in [22]).

More importantly, annotation-guided compilation can substantially improve startup time as shown by the right graph in the figure. Our profile data indicates that almost all compilation in Java programs occurs at startup: (77%) of the compilation occurs in the first 10% (5 seconds) of program execution on average. Hence by reducing compilation delay, annotated execution should substantially reduce this startup cost. The right graph confirms this with graphs of the cumulative distribution of compile time over the average program lifetime (46s). The x-axis is the percentage of execution time (without compilation) of the program. The y-axis is the cumulative compilation time in seconds that must be incurred by the program during execution. Distributions are shown for optimized compilation (O3) and annotation-guided optimization (Annot). For all programs studied, the compilation delay required at startup completes more than 2 seconds earlier using annotation enabling substantially more progress to be made.

2.1.2 Annotation Security

Developers of existing execution environments have been apprehensive about the use of existing Java annotation frameworks since all annotations proposed prior to our work increase file size (which introduces additional transfer delay) and are insecure [16, 19, 36]. We address annotation size in this project by incorporating gzip compression of annotations and through the development of an efficient annotation language representation. In addition, we plan to consider other compression techniques that may prove more appropriate for encoding annotations.

The second issue that must be addressed if an annotation-based system is to be successful is security. Annotations must be verified or guaranteed that their use will not corrupt the execution environment or machine on which they are used. Use of most existing bytecode annotation systems poses serious security risks since the annotations implemented using these systems affect program semantics. If the bytecode stream is intercepted and modified by an untrusted party, illegal and possibly destructive program behavior can result. Up to this point, all of the annotations we propose are guaranteed to be secure: Manipulation of them can only affect program performance.

However, insecure annotation-based optimizations, e.g., register allocation, escape analysis, null check elimination, object placement etc., can potentially improve mobile program performance substantially [16, 19, 5]. We plan to develop mechanisms that enable the security of our annotation-based optimizations so that we are able to achieve additional execution performance gains. Improved overall mobile program performance (balancing transfer, verification, compilation, and execution delay) will continue to be our motivating goal. We believe that by considering these sources in unison we can achieve this goal and enable wide-spread use of our annotation-based execution environment.

We will consider three security mechanisms with which we can extend our annotation framework: Language construction, cryptographic protocols, and proof verification. Prior to our work, such mechanisms have been proposed to enable safe transfer of entire programs. However, each introduces mobile program execution overhead. Our approach is to use the mechanism that proves to be the most efficient for annotations alone; we will rely on other mechanisms for safe transmission of code and data.

Program representations (transfer formats) have been proposed that are safe by construction, e.g., SafeTSA [2]: Any program in such a format can not be created if type-safety is violated. We plan to explore a similar methodology in which construction of our annotations guarantees that the necessary security checks are made (and only redundant ones have been removed). The encoding will also guarantee that the annotations were generated from the file in which they are contained. To enable this, we will consider encoding program information into the annotations, as well as integrating our annotation language into the transfer format itself, e.g., by defining additional bytecode instructions.

Alternately, cryptographic protocols can be used, e.g., we can tag a file at the source to indicate to the recipient that the file originates from the entity that claims to have sent it and has not been tampered with en route. We will consider dual-key (public-key) protocols [4] in which a hashed image of a file is "signed" by encrypting it using a private key of the sender. The signed, hashed version is included in the file and can be verified by the target by applying the sender's public key to the transmission to reveal the hashed file, generating another hashed version of the file, and comparing the two. The hash in our system will be generated for the annotations that are included in the file. In our evaluation of the effectiveness of this methodology, we will measure the transfer and (hash) verification delay that is introduced.

The third security methodology we will consider is proof-carrying codes. Proof-carrying codes [31, 32] (PCC) have been proposed as an alternative to cryptographic protocols and type-safe program representations. A soundness proof generated from an optimized binary at the source is transmitted with the program (optimized binary). A soundness proof is information about the program that shows that the program behaves according to a set of safety rules (established and standardized ahead of time as is done by the Java Language Specification [14]). The proof is verified using an efficient proof checker at the destination [10]. PCC verification has been reported to be efficient [10] and the proof size for an entire program can be small [33]. We believe that by using such security strategies on annotations alone the overhead introduced can be kept very small. We will identify and implement the strategy we empirically find to be the most efficient in a mobile environment.

Using a security-enabling extension to our annotation framework, we will consider and implement new, previously insecure, annotation-based optimizations. We differentiate architecture-independent optimizations from those that are low-level, or architecture-specific. Examples of the former include null pointer and array bounds check elimination, stack allocation of heap objects, static binding of virtual method calls, etc. We will perform such optimizations off-line and communicate the results to the compilation system via annotation. For example for null pointer check elimination, we will use static analysis to determine which checks are redundant, and indicate the instructions for which null checks must be performed. At the destination, no null check code will be generated except immediately prior to the instructions indicated by the annotation.

The second type of optimization is architecture-specific. Optimizations of this kind include register allocation, memory placement of code and data, instruction scheduling, etc. We propose to handle such architecture-specific (low-level) optimizations using a novel methodology: *Annotation parameterization* by hardware characteristics. We will generate multiple annotations for the same architecture-specific optimization based on assumptions about system characteristics, e.g., cache size, instruction set, number of registers, etc. When the mobile program is requested by the target machine, the architectural details about the target will be included in the request. This information will be used to select the most appropriate annotations to include in the transmission. Our goal is to enable very high-performance of mobile programs for a subset of popular architectures. However, to maintain portability of code, only our annotations are architecture-specific and each will have an architecture-independent version available.

An example of an architecture-specific annotation that we plan to implement is heap object placement. We can identify temporal relationships between heap objects using off-line profiles and use this information to guide their memory placement to reduce cache conflicts [6]. Using a parameterized annotation, we can generate one annotation that indicates the critical relationships between objects assuming a direct-mapped, level 1 cache and another annotation assuming a 4-way set associative (SA), level 1 cache. The relationships are different since objects mapped to the same cache line in a 4-way SA cache do not necessarily conflict and in a direct-mapped cache they do. As such, the relationships between objects differs. If a request for the mobile program is made by a machine with a direct-mapped cache, e.g., an Alpha 21164, the direct-mapped placement annotation will be sent with the program to enable improved cache performance.