Elastic Executions from Inelastic Programs

Iulian Neamtiu
Department of Computer Science and Engineering
University of California, Riverside, CA, USA
neamtiu@cs.ucr.edu

ABSTRACT

In this paper we present an approach and tool named ELASTIN for transforming inelastic programs—programs written with a specific platform or a fixed set of Cloud resources in mind—into elastic applications that run on elastic platforms by adapting, at runtime, to changes in the available resources. With ELASTIN, programmers can develop their applications with various specific configurations in mind, and let the compiler and runtime system take care of combining these configurations into a single elastic application that can safely switch between configurations on-the-fly, at runtime. We used ELASTIN to elastify two popular applications, the SQLite database engine and the Kiss FFT library, and found programmer burden to be very low. Benchmarks indicate that ELASTIN is effective in practice, and reconfigurations are in the sub-millisecond range. We envision this approach being useful in any domain where quick runtime adaptation is necessary due to changes in underlying resources.The approach can also be a stepping stone towards migrating legacy applications to the Cloud.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Reliability, availability, and serviceability; D.3.4 [**Processors**]: Compilers, Run-time environments; I.2.2 [**Automatic Programming**]: Program modification, Program transformation

General Terms

Languages, Measurement, Performance, Reliability

Keywords

Elastic software, self-adaptive systems, resource-aware applications, runtime adaptation, Cloud computing, dynamic software updating

1. INTRODUCTION

We are currently witnessing a fundamental shift in how applications are developed and deployed. On the end-user side, Web applications are replacing traditional applications. On the service-provider side, monolithic server programs are transitioning to Cloud computing applications. This shift is beneficial to end-users (Web applications run a variety of hardware platforms and provide increased interactivity) and service providers (Cloud computing significantly reduces administration burden and start-up costs). However, the shift is problematic for application developers, as they can make very few assumptions about the platform (CPU, memory, disk space) their applications will run on, and applications have to be rewritten significantly, if not constructed from scratch, to run on a wide range of platforms. For example, a Web application for online video streaming might run fine on laptops, tablets, and certain smartphone models, but not on resource-poor phones. Similarly, the elastic resources available in the Cloud cannot be taken advantage of unless programs are written with elasticity[1] in mind.

Ideally, applications would be *self-adaptive*: detect the operating conditions, and adapt to changes in conditions to deliver the best possible user experience. This paper presents an approach named ELASTIN that takes a step towards enabling the construction of self-adaptive applications. Given various source code configurations (e.g., one version optimized for maximum speed, another optimized for low memory consumption, and yet another optimized for low storage requirements), in ELASTIN we combine all these versions into a single application that can switch between configurations at runtime, without shutting down the application and while preserving in-memory state and providing certain safety guarantees. The advantage of using ELASTIN is that it allows developers to focus on one application configuration at a time, while enabling the application to safely navigate through configurations at runtime and without disruption. Configurations usually represent various points in the space-time trade-off. For example, one configuration can be memory-intensive but fast, another configuration can be designed with low memory footprint in mind, but running slower. We describe the approach in Section 2.

In Section 3 we show how we used ELASTIN to construct elastic applications based on various configurations for two popular open source programs: the SQLite database engine and the Kiss Fast Fourier Transform library. These configurations must be selected at compile-time, hence applications cannot adapt to changes in resources while they run. The SQLite configurations are designed to trade memory for query execution time, and vice versa. The Kiss FFT configurations are designed to provide optimal speed, depending on whether a floating-point unit is present or not. We were able to apply ELASTIN to these applications with few changes to the source code and minimal effort for supporting on-the-fly configuration changes.

SEAMS '11, May 23-24, 2011, Waikiki, Honolulu, HI, USA

[1]Note that our elasticity definition means runtime adaptation to changes in hardware resources; IBM's definition of elasticity [25] includes fault-tolerance, self-healing and administrative simplicity.

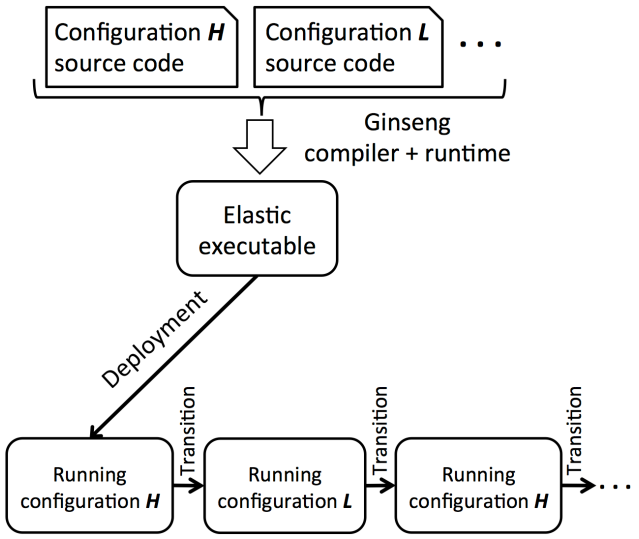

Figure 1: High-level overview of ELASTIN.

In Section 4 we provide the results of an experimental evaluation of ELASTIN. We show how SQLite can take advantage of large amounts of memory available at runtime to increase query processing rate. When the available memory shrinks, SQLite can still function, albeit at a reduced query rate. This scenario is useful in a Cloud context, where additional memory that might become available allows the SQL server to speed up queries. Similarly, we show how Kiss FFT can take advantage of the presence of a floating-point unit to speed up FFT computation, yet still function when the floating-point unit is turned off. This scenario is useful on limited-power devices that turn off the floating-point units in their CPUs to save energy. Finally, we measured the time required to effect configuration changes, and found this to be modest, at most 672 μsec. In Section 5 we present future research directions and in Section 6 we review related work.

In short, this paper makes the following contributions:

- An approach for transforming inelastic programs into elastic applications which will run elastic executions.
- A study and evaluation of our approach on two popular open source programs—SQLite and Kiss FFT—used in the construction of mobile, desktop and server software.

2. APPROACH

In this section we present our approach, named ELASTIN. ELASTIN stands for ELASTicity with gINseng, as it is built upon our existing infrastructure for dynamic software updating, named Ginseng [17, 15]. ELASTIN currently works on programs written in C.

2.1 Overview

A high-level overview of ELASTIN is provided in Figure 1. ELASTIN is a program "elastifier," in that it takes as input different source code configurations, each optimized for a different runtime configuration, and transforms them into an elastic executable that can switch between configurations at runtime.

To make the presentation clearer, throughout the paper we use two standard configurations: ***H***, which stands for "***H***igh performance" and ***L***, which stands for "***L***ow resource consumption." The Ginseng compiler combines all source code configurations into an elastic executable, along with runtime support for safe configuration switching. The elastic executable is then deployed. Whenever conditions in the environment (e.g., as observed by sensors), or changes in resources lead to the decision that the current execution configuration should change, the runtime system, added to the executable by the compiler, will trigger configuration changes (transitions).

2.2 Development

Programmers write their code for versions **H**, **L**, etc. assuming a single version, optimized for a specific configuration—it is ELASTIN's responsibility to perform data migration and ensure safe configuration changes. We believe this model is conceptually simple, as the programmers have to only focus on one configuration, not on state migration or on the safety of configuration changes.

2.3 Compilation

The Ginseng compiler is in charge of several tasks. First, it rewrites the original C code into C code that can be modified at runtime. Second, it performs a static safety analysis and annotates the code with analysis results; this information is used later on, at runtime, for safe configuration changes. Third, it computes the source code differences between different configurations, and adds these code differences to the elastic executable. Fourth, it generates data migration functions that will change the in-memory data upon configuration changes. Finally, it compiles the C code to binary and links-in a runtime system whose job is to perform configuration changes. All these tasks are performed by the Ginseng infrastructure. The performance penalty introduced by Ginseng, i.e., performance decrease due to runtime configuration change support is typically 0–6% [17, 15].

2.4 Configuration Changes

For now, we present configurations to Ginseng as program updates, i.e., a $\mathbf{H} \rightarrow \mathbf{L}$ configuration change represents a dynamic update (dynamic patch loading and state transformation) from the **H** source code to the **L** source code. This solution does not scale very well, though, since with any configuration change Ginseng loads a new dynamic patch, which will lead to memory bloat as configuration changes accumulate; we plan to address this issue in future work.

The Ginseng runtime system carries out an on-the-fly change to the new configuration by loading the dynamic patch: configuration-specific functions are redirected to their version for the new configuration, and a lazy data transfer, that converts data to the format expected by the new configuration is started; the data conversion is lazy for performance reasons [17]. For this work we signaled configuration changes manually, but in the future we expect changes to be triggered by a control module, based on policy and information from sensors.

2.5 Safety

Dependability is an important goal for self-adaptive systems [2]. As the source code configurations are written with a single configuration in mind and no runtime configuration change, it is essential that ELASTIN take steps towards guaranteeing the safety of configuration changes. For example, Kiss FFT defines its main FFT data structure in the **L** configuration as:

```
struct kiss_fft_cpx_old { int r ; int i ;}
```

and as:

```
struct kiss_fft_cpx_old { float r ; float i ;}
```

in the **H** configuration. A runtime change in the functions that operate on these structures without an appropriate update in the un-

derlying data will lead to type safety violations and, most likely, a crash. Ginseng's compilation strategy (that allows data to be updated at runtime so that it changes representation) and safety analyses prevent such type safety violations [17]. Furthermore, Ginseng allows programmers to specify program regions that should be executed *atomically* with respect to a configuration change, but still allow changes to be performed immediately. For example, the programmer might designate an event-processing loop as an atomic region, to avoid situations where **H** code started processing an event, there is a configuration change, and **L** code finishes the processing, which even when type-safe, might lead to inconsistency. We call this property *transactional version consistency* [16], and we used it to ensure that the long-running loops in both SQLite and Kiss FFT are atomic with respect to configuration changes. Note that the safety of dynamic software updating has been proved undecidable, in the general case, by Gupta [8]; nevertheless, with type safety and version consistency, Ginseng goes a long way toward providing configuration change safety guarantees.

3. EXPERIENCE

We used ELASTIN to "elastify" two open-source programs: the SQLite database engine (version 3.6.9), and the Kiss FFT library (version 1.2.9). We chose these programs because they are popular, make compile-time decisions about performance trade-offs, and are deployed on a variety of platforms, from smartphones to desktops and servers. For each program, we selected two configurations, **H** and **L**, and used Ginseng to switch between these configurations at runtime. In the remainder of this section we briefly present each program and its compile-time trade-offs, and discuss changes we made to the source code to prepare it for processing with Ginseng.

3.1 Applications

SQLite is a small-footprint SQL engine [11] written in C. SQLite is very popular, with an estimated 500 million installations [23]. Because it is used on wide range of platforms, from embedded systems to desktops and servers, SQLite comes with several different memory allocators. The default memory allocator—which uses the system malloc—is space-efficient, but possibly not fast or deterministic enough for embedded systems. Therefore, SQLite offers a second memory allocator named **memsys5** that does not call malloc, uses a fixed amount of memory, rounds allocation requests to the next power of two and provides "mathematical guarantees against fragmentation and breakdown" [10]. Switching between the default and **memsys5** allocators requires recompilation [9]. Our **L** configuration consisted of SQLite with the default, malloc-based allocator; the **H** configuration consisted of SQLite with a 64MB **memsys5** allocator. The rationale for choosing these configurations is to demonstrate how an SQL server could adapt to elastic Cloud resources, in this case expanding or shrinking memory.

Kiss FFT is a Fast Fourier Transform library [4] written in C. According to its documentation, the library is "able to do fixed or floating point [FFT] with just a recompile." In addition, Kiss FFT has support for other optimized configurations, such as OpenMP or SIMD. The fixed-point (integer), floating point, OpenMP, or SIMD configurations are selected at compile-time. Our **L** configuration consisted of Kiss FFT with the integer-based FFT computation; the **H** configuration consisted of Kiss FFT with the floating-point based FFT computation. The rationale for choosing these configurations is to demonstrate how performance of multimedia applications using Kiss FFT can be better adapted to changes in the hardware platform, e.g., by allowing FFT processing even when the floating-point unit in the CPU is turned off to conserve power.

3.2 Source Code Changes

When building dynamically updateable applications with Ginseng the programmer may need to intervene at two phases: when preparing the source code for compilation, and when creating dynamic patches—an update consists of loading a dynamic patch. ELASTIN inherits these burdens. We present details on the strategy we followed, and programmer effort (annotations or lines of code) for each of these phases, in turn, for our two test programs.

For SQLite, we had to add 1 line of code for identifying a long-running loop and enforcing transactional version consistency (as explained in Section 2.5), 1 line to pacify Ginseng's type safety analysis, and 14 lines in auto-generated dynamic patches that contain the configuration change logic (transitioning from **H** to **L** and vice versa). The configuration change code consisted of allocating a large block of memory for **memsys5** and telling SQLite to switch allocators.

For Kiss FFT, we had to add 6 lines of code for identifying long-running loops and enforcing transactional version consistency, 2 lines to pacify Ginseng's type safety analysis, and 7 lines in configuration change code (completing Ginseng's auto-generated dynamic patches).

4. EXPERIMENTAL RESULTS

We now show how the elastic applications generated by ELASTIN are effective at exploiting the performance/resources trade-off by measuring the performance and resource consumption in the **H** and **L** configurations. For our experiments, we started each program in a **H** configuration, named $\mathbf{H_1}$, changed the configuration to a **L** configuration named $\mathbf{L_1}$, then switched again to a $\mathbf{H_2}$, and finally switched to $\mathbf{L_2}$. We also present measurements on configuration transition times.

4.1 Experimental Setup

We conducted our experiments on a two-CPU, quad-core Xeon @ 2.33GHz system with 12GB of RAM and a RAID5 array of three Western Digital RE3 1TB@7200 rpm hard drives. The test system ran CentOS 5.5, Linux kernel version 2.6.18. The code generated by Ginseng was compiled with `gcc` 4.1.2 using the default application-specific compile flags.

4.2 Elastic Execution

4.2.1 SQLite

To demonstrate the performance/memory trade-off in SQLite, we conducted a database query performance experiment. We first populated an empty database with 4 tables, each containing 10,000 records. After populating the database, we ran a 10,000 query benchmark [2] in the first configuration ($\mathbf{H_1}$). At the completion of this query benchmark, we switched configuration on-the-fly to $\mathbf{L_1}$, and ran another 10,000 query benchmark; then switched configuration on-the-fly to $\mathbf{H_2}$, benchmarked with 10,000 queries and finally switched to $\mathbf{L_2}$, and benchmarked again. The $\mathbf{H_1}$/$\mathbf{H_2}$ and $\mathbf{L_1}$/$\mathbf{L_2}$ configurations were identical, but we label them with sequence numbers for clarity.

In Figure 2 we plot the SQLite query processing throughput, measured each second. The top of the figure indicates the configuration; arrows configuration changes (transitions). The elastic

[2]The database schema for our experiment was adapted from Mozilla, one of the programs that uses SQLite. The queries follow the frequency we observed in an empirical study on how SQLite is used by Mozilla: 40% `SELECT`, 30% `INSERT`, 20% `UPDATE`, 10% `DELETE`. The database contents, as well as the new tuples and attributes used in `INSERT` and `UPDATE` were created randomly.

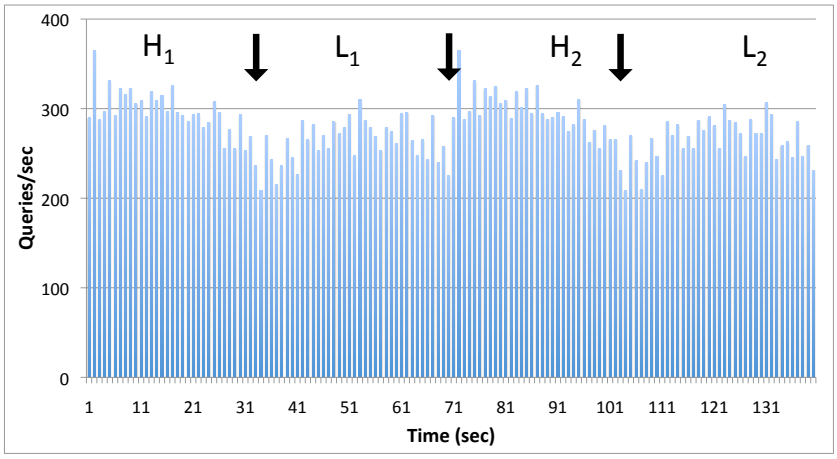

Figure 2: SQLite query performance; arrows indicate configuration changes.

Metric \ Configuration	$\mathbf{H_1}$	$\mathbf{L_1}$	$\mathbf{H_2}$	$\mathbf{L_2}$
Query performance (queries/sec)	296 (+12.3%)	263	295 (+12%)	264
Memory consumption (KB)	2386 (+130%)	1028	1210 (+16.8%)	1044

Table 1: SQLite: Average query performance and memory consumption, as absolute numbers and percents relative to $\mathbf{L_1}$ and $\mathbf{L_2}$'s average.

SQLite ran in configuration $\mathbf{H_1}$ from second 1 to 33, in configuration $\mathbf{L_1}$ from second 34 to 70, in configuration $\mathbf{H_2}$ from second 71 to 103, in configuration $\mathbf{L_2}$ from second 104 to 140. As evident from the figure, while performance varies within a configuration, it is clear that $\mathbf{H}$ configurations have higher performance than $\mathbf{L}$ configurations, and take less time (33 seconds v. 36 seconds).

In Table 1 we present the average values for query performance and effective memory consumption in each configuration. The percentages indicate the performance gain, and increase in memory consumption, respectively, for the $\mathbf{H_1}$ and $\mathbf{H_2}$ configurations compared to the average of $\mathbf{L_1}$ and $\mathbf{L_2}$ configurations. As we can see, query performance is about 12% higher in the $\mathbf{H}$ configurations than in the $\mathbf{L}$ configurations; the trade-off is higher memory consumption, 130% in $\mathbf{H_1}$ and 16.8% in $\mathbf{H_2}$. The memory consumption is higher for $\mathbf{H_1}$ compared to $\mathbf{H_2}$ due to residual data in table caches from the initial table population that precedes benchmarking. Note that the $\mathbf{H_1}$ and $\mathbf{H_2}$ memory values in Table 1 show effective memory consumption, but that might be deceptively low, as 64MB were pinned down for use by the **memsys5** allocator in $\mathbf{H}$ configurations (Section 3.1).

4.2.2 Kiss FFT

To demonstrate the performance/power consumption trade-off in Kiss FFT, we conducted the following experiment: we started the elastic Kiss FFT in configuration $\mathbf{H_1}$, i.e., floating-point FFT computation, and computed 50,000 FFT transforms (on 1,024 points), measuring the FFT computation rate each second; at the completion of this benchmark, we switched configuration on-the-fly to $\mathbf{L_1}$, i.e., integer-based FFT computation and ran another 50,000 FFTs; then switched configuration on-the-fly to $\mathbf{H_2}$, benchmarked with 50,000 FFTs and finally switched to $\mathbf{L_2}$, and benchmarked again.

Application	Transition time (μsec)		
	$\mathbf{H_1} \rightarrow \mathbf{L_1}$	$\mathbf{L_1} \rightarrow \mathbf{H_2}$	$\mathbf{H_2} \rightarrow \mathbf{L_2}$
SQLite	672	158	346
Kiss FFT	125	144	139

Table 3: Transition (configuration change) times.

In Figure 2 we plot the FFT processing throughput. Again, the top of the figure indicates the configuration and arrows indicate configuration transitions. The elastic Kiss FFT ran in configuration $\mathbf{H_1}$ from second 1 to 25, in configuration $\mathbf{L_1}$ from second 26 to 62, in configuration $\mathbf{H_2}$ from second 63 to 92, in configuration $\mathbf{L_2}$ from second 93 to 129. As in the SQLite case, $\mathbf{H}$ configurations have higher performance.

In Table 2 we present the average values for FFT throughput during each configuration. Performance is about 46.9%, and 23.4%, respectively, higher in the $\mathbf{H}$ configurations than in the $\mathbf{L}$ configurations. While the performance is low in the $\mathbf{L}$ configurations, this might be acceptable if the multimedia application using Kiss FFT runs on a device low on battery, as it allows the application to continue running without quickly depleting the battery. We did not measure memory consumption for the Kiss FFT experiments, as for this application high performance is achieved when using the floating point unit, rather than when using more memory.

4.3 Service Disruption

One of the main goals of this work is to allow on-line configuration changes, i.e., without service interruption. By performing configuration switches on-line rather than by stopping the application and restarting at a different configuration we preserve useful application state, leave connections open, and sustain service.

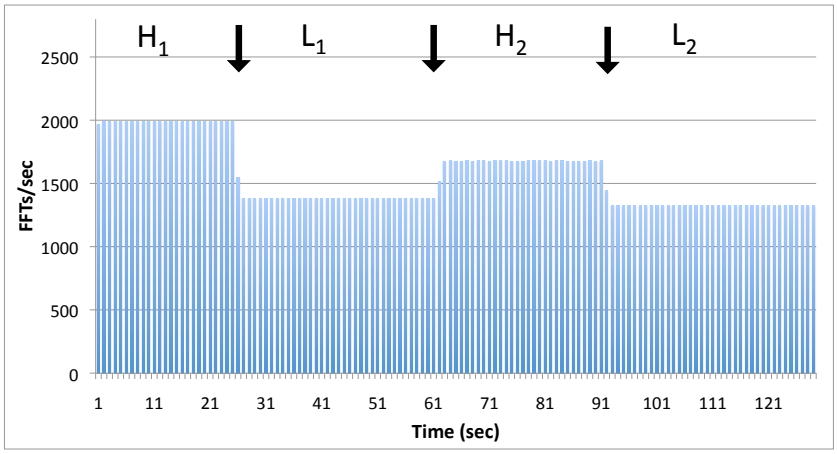

Figure 3: Kiss FFT throughput; arrows indicate configuration changes.

Metric \ Configuration	H_1	L_1	H_2	L_2
Throughput (1,024-point FFTs/sec)	1990 (+46.9%)	1383	1672 (+23.4%)	1326

Table 2: Kiss FFT: Average throughput, as absolute numbers and percents relative to L_1 and L_2's average.

However, the service will still be paused shortly, while the runtime system performs configuration changes (Section 2.4). Table 3 shows the transition times (service pauses) introduced by a configuration change. For SQLite, transition times vary between 158 and 672 μsec, hardly a problem for application users; for Kiss FFT, these pauses are even shorter, in the 125–144 μsec interval.

5. FUTURE WORK

We plan to extend this work in several directions. In the current ELASTIN implementation, configuration changes are triggered manually. However, a self-adaptive system needs to monitor the program and the environment, and automatically detect conditions that warrant a configuration change, and trigger the change. For example, for Kiss FFT, a monitor and trigger could look like this:

```
switch ( battery_life ) {
  case (< 10%) :
    configuration <– L;
  case (> 30%) :
    configuration <– H;
  default:
    // no configuration change needed
}
```

A second future direction is to elastify, and then run benchmarks on, Cloud applications. The Cloud is an excellent target for our approach, due to its elastic resource model. Moreover, due to the novelty of the Cloud platform, there are few applications written from scratch for the Cloud, that can take advantage of elastic resources; rather, legacy applications must be retrofitted for the Cloud, and ELASTIN is ideal in that regard, since it accepts legacy applications and converts them into elastic ones.

6. RELATED WORK

Runtime adaptation has been achieved by various means: with programming language support [7], with architecture-based support [18, 19, 6, 13], via adaptation frameworks [5], or with compiler, runtime, and OS support [5, 12, 22, 3, 1].

Ghezzi et al. [7] recently introduced an approach called ContextErlang that implements Context-Oriented Programming in Erlang, to support the construction of self-adaptive software. They implement context-dependent behavior by allowing messages to be processed by different callback modules called *variations*; Erlang permits dynamic binding of variations, which in effect changes the context. Our approach is similar, in the sense that Ginseng, a dynamic updating system, provides the necessary on-the-fly change mechanisms to ELASTIN. Their notions of variation, and variation activation, respectively, are similar to our notions of configuration and configuration change. However, their solution is at the programming language level, whereas ours works at the compiler and runtime level; also they target Erlang applications, whereas we target programs written in C.

Architecture-based runtime evolution [18, 19] relies on an explicit architectural model that allows runtime architectural changes, e.g., component addition, removal, replacement and structural reconfiguration. In this model, a part of the system's architecture is available at runtime (as connectors, components, and implementation mappings) so it can be inspected and changed. The model permits runtime changes, described by architectural modification operators. Their notion of safety is that visible system states are always consistent (pre- and post-change). The Rainbow framework [6] uses a two-layer model (architecture layer and system layer) to implement architectural-based self-adaptation. System operation is specified via invariants and adaptation strategies. The system layer uses probes, resource discovery and effectors to cooperate with the architecture layer. The architecture layer takes into account probe and resource information and uses gauges to detect whether the system conforms to the specified invariants. If a violation is detected, an adaptation engine evaluates possible adaptation courses (as specified by strategies) and triggers adaptation by notifying the effectors. Our approach is at a lower level. We do not require ap-

plications to be designed using a special architecture, rather we aim to elastify off-the-shelf applications. As a consequence, albeit formally proved, our notion of safety is low-level (code-based version consistency and type safety); we cannot provide guarantees that certain architectural consistency properties or integrity constraints are preserved when configurations change.

Chang and Karamcheti [5] developed a framework for enabling runtime application adaptation on distributed platforms; applications are developed using a tunability interface—language-level annotations that allow an application to specify alternate control paths, resource requirements for a configuration, QoS metrics and configuration transition functions. Given these annotations, their framework generates monitoring and steering agents that make adaptation decisions based on application behavior and available resources. A virtual-machine based profiling infrastructure computes a mapping from control parameters to output quality in various configurations, to derive runtime transition decisions. Their approach explores a different point in the design space: programmers are much more involved in adaptation in their case, by structuring their applications around the tunability interface. In contrast, in our approach the focus is on one source code configuration at a time (e.g., conceivably each configuration being programmed by a separate team with expertise in that configuration space), and we require no application restructuring. Their system, however, implements automatic monitoring and steering, which we currently lack.

The K42 operating system from IBM Research [12, 22, 3, 1] supports autonomic computing via hot-swapping classes: all the code is encapsulated behind class interfaces, and reconfigurations to code or data consist of dynamic updates to classes. All classes that might be subject to dynamic updating have to provide state import and state export methods, thus upon update, the new version imports the old version's exported state. In this approach, applications have to run on K42 and be written from scratch with hot-swapping in mind. In contrast, with ELASTIN we can take off-the-shelf inelastic applications and convert them into elastic applications without the need for large-scale changes or re-engineering.

Sadjadi et al.'s work on TRAP/J [20] and ACT/J [21] achieves dynamic adaptation for Java programs by using aspects to compile Java programs into adapt-ready programs that can be adapted ("transparently shaped") at runtime. Given an application, the programmer selects a subset of classes that should be adaptable, and the compiler uses aspect weaving to permit changes to these classes' code. It is unclear to us what steps these systems take to guarantee the safety of runtime adaptation. In our approach the programmer must specify a small number of annotations, mainly for safety, however the programmer is not required to anticipate which parts of the software might change.

Kramer and Magee [14] introduced the notion of *quiescence* for reasoning about, and guaranteeing, dynamic update safety (pre- and post-update state consistency). Vandewoude et al. [24] refined this into a notion of *tranquility* that is less strict than quiescence, yet ensures state consistency. Unfortunately both quiescence and tranquility are too strict to be applicable to the types of applications (large C programs with long-running event loops, large updates) we examined in this paper or in the past [17, 15].

Acknowledgments. This research was supported in part by NSF grant CCF-0963996. We thank the anonymous referees for their helpful comments on this paper.

7. REFERENCES

[1] Appavoo, J. et al. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 2003.

[2] B. Cheng et al. Software engineering for self-adaptive systems: A research road map. In *Dagstuhl Seminar Proceedings*, number 08031, 2008.

[3] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *USENIX ATC*, pages 279–291, 2005.

[4] M. Borgerding. Kiss FFT. `http://sourceforge.net/projects/kissfft/`.

[5] F. Chang and V. Karamcheti. Automatic configuration and run-time adaptation of distributed applications. In *HPDC'00*, pages 11–20.

[6] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37:46–54, 2004.

[7] C. Ghezzi, M. Pradella, and G. Salvaneschi. Programming language support to context-aware adaptation: a case-study with erlang. In *SEAMS '10*, pages 59–68.

[8] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE TSE*, 22(2), 1996.

[9] D. R. Hipp. Compilation options for sqlite. `http://www.sqlite.org/compile.html`.

[10] D. R. Hipp. Dynamic memory allocation in sqlite. `http://www.sqlite.org/malloc.html`.

[11] D. R. Hipp. Sqlite. `http://www.sqlite.org/`.

[12] The K42 Project. `http://www.research.ibm.com/K42/`.

[13] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07*, pages 259–268.

[14] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE TSE*, 1990.

[15] I. Neamtiu and M. Hicks. Safe and timely updates to multi-threaded programs. In *PLDI'09*, pages 13–24.

[16] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL'08*.

[17] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *PLDI'06*, pages 72–83.

[18] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *ICSE '98*, pages 177–186.

[19] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion '08*, pages 899–910.

[20] S. M. Sadjadi, P. K. McKinley, B. H. Cheng, and R. K. Stirewalt. TRAP/J: Transparent generation of adaptable Java programs. In *DOA'04*, pages 1243–1261.

[21] S. M. Sadjadi, P. K. McKinley, and B. H. C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. DEAS '05.

[22] C. Soules, J. Appavoo, K. Hui, et al. System support for online reconfiguration. In *USENIX ATC'03*, pages 141–154.

[23] SQLite team. Most widely deployed SQL database. http://www.sqlite.org/mostdeployed.html.

[24] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE TSE*, 2007.

[25] R. Wisniewski. Innovations within reach: Stretching the limits with elastic software. In *IBM WebSphere Developer Technical Journal*, March 2010. `http://www.ibm.com/developerworks/websphere/techjournal/1003_inreach/1003_inreach.html`.