Squirrel: Architecture Driven Resource Management

Inti Gonzalez-Herrera, Johann Bourcier, Walter Rudametkin, Olivier Barais, Francois Fouquet

Squirrel: Architecture Driven Resource Management

Inti Gonzalez-Herrera
University of Rennes 1 - IRISA
35042 Rennes, France
inti.glez@irisa.fr

Johann Bourcier
University of Rennes 1 - IRISA
35042 Rennes, France
johann.bourcier@irisa.fr

Walter Rudametkin
University of Lille - INRIA
Villeneuve d'Ascq, France
walter.rudametkin@inria.fr

Olivier Barais
University of Rennes 1 - IRISA
35042 Rennes, France
barais@irisa.fr

Francois Fouquet
University of Luxembourg
Luxembourg, Luxembourg
rancois.fouquet@uni.lu

ABSTRACT

Resource management is critical to guarantee Quality of Service when various stakeholders share the execution environment, such as cloud or mobile environments. In this context, providing management techniques compatible with standard practices, such as component models, is essential. Resource management is often realized through monitoring or process isolation (using virtual machines or system containers). These techniques (i) impose varying levels of overhead depending on the managed resource, and (ii) are applied at different abstraction levels, such as processes, threads or objects. Thus, mapping components to system-level abstractions in the presence of resource management requirements can lead to sub-optimal systems. We propose Squirrel, an approach to tune component deployment and resource management in order to reduce management overhead. At runtime, Squirrel uses an architectural model annotated with resource requirements to guide the mapping of components to system abstractions, providing different resource management capabilities and overhead. We present an implementation of Squirrel, using a Java component framework, and a set of experiments to validate its feasibility and overhead. We show that choosing the right *component-to-system* mappings at deployment-time reduces performance penalty and/or volatile main memory use.

Keywords

resource management, components, architecture adaptation

1. INTRODUCTION

Component model implementations represent high level concepts, such as component instances, by means of mapping them to system-level abstractions like objects, threads, processes or virtual machines. Each mapping has unique features in terms of performance, memory footprint, etc. However, these mappings are often done in a once-size-fits-all manner, allowing some choices to optimize for memory use while others might, for example, improve inter-component communication. Interestingly, system abstractions offer varying resource management capabilities that differ in how they impact the application.

To address this issue, we propose Squirrel, an approach to resource management for component models that aims at reducing overhead. In Squirrel, the application is deployed with a model containing resource usage contracts for each component and a detailed view of the system. These metadata are used to choose at deployment-time the best way of representing each component in terms of system abstractions. This contrasts with the *traditional* approach of binding the representation during the design of the component model and results in the final runtime representation of the system only being known after deployment.

In this paper we discuss an approach to resource management applicable to any component model. To validate the feasibility of our proposal, we present a reference implementation for a Java-based component model. A set of experiments validate its feasibility and show various aspects of its overhead. The results demonstrate that choosing the right *component-to-system* mappings at deployment-time can reduce performance overhead and/or volatile main memory footprint. The contributions of this paper are as follow: i) An approach for architecture driven resource management that leverages structural information to guide the mapping of component model concepts onto system-level abstractions. ii) A reference implementation of the Squirrel framework for a Java-Based component platform. iii) A performance comparison showing how different mappings can impact the overhead of the system and how the approach behaves in comparison to state-of-practice approaches for resource management.

The remainder of this paper is organized as follows. The next section presents some foundational work we use along this paper. Section 3 describes the approach and presents how we leverage metadata to drive resource management. In Section 4 we propose a reference implementation of Squirrel for a Java-based component platform. A validation of the

ACM 978-1-4503-3738-0.
http://dx.doi.org/10.1145/0000000.0000000

implementation through a set of experiments is presented in Section 5. Finally, Section 6 discusses related work and Section 7 presents our conclusions and future work.

2. BACKGROUND ON RESOURCE MANAGEMENT

Current middleware provide limited resource awareness capabilities. Following the same ideas as in [9] where authors explain that the distribution concern should not be hidden from developers, we believe that resource management should not be hidden from application developers. We envision combining monitoring techniques and system-layer resource management techniques to achieve this goal. This section presents a summary of two underlying techniques used to monitor and reserve resources.

Cgroups (control groups) is a Linux kernel feature to limit, account, and isolate the resource usage of processes. It provides a low-level API to access properties on resource usage that allow to i) limit memory consumption per task, ii) assign a minimum percentage of CPU time to the task, iii) establish a minimum and maximum throughput for I/O block devices and network throughput per task, and iv) measure the resources a task uses. Cgroups are used in particular in the context of lightweight process virtualization.

Scapegoat provides an application-level adaptive resource monitoring framework [8]. Each component is augmented with a contract that specifies its resource usage . The framework adjusts the monitoring level to minimize overhead while still allowing precise accounting when needed. The adjustment is done by selecting, through a heuristic, components that should be deeply monitored using intrusive instrumentation to check their contracts, while using a lightweight monitoring mode the rest of the time.

3. APPROACH

The main concept in Squirrel is the *resource-aware container*. Such containers are logical entities that take care of the resource management concern. By logical we mean that it is not important, from a functional point of view, how a container achieves resource management. Instead, a resource-aware container is an entity that *wraps* a set of components and offers the following properties:

- **Resource consumption monitoring** refers to the ability to assess the quantity of resources used by a component.
- **Resource reservation** is the capacity to ensure a given amount of resources will be available whenever a component demands it.
- **Resource isolation** guarantees that a component's behavior in terms of resource usage does not interfere with the behavior of another component.

Wrapping a set of components can be considered a soft definition because the *membrane* of a resource-aware container limits the behavior of the contained components only when it is relevant to the resource management concern. For instance, components within different containers can still communicate directly with each other through their interfaces without intervention of their containers as long as such communication does not affect the resource under management.

In Squirrel we propose to automatically select, deploy and configure resource containers to manage resource usage. The novelty is that we delay the selection of the container's implementation till deployment-time in order to have knowledge about the exact conditions of the system and thus minimize the overhead of the resource management system. This idea is supported by the claim that components often require disjoint sets of resource types. Our framework is composed of three elements: i) a mechanism to describe the management requirements of an application, ii) an admission control scheme in the middleware to handle the global view of resource availability, and iii) mechanisms to map component model concepts to system-level abstractions. In the following subsections we describe our framework and its elements.

3.1 Architecture adaptations for managing resources

Modern application development models, such as component-based systems, promote the usage of Architecture Description Languages (ADL) or configuration models to check properties on the system's structure and to drive system deployment. In Squirrel, we propose to enhance this layer with metadata regarding resource reservation and to use these metadata to efficiently drive resource reservation offered at the system level. The idea is to follow a gray-box approach where we automatically adapt a component-based application by applying an architecture pattern to isolate a component within a resource-aware container.

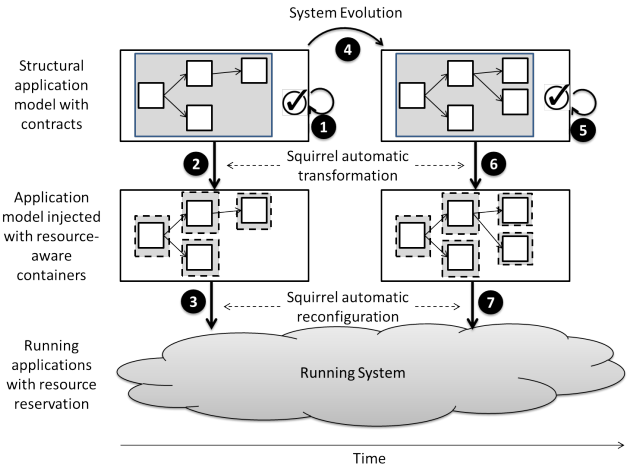

Figure 1: Approach for resources reservation

As illustrated in Figure 1, Squirrel follows an automatic process to manage resources. Squirrel receives an application's model, enhanced with contracts on resource reservation. Squirrel performs admission control to check the validity of the contracts on resource usage with respect to the available resources in the execution environment. If the contracts are consistent with available resources, the process continues, if not, the application's model is refused. Then, as depicted by arrow 2, Squirrel automatically transforms components or the configuration/architecture model by iso-

lating components in resource-aware containers that can be finely configured to decrease the resource management overhead. Finally, as depicted by arrow 3, Squirrel reconfigures the running system. When the application evolves (arrow 4), Squirrel attempts to preserve resource reservation properties while processing the new model (arrow 5, 6 and 7).

3.2 Describing resource requirements

Beugnard et al. discuss the extending Meyer's *Design-By-Contract* idea to software components [1]. They classify component contracts into four categories: syntactic, semantic, synchronization, and Quality of Service (QoS level). There is no de-facto standard to describe component contracts, but many domain specific interface description languages contain such metadata. This paper assumes that components have contracts to deal with resource reservation (QoS level). A contract in Squirrel defines component resource requirements written in terms of resource types, quotas and expected component usage.

- **Definition 1** A resource type indicates any class of computational resource that is useful to a component. Its consumption must be susceptible to monitoring and reservation. In this paper, we consider CPU, Memory, Network Bandwidth and IO Throughput.
- **Definition 2** Expected component usage describes the expected number of external invocations of each method of the component interface. In short, let C be a component instance, then

  $$\forall I \in C_{Interfaces}, \forall M \in I_{methods} EU_{IM}$$

  is the number of expected invocations of method M, per second.

A **contract in Squirrel** is a set of tuples with the form $\langle RT, N, MU \rangle$ where RT is a resource type, N the maximum amount of resources to reserve, and MU is the measuring unit used for this resource type. Optionally, Squirrel supports the definition of a set of tuples with the form $\langle I, M, EU_{IM} \rangle$ where I is a component interface, M is a method of the interface, and EU_{IM} the expected usage. Implementations of the Squirrel approach must provide a way to define contracts with these concepts. We use a domain-specific language to describe contracts.

3.3 Admission control

Providing resource reservation in a component based framework requires checking if components' resource-aware contracts are compatible with the resources available in the execution environment. By checking the availability of resources, the platform controls component admissions.

To support resource management at runtime, Squirrel takes into account two events: i) component deployment, and ii) component removal. Whenever the application is modified, the system automatically recalculates the aggregated resources required by the application and compares it to the available resources in the execution environment. If the available resources are greater than those required by the application, the reconfiguration is accepted, else, the application model is refused and the reconfigurations are discarded.

3.4 Mapping component-model concepts to system-level abstractions

Squirrel defines a set of steps to map component-model concepts to system-level abstractions in a way that support resource management. During platform design/implementation, developers identify system abstractions that are suitable to represent each concept and implement the respective mappings. As a second step, resource management methods for each abstraction are implemented and evaluated. This evaluation is used to determine the management methods with lowest overhead for each pair of system abstraction and resource type. Later on, at deployment-time, the platform selects a component-to-system mapping using optimization techniques and the data obtained at design-time. In the rest of this section, we briefly explain each step.

As we have mentioned, components can be represented using different system abstractions. This requires *identifying possible mappings from components to system-level abstractions*. These mappings must respect the semantics of the component model, and must provide resource management capabilities. A key problem is that non-functional properties vary among mappings; hence, optimizations are often needed to make some mappings attractive. Additionally, an extensible design of the component platform – where it is easy to accommodate new mappings – facilitates their coexistence at runtime. The system abstractions SA that can represent a concept, along with the recommended optimizations for each abstraction, are defined in this step.

During the design/implementation of the platform it is necessary to *define methods to manage resources* for each pair of system abstraction and resource type. Developers must devise management methods for each mapping and identify the least costly. If we consider different abstractions and resource types, we can define the matrices M and C where $\forall sa \in SA, rt \in RT$ the values $M_{sa,rt}$ and $C_{sa,rt}$ indicate the method that minimizes the cost of managing the resource rt when the abstraction sa is used to represent a component. We make two assumptions about the resource management mechanisms: i) mechanisms are always composable if they manage different resource types, and ii) the costs of any pair of management mechanisms are independent.

At deployment-time, *the platform selects the mapping* to use for each component in the application. To do so, it uses the information contained in the matrices M and C, the set of possible optimizations for each mapping, and the resource requirements of the application. At this stage, the only data needed regarding resource requirements is the type of resource. Notice that, although we only use a single cost matrix that contains the overhead of each management mechanism, it is possible to generalize the approach to handle multi-objective optimizations.

Others refinement to evaluate the cost of a mapping are possible. For instance, we can consider the cost of using a specific binding to connect two components that use a given mapping. Finally, there are many optimization methods that can compute the mappings, we do not propose any particular method in the approach. However, the results shown in section 5 suggest that very simple heuristics can lead to good performance.

4. REFERENCE IMPLEMENTATION

Squirrel's reference implementation exploits the models at runtime approach and provides resource-awareness capabilities to the Kevoree component framework [6]. Models at runtime denotes model-driven approaches to tame the complexity of dynamic adaptation [16]. A "model at runtime" is a reflection model that can be decoupled from the application and then automatically resynchronized. Models can manage not only the application's structural information, but can also be populated with other information, such as runtime monitoring data.

Kevoree is a component framework for distributed systems that builds and maintains a structural model of the system, following the models@run.time paradigm. Kevoree is mainly used because: (i) it presents a snapshot of the distributed system, and (ii) it provides a language to drive the reconfigurations. *Component* and *Channel* are two of the concepts used in Kevoree models. The former represents software units that provide the business value. The latter, with the same role as connectors, are in charge of inter-component communication. Channels encapsulate communication semantics (e.g., synchronous, unicast).

4.1 Containers for CPU and I/O reservation

Our implementation leverages resource-reservation mechanisms at the system-level to provide containers for CPU and I/O reservation. More specifically, it maps the concept of *container* onto a *cgroup*. Both *containers* and *Kevoree components* are hierarchical structures that are easy to map onto cgroups' hierarchy. Indeed, a container deploys components and a component runs threads. To configure the container we setup a hierarchy of cgroups using the following rules:

1. The Kevoree framework is started under a cgroup, using a fixed amount of resources that will be divided among the system's components.

2. Each component gets a new resource-aware container, also represented by a cgroup. The component's contract is translated into a slice S_c of the initial resource allotment, and the result is passed to the cgroup as configuration parameters. A slice represents the resources the component has available.

3. Since the scheduling unit for cgroups is a thread, we assign the component's threads to the cgroup to enforce resource reservation.

This scheme is used for CPU, I/O throughput and network bandwidth. Each type of resource requires a different *container type*. Figure 2 shows an example using cgroups to reserve CPU for a system with two components. In the tree, every edge is labeled with the cgroup's CPU slice. A slice is set for Kevoree, while unmanaged apps are maintained in separate containers. Applying rule 2, CPU slices are assigned to *Component 1* and *Component 2* according to their resource contracts. Following rule 3, every thread in *Component1* receives 33% of the component's slice.

4.2 Containers for Memory reservation

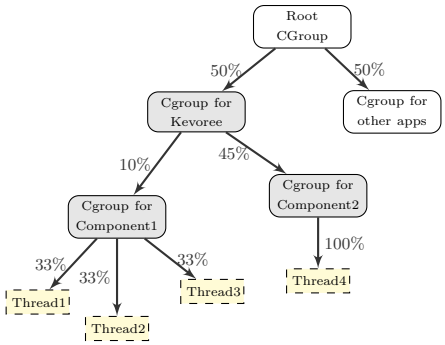

Figure 2: Reserving CPU by mapping components to cgroups

Memory reservation poses a unique challenge. Although there is a cgroup-hierarchy for memory, it is not well suited for shared memory because the subsystem cannot precisely account for the consumption of each thread. As a result, if we use cgroups to deal with memory, accounting would depend on the behavior of the garbage collector, which is hard to predict. That makes cgroups inadequate to check or enforce component contracts in a single JVM process. We have devised two mechanisms to serve as memory containers. In the first mechanism, all containers exist in a single process and resource limits are enforced by leveraging previous approaches that use bytecode instrumentation to account for consumption. The second mechanism isolates components into new processes and then uses cgroups. The rest of the section describes both solutions.

4.2.1 Memory management based on monitoring

A memory reservation container ensures that its components have access to the memory they require. Memory requests should only fail if a component violates its contract. Implementing such a container is simple if memory monitoring is available at the application-level and memory requests are intercepted. We use Scapegoat [8] for memory consumption monitoring by defining multiple memory-aware containers within a single JVM. In short, each container registers its component in ScapeGoat and receives a notification if a component violates its contract. Unfortunately, this introduces CPU and memory overhead for each component. The main advantages are portability and simplicity.

4.2.2 Isolation of components in separate processes

The second approach maps each container onto a separate process. Reservation is achieved using cgroups as described in Section 4.1. To do so, we start from an extended deployment model as shown in Figure 1. The model is then transformed using the following rules:

1. Component isolation: each *set of components* with a shared memory contract is isolated in a separate *JVM node* within the same *physical device*.

2. Channel adjustment: channels that connect isolated components are updated to reflect the semantics of the source model. This includes changing the channel

type and modifying some of its properties.

Afterward, the resulting model can be deployed.

Runtime initialization through cloning.

The approach to memory reservation based on isolation deploys each *set of components* into separate processes. This involves two steps: creating new instances of the runtime, and deploying a *set of components* on each instance. To reduce deployment time, instead of starting processes from scratch, new instances are cloned from a base runtime. The base runtime is created offline. Both steps, base runtime creation and cloning, are based on CRIU.[1] This tool allows snapshoting a process and starting any number of clones from the snapshot. In essence this *forks* the process.

Channel for intra-node communication.

Channels are meant to send arbitrary POJO [2] structures from one component to another. When components are isolated into separate processes, a channel must (un)marshal the POJO using a representation suitable for inter-process communication. In practice, a channel must copy data at least twice, no matter what IPC [3] mechanism is used.

In this paper we propose a new channel for intra-node communication. It is based on a message queue built on top of shared memory using an alternative high-performance framework to serialize objects. Each channel is mapped to a shared-memory region that hosts a synchronized queue of messages. This region contains three sections: an array of blocks to store message chunks, a set of free blocks, and a circular queue in which an element points to a list of chunks. We use the procedure described in [20] to synchronize senders and receivers, but we also support broadcast semantics without unnecessary additional copies. Our approach copies the POJO from the sender's heap to shared-memory during data marshaling, then every receiver makes a copy from shared-memory. The implementation uses a high-performance marshaling framework [4] instead of the built-in mechanism to serialize objects with better performance.

5. EVALUATION

This section presents experiments that determine the performance of our reference implementation. We compare different design decisions presented in the previous section. The experiments include:

- Measuring the CPU and memory overhead introduced by *Scapegoat* and *component isolation*.
- Determining how isolation affects deployment time and to what extent process cloning and our high-performance IPC alleviate it.
- Evaluating the extent to which known high-performance IPC techniques reduce communication overhead due to component isolation.

We used the same hardware across all experiments: a laptop with a 2.90GHz Intel(R) i7-3520M processor, running Linux with a 64 bit kernel 3.13.5 and 8GiB of RAM. We used the Oracle HotSpot JVM v1.7.0-55, and Kevoree framework v5.0.1.

5.1 Evaluating performance overhead

In section 4.2, we describe two approaches for memory reservation: 1) Scapegoat [8], an instrumentation-based resource management container, and 2) using isolated processes with cgroups. To compare the overhead produced by these approaches, we devised use cases that contain two components that execute a test from the Dacapo Benchmark Suite [3]. This benchmark suite consists of a set of real world applications with non-trivial memory loads. These components run in parallel to simulate realistic conditions where components demand resources simultaneously. The use cases are executed with different settings, as follows:

1. Using cgroups to assign 50% of the CPU time to each component (no memory monitoring). Both components run on a single Kevoree instance with 2GiB maximum heap size. This is the baseline because there is no CPU nor memory overhead. Nevertheless, execution time is affected due to the CPU allotment. We call this setting *Memory unaware*.
2. Using *Scapegoat* to monitor per-component memory consumption and bounding CPU consumption to 50%. Again, both components run on the same Kevoree instance with 2GiB maximum heap size.
3. Isolating each component in a new process, allotting 256MiB of memory to each process, and bounding its CPU consumption to 50%.

To measure CPU overhead, we use the result reported by each *Dacapo test* and we keep the maximum value. Measuring memory overhead is more complex because the garbage collector hides the real consumption. We address this by approximating the consumption with the usage after each major garbage collection. As there are many collection cycles during the use case's execution, we define a scheme to aggregate the values: the consumption MC_i of every Kevoree instance is the maximum among all the usages reported after each collection, while the total consumption is defined as $\sum_{i \in Isolates} MC_i$.

Figure 3a depicts the CPU overhead for *Scapegoat* and *Isolating components*. The overhead of *Scapegoat* was expected because of the instrumentation. On average, it performs 2.27 times worse than *Memory Unaware*, which is consistent with [8]. In contrast, isolating components produces no appreciable CPU overhead for these use cases (small differences are likely due to environment fluctuations) because the components do not interact.

Both *ScapeGoat* and *Isolating components* cause memory overhead. As shown in Figure 3b, *ScapeGoat's* is higher than

[1] criu.org
[2] Plain Old Java Object
[3] Inter Process Communication
[4] https://github.com/RuedigerMoeller/fast-serialization

when using component isolation. *Isolating components*'s overhead is the result of JVM duplication and is 99% over baseline. Meanwhile, *ScapeGoat's* overhead is due to tagging objects with the identifier of the component that owns it. Tagging adds either a field and a finalization method to an object, or wraps the object with a *weak reference* held by the framework, resulting in memory overhead.

In synthesis, *isolating components* produces no CPU overhead, and low memory overhead in comparison to the performance of the same component model without resource management features.

5.2 Evaluating starting time

We compare three methods to deploy components: 1) in a single JVM (the baseline), 2) in isolated JVMs, starting processes from scratch, and 3) in isolated JVMs using CRIU to clone processes. We study the scalability of each method by deploying many components. To do so, we created a template architectural model with two component types: *Component A* runs in the *management runtime* and deploys a new architectural model with resource-aware contracts; and *Components B* records the *timestamp* after initialization is completed. The experiment is as follows:

1. Component A is deployed in the *management runtime*. Afterwards, it forces the deployment of a new model with components of type B.
2. After deployment, each component $c \in List_B$ sends A the timestamp T_c.
3. Component A collects $T_c - T_0, \forall c \in List_B$, where T_0 is the timestamp before deployment.

Figure 4 shows the results for a varying number of components. As expected, using plain Kevoree is faster than deploying with other methods. Leveraging isolation with CRIU's process cloning is 19.75 times worse than plain Kevoree, and this overhead grows with the number of components. This is because CRIU-based deployment still spawns new threads in order to clone and create new instances. However, using cloning instead of starting processes from scratch reduces the isolation overhead by a factor of 41.79.

5.3 Evaluating communication

We present two experiments that highlight how we mitigate the impact of isolation on communication performance. First, we use a micro-benchmark to compare the performance of a shared-memory based IPC queue and a socket-based IPC queue. Then we benchmark the performance gain of using a specialized serialization framework that uses POJO structures – a common way to encode the business logic in real-life applications.

We evaluate our channel by comparing it against TCP communication, which is a widely used IPC mechanism in Java. To measure latency and bandwidth, the metrics we use for comparison, we developed a Netpipe clone [19] for Java[5]. The clone is delivered with three protocols: 1) socket-based, 2) our channel, and 3) a protocol for *named pipes*. The first uses simple TCP sockets with synchronous operation in its implementation. The second requires two channels, configured to use a queue with 128 chunks of 512Kb, because our channel is unidirectional. Figure 5a shows the latency for messages shorter than 128 bytes. Memory-based communication outperforms NIO-sockets for all the values in the range. Likewise, figure 5b shows the throughput of both mechanisms. In general, memory-based communication behaves better than TCP sockets. Our approach outperforms sockets by an average of 652.36% for messages shorter than 512 bytes. Meanwhile, it behaves on average 46.26% better for messages between 1 Mb and 64 Mb, which is the range where the benefits of large copies surpass the disadvantages of having a synchronized queue.

Latency between Java isolates is affected by the time spent marshaling and unmarshaling messages. To evaluate the benefits of *fast serialization*, we designed a micro-benchmark that sends a POJO structure, 16 bytes long, back and forth a million times. [6] We then measure the effect of marshaling for different numbers of consumers. Figure 6 shows the results of using built-in serialization against *fast serialization* for up to 8 components. It depicts the results in *messages per second* because different serialization methods flatten the same POJO structure into buffers with varying sizes. The comparison includes two serialization and two IPC mechanisms. However, the effect of the IPC mechanism is low due to the fact that serialization dominates the execution times. As a result, curves with the same serialization mechanism are close no matter what IPC mechanism we use.

5.4 Synthesis and Threats to validity

We evaluated our implementation of Squirrel regarding three aspects: overall performance overhead, starting time and communication performance. We believe that the performance of our reference implementation is good enough to enable resource management, while not excessively affecting the application's behavior. Although some metrics exhibit high overhead, we think that the trade-off given the new features offered in Squirrel is worth considering. Memory overhead is the biggest concern. Nevertheless, isolating components within separate processes greatly reduces the memory overhead and eliminates CPU overhead in comparison to the instrumentation-based solution from Scapegoat.

A threat to validity of our experimental protocol is that we evaluate different features as orthogonal concerns. Our experiments do not study the impact of all of Squirrel's features together in a real scenario, although the assumption of orthogonality of each concern is reasonable, particularly because Squirrel mainly relies on the well tested cgroups to enable CPU and I/O reservation.

6. RELATED WORK

The Squirrel approach is related to i) component isolation for fine-grained resource management, ii) efficient communication between containers, and iii) efficient container initialization. We discuss the related works for these topics.

Component isolation for fine-grained resource man-

[5]Source code at: http://goo.gl/h7OVm5

[6]Source code at: http://goo.gl/FXuUxc

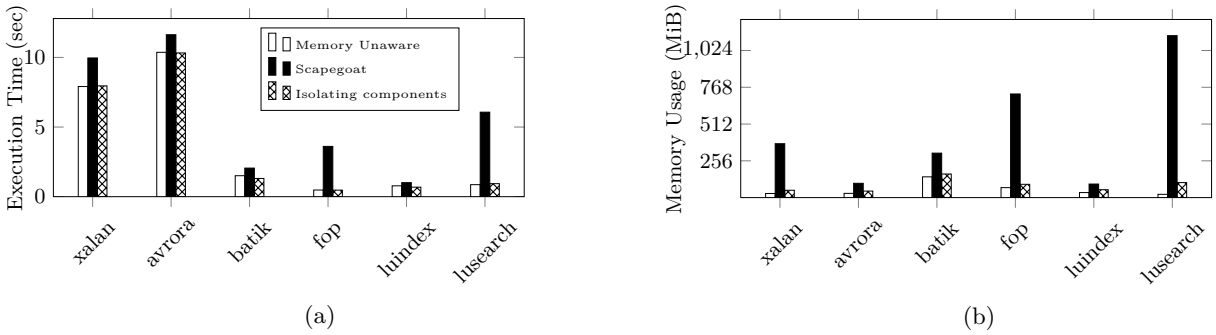

Figure 3: In 3a) and 3b), we show CPU and memory overhead, respectively, caused by resource management during the execution of Dacapo benchmarks.

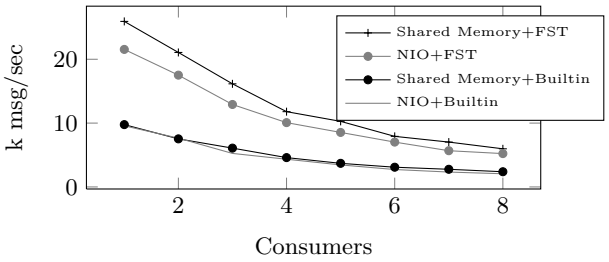

Figure 6: Communication throughput for different channels.

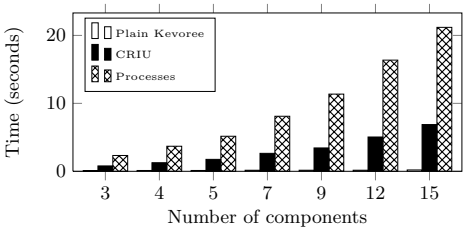

Figure 4: Average deployment time per component using different strategies

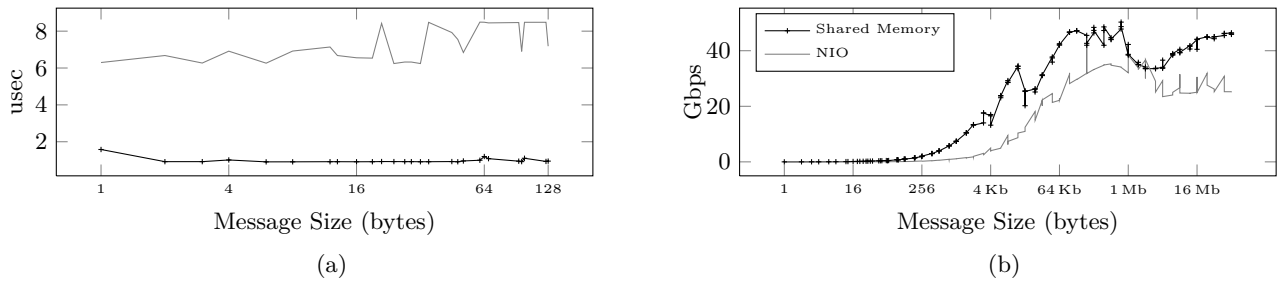

Figure 5: Comparing our queue against another IPC mechanism. Latency and bandwidth for a single receiver are shown in 5a) and 5b).

agement. In the Java world, several approaches describe the use of monitoring to achieve resource-aware programming [10, 14, 15]. In [2, 11] the authors propose mechanisms based on bytecode instrumentation to monitor CPU and memory consumption. Yet, instrumentation introduces high overhead and the monitoring framework contaminates the measurements because it is performed in the context of the monitored JVM. Geoffray et al. [7] introduce a modification to the garbage collector to achieve lightweight memory accounting for OSGi containers. Our work shows how we can automatically isolate components with resource contracts with low overhead through the execution of components on isolated system containers.

Efficient Process communication. In [20], the authors devise mechanisms for IPC based on building a queue in shared memory. As in our case, these solutions perform two data copies. Nonetheless, the authors do not consider the problem of sending messages to multiple targets, which is a concern for us. In Android, Binder is used to support IPC. Although it is a single-copy mechanism, its common

usage in applications still requires two copies due to marshaling. The limitations of the built-in serialization in Java are described by Bouchenak et al. [4]. Protocol buffers [21] provides an Interface Definition Language to generate RPC artifacts in different languages. Although the performance is good, it makes mandatory the use of an IDL, which is often a burden for engineers. We propose to use a combination of shared-memory-based channels and a high-performance serialization library to marshal POJOs.

Fast container initialization. A goal of MVM [5] is to provide fast initialization of Java Applications. By sharing the standard library among JVM instances, the authors reduce initialization time because the bootstrap classes are already loaded. However, the MVM is a non-trivial modification to a JVM and its current implementation does not support the latest version of the JDK [5]. Android [12, 18] uses a process, named Zygote, that loads the standard classes and is a sort of pre-initialized Dalvik VM. Zygote receives demands to create copies of itself that mutate into Dalvik VMs. We use an external tool, CRIU, to speed-up the creation of new JVMs because current JVM implementations do not provide built-in facilities for doing so.

7. CONCLUSION

This paper presents Squirrel, a framework that provides resource management capabilities to dynamic component-based frameworks. Squirrel proposes choosing component-to-system mappings at deployment time for better resource management. This strategy is performed automatically by checking the resource availability and transforming the application's structure to run the application on resource-aware containers. Containers describe how to map components to system abstractions allowing for different trade-offs in resource management.

We present an implementation of Squirrel that manages CPU, I/O and memory, and provide performance analyses and a comparison of different design decisions. The experiments show that choosing the right component-to-system mappings at deployment-time reduces CPU overhead and/or memory use. They also highlight that optimizing mappings is essential to reducing isolation and communication overhead to acceptable levels.

In the future, we envision augmenting the Squirrel framework with dynamic reasoning capabilities to automatically place and migrate components with resource contracts over a distributed architecture. Applying the concepts in this paper to other domains with strong safety and security concerns is of interest. The possibility of applying a pattern-based approach to increase an application's security without compromising its efficiency looks promising [17].

8. REFERENCES

[1] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[2] W. Binder. Portable and accurate sampling profiling for java. *Softw. Pract. Exper.*, 36(6):615–650, 2006.

[3] S. M. Blackburn and et al. The dacapo benchmarks: Java benchmarking development and analysis. OOPSLA '06, pages 169–190, NY, USA, 2006. ACM.

[4] S. Bouchenak, D. Hagimont, and N. De Palma. Techniques for implementing efficient java thread serialization. AICCSA'03, pages 14–18, 2003.

[5] G. Czajkowski and L. Daynàs. Multitasking without compromise: a virtual machine evolution. *ACM SIGPLAN Notices*, 47(4a):60–73, 2012.

[6] F. Fouquet, B. Morin, F. Fleurey, O. Barais, N. Plouzeau, and J.-M. Jezequel. A dynamic component model for cyber physical systems. In *Proc. of the 15th Symp. on Component Based Soft. Eng.*, CBSE '12, pages 135–144, New York, NY, 2012.

[7] N. Geoffray, G. Thomas, G. Muller, P. Parrend, S. Frenot, and B. Folliot. I-jvm: a java virtual machine for component isolation in osgi. DSN '09, pages 544–553, June 2009.

[8] I. Gonzalez-Herrera, J. Bourcier, E. Daubert, W. Rudametkin, O. Barais, F. Fouquet, and J.-M. Jézéquel. Scapegoat: an Adaptive monitoring framework for Component-based systems. In *Proc. of WICSA 2014*, Australie. IEEE/IFIP.

[9] R. Guerraoui and M. E. Fayad. Oo distributed programming is not distributed oo programming. *Communications of the ACM*, 42(4):101–104, 1999.

[10] F. Guidec, Y. Mahéo, and L. Courtrai. A java middleware platform for resource-aware distributed applications. ISPDC'03, pages 96–103, USA, 2003.

[11] J. Hulaas and W. Binder. Program transformations for light-weight cpu accounting and control in the jvm. *Higher Order Symbol. Comput.*, 21(1-2):119–146, June 2008.

[12] I. Kalkov, D. Franke, J. F. Schommer, and S. Kowalewski. A real-time extension to the android platform. JTRES '12, pages 105–114, NY, 2012.

[13] R. Krebs, A. Wert, and S. Kounev. Multi-Tenancy Performance Benchmark for Web Application Platforms. ICWE 2013. Springer-Verlag, July 2013.

[14] Y. Maurel, A. Bottaro, R. Kopetz, and K. Attouchi. Adaptive monitoring of end-user osgi-based home boxes. CBSE '12, pages 157–166, USA, 2012. ACM.

[15] L. Moreau and C. Queinnec. Resource aware programming. *ACM Trans. Program. Lang. Syst.*, 27(3):441–476, May 2005.

[16] B. Morin, O. Barais, G. Nain, and J.-M. Jezequel. Taming dynamically adaptive systems using models and aspects. In *In Proc.*, ICSE '09, pages 122–132, USA, 2009.

[17] B. Morin, T. Mouelhi, F. Fleurey, Y. L. Traon, O. Barais, and J.-M. Jézéquel. Security-driven model-based dynamic adaptation. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE*, pages 205–214. ACM, 2010.

[18] K. Nagata, Y. Nakamura, S. Nomura, and S. Yamaguchi. Measuring and improving application launching performance on android devices. CANDAR '13, pages 636–638, Washington, DC, USA, 2013.

[19] Q. O. Snell, A. R. Mikler, and J. L. Gustafson. Netpipe: A network protocol independent

performance evaluator. IASTED '96, 1996.

[20] R. C. Unrau and O. Krieger. Efficient sleep/wake-up protocols for user-level IPC. ICPP '98, 1998.

[21] K. Varda. Protocol buffers. http://code.google.com/apis/protocolbuffers/.