A Virtual Machine for Multi-Language Execution

Thorsten Brunklaus
Programming Systems Lab
Saarland University
Postfach 15 11 50
66041 Saarbrücken, Germany
brunklaus@ps.uni-sb.de

Leif Kornstaedt
Programming Systems Lab
Saarland University
Postfach 15 11 50
66041 Saarbrücken, Germany
kornstaedt@ps.uni-sb.de

ABSTRACT

This paper presents the architecture of a virtual machine designed specifically for the execution of multiple languages, which we call SEAM. The architecture consists of a number of generic components, usable by all languages, and of a number of interfaces for which implementations have to be provided by language implementors. Our contribution is the identification of the generic services and the clean design for the parameterization over their language-specific aspects. The goal of SEAM is to provide both for ample reuse and simple language implementation, concerning both compilers and runtime[1] components, and to be a platform for language interoperation. We have implemented a prototype version of SEAM and validated it with two language implementations. We present a full running implementation of Alice and a naïve implementation of a Java Virtual Machine running on SEAM. The paper presents first implementation effort and performance results for the prototype.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture; D.2.12 [**Software Engineering**]: Interoperability

General Terms

Design, Languages

Keywords

Virtual Machines, Interoperability, Pickling

1. INTRODUCTION

Practice of programming language implementation has in the last years shown an increasing interest in reusable virtual machine technology. Language implementors write compilers that target existing virtual machines, for example for MLj [2] or Active Oberon [9]. Virtual machine providers design and standardize virtual machines as targets for language implementors; examples are the Java Virtual Machine (JVM) [15] or Microsoft's .NET Common Language Runtime (CLR) [30].

The reasons for this trend are obvious. Language implementors want to spare themselves the effort to reimplement runtime services like memory management or operating system interfacing. Targeting an existing runtime environment yields easy reuse of available libraries, and a path to interoperation with other languages. From the perspective of the virtual machine provider, a reusable runtime environment opens up a market for tools, such as debuggers, profilers, or integrated development environments.

Performance and elegance of existing approaches have remained unsatisfying, however—either because the virtual machines have not been designed for generality, or because they remained inadequate for some language paradigms [5, 3, 33, 13, 6]. Criticism includes missing support for tail calls, overhead of defining data as classes and representing it as objects, or restrictive type systems of code verifiers.

This paper presents SEAM (the Simple Extensible Abstract Machine), a virtual machine designed specifically for the execution of multiple languages. Our goal was to provide a virtual machine for language implementors who want to reuse technology, have a common ground on which to interoperate with other languages, and nevertheless implement their language faithfully—without having to extend or modify it to make it match with the virtual machine. SEAM is an attempt to reconcile the four seemingly contradicting goals of simplicity of a language's compiler, ease of implementation of a language's runtime, ease of interoperability with other languages, and language specification fidelity.

The design of SEAM was driven by the key idea to assess every runtime service by its generality: If we can define a service with an interface generic enough to be adequate for very many languages, SEAM provides this service under that interface; where some service does not appear to have a generic incarnation, we instead parameterize over the service. As it turns out, a broad range of services can be provided, the effort for implementing a new language is greatly reduced, and language implementations gain in elegance. We have validated these claims with a complete implementation of Alice [1]—an extension of SML [19] providing for concur-

[1] We use the spelling "runtime" to stand for the runtime environment in which programs are executed, as opposed to "run-time" for the things that happen while programs run.

rency, data-flow synchronization and laziness—, and with a prototype implementation of a JVM on top of SEAM.

This paper is structured as follows. First, Section 2 presents SEAM's architecture, describing the design of the generic components and of the parameterization interfaces. Section 3 introduces generic synchronization support. Section 4 details how language implements can target SEAM. We address interoperability between languages in Section 5. Section 6 demonstrates the expressivity of SEAM's architecture with the design of a generic persistence facility for SEAM languages. We present and evaluate our prototype implementation in Section 7. Section 8 concludes the paper with an outlook on future work.

2. VIRTUAL MACHINE ARCHITECTURE

A virtual machine, or VM, consists of a number of components. The *store* provides a model of data representation and memory management. The *scheduler* coordinates the execution of concurrent threads. The *execution unit* actually executes code. The *I/O subsystem* abstracts away the interface to the operating system's input and output channels.

In traditional virtual machines designed for a single language, the design of these components is tuned to the language in question. For example, a virtual machine for an object-oriented language would represent data structures in the store as objects; object types (class definitions) then drive services such as garbage collection. The scheduler would directly implement the concurrency and synchronization semantics of the language. The execution unit would only know code representations for the particular language, for example a byte code with opcodes implementing the elementary language concepts, and so forth.

SEAM differs from this approach, in that it strictly separated language-independent services and language-specific bindings to those services. Where possible, SEAM provides generic components valid for all languages. Where this is not feasible, SEAM is parameterized, making language-specific implementations of those components possible. The following sections describe the realization of each of the abovementioned components in SEAM in turn.

2.1 Store

SEAM's abstract store implements a simple data graph: The basic node types are integers, *chunks* (raw byte data of arbitrary length) and *blocks* consisting of a number of ordered directed edges to other nodes. The data graph is an arbitrary directed graph with sharing and cycles. When configurable allocation thresholds are reached, the store sets a flag in SEAM's global status register to request that its *garbage collection* operation be invoked. Garbage collection frees memory taken up by unreachable nodes in the graph, given a *root set* of nodes in the graph. To accomodate for advanced garbage collection techniques, the interface distinguishes between initializing and reconnecting edges (which are different operations in the presence of, for instance, a generational garbage collector).

The data graph has been designed to be as simple as possible: By conceptually separating the abstract store and the representation of language data structures, we avoid premature commitment to any particular language. Instead, language implementors model language data structures in terms of the data graph. Also, all control structures used within the virtual machine (task stacks, threads, and the scheduler's queue of runnable threads) need to be encoded in the data graph. To enable discriminating between various internal control structures and user-defined data structures, all blocks are tagged by integer labels, of which some will be reserved in latter sections.

Related Work. Other virtual machines represent specialized and complex data in their store. In Mozart [31, 17] for instance, the store not only knows about Oz's rich data structures, but also about threads, task stacks, and computation spaces [24]. Mozart's store is generic in that its foreign function interface allows the implementation of user-defined data types (so-called *extensions*) in C++ [16], for which specialized garbage-collection and cloning routines have to be specified. Extensions are considerably less efficient than Mozart's built-in data structures and as such are no valid basis to support other languages on the Mozart VM.

The JVM and the .NET CLR represent objects in their store. The garbage collector has to know about class definitions for typing nodes in the store, and thus has to know about static and instance fields and inheritance. Experience has shown that an object-oriented store is not suitable for efficient implementation of functional languages, as it imposes too large an overhead for the numerous small allocations typically performed by functional programs [5, 3, 33, 13].

2.2 Scheduler

SEAM supports concurrent execution of multiple *threads* in the traditional way: A thread is a control structure that maintains a stack of activation records or *frames*. Each frame corresponds to a task to be executed; execution of a task can cause the frame to be modified or popped or new frames to be pushed. The *scheduler* coordinates execution of multiple threads, which it maintains in a queue. When the scheduler becomes active, it fetches a thread from the queue and passes it to the *worker*. The worker obtains the topmost frame from the thread's stack and executes it (see Section 2.3 for details). The worker returns control to the scheduler when a *preemption condition* is signalled, for which the worker periodically checks. The preemption condition is met when one of the flags in the global status register is set. A timer periodically sets a flag in the status register to implement preemptive scheduling (*time slicing*). When control returns to the scheduler, the preempted thread is enqueued again, the actions requested through the flags of the global status register are performed and the next thread is dequeued.

2.3 Execution Unit

As illustrated by the downfall of UNCOL [27], it is next to impossible to design a single intermediate language to accomodate all programming languages. As a consequence, we do not define a concrete execution unit in SEAM. Instead, all control structures interpreted by the execution unit are parameterized over the actual execution unit. SEAM defines an abstract *task manager* interface, which comprises the neces-

sary functions for interpreting stack frames: This includes executing a task, handling exceptions (or raising them to the next stack frame), and clearing no-longer-needed references to store nodes from stack frames prior to garbage collection. We chose per-frame parameterization (instead of per-thread): Stack frames store a reference to the associated task manager in their first slot; operations on stack frames retrieve the actual task manager and delegate to it. This implies that the computation within a single thread can be carried out by several task managers in conjunction.

Accordingly, we need a generic way to create stack frames, as a function application does (or a procedure call, or a method invocation). SEAM therefore defines an abstract *task creator* interface to perform stack frame creation. A first-class computation (a function closure, procedure, or method) is a pair of code and environment. In our representation, it is a block whose first subnode stores the code, while the remaining subnodes store the environment. Code in turn is a block whose first subnode is the task creator implementation, while the remaining subnodes are the code representation as defined by the task creator. A global register bank provides for passing arguments. We obtain a generic way to execute a task, regardless of the language in which it is implemented.

The downside of parameterizing stack frames and first-class computations is the extra indirection (virtual call) this implies. This overhead can be eliminated, however, when a language implementor knows that the same task manager/ task creator is associated with the caller and the callee in an invocation ("intra-language invocation"). In practice, language implementors can then make performance gracefully degrading with increasing use of cross-language calls. One implementation that does this is described in Section 7.2.

Related Work. Traditional virtual machines define a single worker and a single type of frame. The Mozart VM only knows about frames corresponding to Oz byte code; special tasks as for calling built-ins, releasing locks or handling exceptions are encoded by internal instructions camouflaged in a look-alike Oz byte code frame. The .NET CLR defines a single code representation, called IL (Intermediate Language), which—although much more expressive than JVM byte code—is very much influenced by the object-oriented paradigm as implemented by all of the Microsoft languages C# [29], Visual Basic.NET, and Managed C++.

Vmgen [8], despite its name, generates just an interpreter, and can be used to complement our approach—to actually generate a task manager implementation. The XVM approach [10] discusses bridging between virtual and physical machines, and considers extensibility with regard to this bridging. The Virtual Virtual Machine, or VVM [21], is designed for safe dynamic reconfiguration of systems. Its understanding of extensibility is to lift virtual machine components to the application level.

2.4 I/O Subsystem

SEAM's input/output subsystem corresponds to what is more or less standard in virtual machines: It provides primitives for interacting with files, pipes, and sockets. When a thread cannot immediately perform an I/O operation due to data inavailability, its execution has to be temporarily suspended so that other threads can be executed until data becomes available.

This means that a thread can be in one of several states: We say it is *runnable* when it resides in the scheduler's thread queue. When it is being executed, it is *running*. While a thread is waiting until an I/O operation can be performed, it is *blocked*. When the last task has been popped from a thread's stack, the thread becomes *terminated*.

The scheduler periodically polls I/O channels to eventually make blocked threads runnable again. When the thread queue is empty and there are still blocked threads, the scheduler enters an *idle loop*, blocking the whole process until a thread can be made runnable again. If the thread queue is empty and no thread is waiting for I/O, the virtual machine terminates. On some operating systems, a separate system thread can be used to wait for I/O and signal data availability asynchronously, by setting a flag in the global status register to avoid polling.

The next section addresses a more general case of thread synchronization and reduces the I/O special case to it.

3. CONCURRENCY WITH TRANSIENTS

Concurrent systems need a synchronization and communication facility to enable cooperation between threads. SEAM provides for data-flow synchronization and laziness in the form of *transients*.

A transient is a placeholder for a value which is not yet known. When the value becomes known, the transient is replaced by the value. Computations that find a transient in place of a value block until the value becomes known. Transients come in several flavors, of which we describe two in this paper. A *future* is a transient which can be explicitly replaced by its value. Every future stores a queue of blocked threads to reschedule when it is replaced. A *by-need* is a transient associated with a first-class computation (see Section 2.3). When a thread requires the value of a by-need, the by-need turns into a future (on which the requesting thread blocks), while simultaneously a new runnable thread is created with two tasks on its stack: one to explicitly replace the future, and on top of it a task created from the first-class computation. In other words, the first-class computation is executed at most once. By-needs are used to implement lazy (that is, demand-driven) computations.[2] Further kinds of transients are supported by the implementation, which we will not present here.

The introduction of transients affects the virtual machine's components as follows. The **store** represents transients as blocks whose label denotes the kind of transient. Furthermore, a reserved label, internal to the store, is used to represent *references*. A replaced transient updates its label in place to become a reference, and stores an edge to the node the transient was replaced with. The garbage collector elim-

[2]A variant of by-needs is imaginable which would simply push the two tasks on the requesting thread, to dispense with the overhead of explicit thread creation. The difference is observable only in languages which have a notion of a first-class thread with identity.

inates references. The store has two APIs: One to use on nodes that can be transients, which handles replaced transients transparently (by following reference chains) and tests for transients, and another to use on nodes where it is statically known that they can never be transients (where preconditions assert that the nodes effectively are no transients or references). Thus, internal control structures and languages that do not support transients (or only in special contexts) are not penalized by SEAM's support for transients. This design makes performance gracefully degrading with increasing use of transients.

The **scheduler** is extended to the handle different kinds of transients when they are requested by computations. The thread needs to be removed from the runnable queue, and maybe a new thread is created.

The task manager implementations making up the **execution unit** need to use the corresponding store API, depending on whether they wish to handle transients or not, and to notify the scheduler of transient requests.

The **I/O subsystem** actually builds on transients to handle blocking on I/O. This implies a single mechanism to block and wake up threads, which has a straightforward and elegant formulation.

Related Work. The transients supported by SEAM are the implementation basis for futures as supported by Alice, and as explored in a formal calculus [20]. Transients can be used to express data-flow synchronization, logic variables as found in logic programming languages such as Oz [26], and laziness as found in Haskell [11]. By-needs are closely related to *thunks* as used in the implementation of lazy functional languages.

4. LANGUAGE IMPLEMENTATION

A language implementor targeting SEAM reuses the generic infrastructure as shown in Figure 1, by providing the components which the infrastructure parameterizes over. This set of components makes up what we call a *language layer*. This section outlines what it means in practice to implement a language layer.

Data Representation. The store provides an expressive, but low-level model of data representation. Language implementors need a high-level interface to their data structures for conciseness and for making representation invariants explicit. The abstract store currently provides the data graph nodes in the form of classes and objects. A high-level interface to a specific language's data structures would specialize the store node classes to define data layout and high-level types for subnodes. Virtual functions can be simulated by assigning distinct integer labels to blocks and using them for dispatch.

Code Representation. SEAM makes no assumption as to *how* code is represented. Language implementors must therefore define their own code representation. If code is to be subject to garbage collection, it must be encapsulated within store nodes.[3] For instance, byte code and native code can be wrapped into a store chunk.[4]

Task Manager. To actually execute code, language implementors implement the task manager interface. They first define a stack frame layout. Arbitrary layouts are possible, to represent an activation record as required by the language (the implicit first subnode storing the actual task manager is hidden by SEAM's interface). Stack frames can be of arbitrary size. An interpreter has to be implemented to operate on code representation and stack frame layout.

Task Creator. To actually allocate a stack frame with the layout defined above, the task creator interface has to be implemented. First-class computations as defined by SEAM store the task creator along with the represented computation's environment, whose layout can to be chosen by the language implementor.

Language Primitives. Finally, the language implementor provides the primitives required by the language and its accompanying standard libraries. One possibility is to represent each primitive by an instruction in the code (or, in the case of native code, to inline or directly call it, for instance, as a C function). The second alternative is to wrap each primitives up as a first-class computation with its individual task creator and task manager. This case is handled by SEAM's generic interoperability features discussed in the next section.

5. INTEROPERABILITY

Both for language implementors and for language users, interoperability is a desirable feature. Language implementors can use interoperability to reuse existing technology, originally intended for another language, to facilitate their implementation task. Language users benefit from interoperability because it enables them to write mixed-language applications: They can "use the right tool for each task".

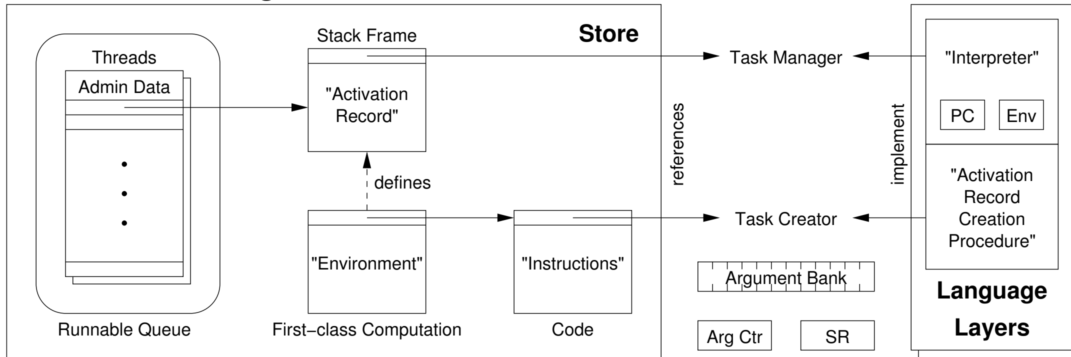
A prerequisite of language interoperation is a common understanding of the idioms with which languages are meant to interoperate. The more idioms are shared between language implementations, the simpler interoperability becomes for the language implementor—provided the effort required to abide to those idioms is low.

SEAM provides a common unterstanding of a number of idioms that programming language designers seem to widely agree upon. In particular, SEAM defines generic first-class computations and their invocation by means of the task creator and task manager interfaces. SEAM's global argument register bank provides a uniform way to pass arguments and return values. An argument count register enables variable-argument functions as featured by Lisp or C, as well as conversion between one-argument and multiple-argument functions as typically performed in ML implementations. A common light-weight concurrency model along with generic synchronization primitives in the form of transients enables mixed-language concurrent applications, in-

[3] An alternative is to store code outside the garbage-collected store. Our current implementation provides weak maps [7], whose finalization mechanism can then be used to explicitly free memory taken up by code.

[4] If the store uses a relocating garbage collector, then of course native code must be position-independent.

Figure 1: Virtual Machine Architecture Overview



cluding automatic data-flow synchronization and laziness. SEAM's I/O subsystem makes it possible to share input and output channels. All this is based on a single store which represents data specific to all languages in a unified data graph.

Of course, this is just the technical side of interoperation. On the semantic side, language implementors have to take care of compatibility themselves, such as matching number and order of arguments when performing cross-language calls. It is the responsibility of language implementors to provide for data representation conversion and marshaling. As such, real interoperability still implies pairwise matching up of languages. Improving this situation is the subject of future work.

From our experiments so far, this interoperability concept also forms a valid basis on which to build a foreign-function (resp. native method) interface. This is further discussed in Section 7.

Related Work. .NET's Common Language Specification (CLS) [18] defines a very rich set of idioms that languages are encouraged to share. The standard Microsoft set for language implementations to call themselves CLS-compliant is very high—so high that it incurs a performance cost for languages that do not match the CLS closely enough. It can be argued that this cost only has to be paid at interoperation boundaries, but actual applications may want to cross these boundaries with arbitrarily fine granularity. Moreover, distinguishing inter- and intra-language calls increases the complexity of the language's compiler, which in the extreme has to support two distinct data representations and calling conventions and has to convert between them. A picky observer of the evolution from Visual Basic 6 to Visual Basic.NET could rather regard it as a concealed effort to make Visual Basic CLS-compliant.

6. PICKLING AS A GENERIC SERVICE

SEAM's parameterization over task managers is very expressive. To demonstrate this, we present the generic pickling and unpickling service SEAM provides—in a way that makes them configurable by language implementors. Providing services like pickling is interesting for turning SEAM into a *middleware* instead of just a virtual machine. Note that pickling is a basis for other openness services, like components and distribution, as exemplified by Mozart [31].

This section is structured as follows. After introducing terminology for pickling, we identify the different types of nodes that require treatment during pickling. We outline the basic process, then incrementally refine it to handle the different types of nodes.

Terminology. *Pickling* (also called *serialization*) is the process of externalizing a data graph to a sequence of bytes, called a *pickle*. Pickles are used for persistence (when stored on a file) or for communicating data between distributed computation sites (when transferred to another process). The process of reconstructing store nodes from a pickle is called *unpickling* (or *deserialization*). The reconstructed graph must be isomorphic to the graph as it existed at time of pickling, in particular, have the same sharing between the nodes contained in the pickle (including cycles).

Node Types. Different types of data require different treatments during pickling. We can identify the following four per-node pickling semantics:

Cloneable Nodes. Some nodes need to be cloned, for instance if they represent stateless data. Stateful data structures can be cloned as well, resulting in a snapshot of the state the data structure was in at the time of pickling.

Translated Nodes. Other types of nodes need to be transformed, for instance into a platform-independent representation. Conversely, unpickling may construct a platform-specific representation.

Resource Nodes. Still other types of nodes have no meaningful external representation, for instance, if they represent resources pertinent to the local computation site (such as operating system handles to open files or to graphical windows). Such nodes cannot be pickled.

Transient Nodes. Last but not least, transients have to be considered. Pickling a future may not make sense, since a clone constructed by unpickling would not have an associated computation to eventually replace it.

For pickling to be a generic service, it has to distinguish all of these node types, and language implementors must be able to associate the appropriate pickling semantics with their data structures.

Basic Process. The basic pickling procedure for clonable nodes consists of a depth-first traversal of the data graph, starting from a root node, and producing a byte sequence from which an isomorphic graph may be reconstructed. SEAM provides a pickler that builds on SEAM's infrastructure. We define a task manager to perform the traversal. Its task is invoked with the output byte buffer as argument, and the activation record stores a reference to the node to visit. In the process, nodes that are encountered more than once need to be recognized. Remembering nodes requires a mapping from nodes to offsets in the serialized representation. This mapping is therefore supplied as an argument to the task. The task then consists of writing a serialized representation of the node (its kind, label, and number of edges) and of pushing new tasks for the subnodes—unless the node is already contained in the mapping, in which case a reference to that node is written.

The unpickling process is similar. We define a task manager to perform a single unpickling task. An unpickling task consists of a node that requires its edges to be initialized, and the number of the edge to initialize next. Its arguments are the input byte sequence and a mapping of offsets in the byte sequence to nodes, to reconstruct sharing. The task manager reads a node specification from the input byte sequence and constructs the corresponding node, then initializes the parent's edge specified in the stack frame, updating the stack frame to advance to the parent's next edge (if any) and pushing new tasks for initializing subnodes.

Expressing the pickler and unpickler as task managers makes them concurrent (that is, preemptable). Their state is automatically saved along with the task stack and arguments when a thread switch occurs.

Note that if code is modeled as ordinary store nodes, it automatically becomes part of pickles of higher-order data structures.

Translation. The process outlined above deals only with cloneable data. We now extend this process to deal with translated nodes.

Externalized data is often intended for cross-platform information interchange, while data in the store is intended for computations. These differing goals require different representations. We call the externalized representation an *abstract representation*, as opposed to the *concrete representation* used for computation. When a concrete representation is to be externalized, it needs to be abstracted; conversely, when an abstract representation is read in, it needs to be instantiated. Each language defines its own abstract and concrete representations, and we want pickles to be able to represent mixed-language data graphs. For this reason, the abstraction and instantiation functions need to have pointwise definitions—that is, one definition for each type in every language.

These concepts are introduced in the design by defining concrete and abstract representations as well as abstraction and instantiation functions in SEAM. Concrete and abstract representations are store data structures. For most nodes, abstraction and instantiation are the identity function. We introduce one new specialized store block to each kind of representation. *Instantiated nodes* are nodes on which the abstraction function is not the identity, but user-supplied. Conversely, *abstract nodes* are the nodes on which the instantiation function is user-supplied. Both instantiated and concrete nodes are store blocks with reserved integer labels. The first subnode of an instantiated node is a handler that can be queried for the abstract representation, as a function of the other subnodes. The result can be any store node, but will typically be an abstract node. The first subnode of an abstract node is the type identifier for an instantiation function, which expects the abstract node as argument and returns the concrete representation—typically a concrete node.

Operationally, these nodes are treated as follows. When the pickler encounters an instantiated node, it queries the handler for the abstract representation and processes it in place of the original node. The unpickler is parameterized over a mapping from type identifiers to instantiation functions. When it has reconstructed an abstract node from the pickle, it applies the instantiation function specified by the node's type identifier, and inserts the returned concrete representation into the constructed graph instead of the abstract node. The mapping table is extended with language-specific type identifiers when the corresponding language layer is initialized.

One typical data structure for which there exist both abstract and concrete representations is code. The abstract representation might be a byte code, while the concrete representation might be native code. This offers a simple scheme to implement demand-driven run-time compilation: The instantiation function for code can return a by-need, which, when requested, causes the run-time compiler to be run. Representing code as instantiated nodes is the only modification we need to make to the generic components in order to make abstraction work.

Resources. Nodes that represent resources are not allowed to be pickled, in other words, they have no abstract representation. It is logical, therefore, to represent resource nodes as instantiated nodes with a handler that returns an error value instead of a store node. The pickler checks for this error value and, if found, raises an exception.

Transients. Since we do not want transients to be written to a pickle, we simply let the pickler request them. This means that the pickling thread can block in the middle of writing a pickle. This is fine since the pickler is preemptable (the state is saved due to the explicit representation of pickling tasks on the thread's stack). We say that pickling is *incremental*—when the pickling thread is woken up, it continues with the node that the transient was replaced with, and needs not start again from scratch.

Related Work. Pickling is related to serialization as found in other programming systems, for instance in the Java 2 Platform [28]. In contrast to our approach, Java's approach does not support including code in serialized data; deserialization instead links to classes referenced by identifiers in the serialized representation. Java's serialization is highly configurable and can handle all of the node kinds we have

described above, although in a less principled manner. An application of Java serialization to achieve similar semantics as Alice's pickling, which corresponds to what is described above, has been implemented for DML [25]. Java's serialization specification is a lot more complex, however, than the one we propose.

7. PROTOTYPE

In this section, we present our prototype implementation of SEAM in C++ and the language layers built on top of it. Section 7.1 discusses implementation aspects of the generic virtual machine services. The design is validated by two language layers built on top of it: A complete implementation of the Alice programming language is described in Section 7.2 and a low-effort implementation of a Java Virtual Machine is presented in Section 7.3. We conclude with a performance evaluation in Section 7.4.

7.1 Generic Services

Store. The store implements the data graph described in Section 2.1. Our tagging scheme and heap layout are inspired by that described by Leroy [14]. We use *tagged pointers* as the type of store nodes, with one tag bit serving to distinguish between 31-bit integers and pointers to heap-allocated data structures. Pointers have an additional tag bit, so we can distinguish pointers to transients from pointers to blocks or chunks without requiring a memory access. This improves efficiency of transient tests and of dereferencing.

In addition, the store provides two built-in collection types: *weak maps* [7], which provide for finalization, and generic maps, whose keys can be arbitrary (non-transient) store nodes. Both require special handling in the garbage collector. Nodes in weak maps are finalized when the garbage collector determines that the last reference (excluding that from the weak map) has become garbage. Generic maps are based on hashing, using token equality and *address-of* as hash function. Thus, they have to be rehashed after every garbage collection.

Our store implements a generational copying garbage collector with Cheney-style scanning and Ungar remembered sets [34]. Scanning eliminates reference nodes, that is, transients that have already been replaced.

Execution Unit. The task manager and task creator interfaces are defined as abstract C++ classes. They define a small number of virtual functions that language implementors must provide in derived classes. Pointers to task manager and task creator objects are represented as integer nodes in the store, using casts.

Pickling. As described in Section 6, the pickler maintains a mapping from visited nodes to offsets in the emitted pickle. A well-known technique is to destructively mark nodes as visited in the heap, similar to *forward pointers* as used by garbage collectors. This technique is not applicable for incremental pickling, which can be interrupted to run other threads, which may operate on the same data structures. Our implementation therefore represents this mapping as a generic map. Checking for sharing is then constant time, which makes pickling linear in the number of nodes despite its concurrency.

Table 1: Component Size Overview

Component	#LOC	Subcomponent	#LOC
Generic	11500	Store	3500
		Pickling	1650
Alice	12300	Run-time Compiler	3400
		Interpreter	1100
		Primitives	5750
Java	8500	Primitives	1500

Implementation Effort. The implementation effort of our prototype system, measured in lines of C++ code, is shown in Table 1. The left column shows the total size of the implementation layers, while the right column details how much of the code pertains to which subcomponent. The numbers show that our implementation is very compact.

Related work. Java serialization, too, is concurrent and needs to represent visited nodes explicitly. Java provides a method to compute a first-class hash value for each object, sometimes implemented by representing it in the same field as the object's lock (which would then be an index into an array of locks). Where Java sacrifices space for every hashed object once and for all, we only pay the price for hashing during the lifetime of the generic map.

7.2 Alice-on-SEAM

Alice is an extension of SML for concurrent open programming [1]. It features transients in the form of lazy futures (that is, SEAM's by-needs), concurrent futures (SEAM futures), holes for top-down construction of data structures and failed futures for signaling failed computations, by raising exceptions upon request. Alice has a component system which builds on pickling: A component is a pickled triple of import specification, export specification, and a function that, when applied, evaluates the declarations in the body of the component. The component manager, which is itself implemented in Alice, type-checks and links components, and does so lazily via the use of lazy futures. The component manager is bootstrapped by a *boot linker* provided in the Alice language layer. A stand-alone Alice application is a component that executes the application as a side-effect of its evaluation.

Transients are allowed in data structures of all Alice types, and the primitive operations of the language implicitly synchronize, such as pattern matching or procedure application. This is similar to Oz [26]. Accordingly, practically all of the Alice language layer uses the transient-aware store API.

The representation of Alice code is designed for compilation rather than interpretation. It has, in its abstract representation, the form of a direct acyclic graph (loops are translated to recursion) whose nodes are the "instructions". The code is in static single assignment form. We implemented two concrete representations and according task managers for code execution: The first concrete representation corresponds to the abstract representation, and the task manager interprets the instruction graph. The second one is native

code: The instantiation function returns a by-need, which, when requested, runs a run-time native code compiler. The compiler allocates registers using linear scan register allocation [22] and emits position-independent machine code into store chunks using GNU Lightning [4]. The corresponding task manager bridges between C++ and the generated native code. Run-time compilation recognizes intra-language calls of functions in the closure of the compiled code, and optimizes them to remove the indirection imposed by task creators. Native code performance is superior to the interpreter by about a factor of three.

Our Alice implementation provides a foreign function interface along the lines of what has been outlined in Section 5. A dedicated task creator and task manager handles the invocation of foreign functions. Foreign functions can be higher-order and call back into Alice. If a foreign function calls an Alice function in non-tail position, it needs to define a task manager to push a continuation implemented in C++.

7.3 Java-on-SEAM

The Java-on-SEAM implementation consists of three main components: the class file parser, the class loader and the byte code interpreter.

Class file parsing is implemented as an atomic operation. The class file is read into a chunk and a symbolic type representation is constructed from it. Used types are represented as symbolic references in the *constant pool*. We support only class files as specified by the JVM specification [15]; we intend to support pickled classes in the future.

Type loading is performed by the class loader, which invokes the class file parser to obtain and verify the symbolic type representation. Verificiation is not yet implemented and thus always succeeds. The type is *prepared*, that is, a concrete type is constructed from the symbolic representation, with a run-time constant pool consisting of concrete references.

The byte code interpreter executes Java byte code. Methods are first-class computations whose environment is the run-time constant pool. Method activation records are allocated on the task stacks provided by SEAM. Currently, the instructions are implemented naïvely following the JVM specification. Thus, every static field access, static method invocation and object creation has to acquire the class lock, and check wether the class object has yet been initialized. If not, the static initializer is run. Thread-reentrant locks are implemented as first-class values based on futures—waiting for a lock corresponds to SEAM's concept of blocking. Before a static method call, a special task manager is pushed to release the class lock when the method returns.

The JVM specification gives implementations some freedom in the timing of type loading and linking, but requires that each class or interface must be initialized on its first active use [32]. Our implementation is fully lazy in that loading and linking are postponed until the first active use of a type. To achieve this, preparation creates the references in the run-time constant pool as by-needs. When the by-need is requested by, say, a method invocation, this triggers method lookup, which in turn triggers (if not yet performed) class loading.

As the JVM specification does not impose any particular binary data layout, Java types and objects are straightforwardly mapped onto store blocks. We try to represent Java integral types as store integers. Since store integers are only 31 bits in size, long integers must be boxed in a store chunk. A compilation option for experiments represents Java's 32-bit integers as 31 bits, losing one bit. All floating point values are also boxed.

Table 1 again summarizes the implementation effort of the JVM language layer. We want to mention that it took only three person-weeks to complete.

7.4 Performance Evaluation

This section experimentally assesses the performance results of our current implementation. The performance results in this section were obtained on a Sony GRX316MP with a 1600MHz Pentium 4 processor running Windows XP. The system has 512MB of main memory.

We evaluate the system with benchmarks Scheidhauer used to evaluate the Mozart VM [23]. We run them on four different systems. Two of these are Alice-on-SEAM and Java-on-SEAM as presented above. The Alice compiler has a second backend, which emits Oz byte code for the Mozart VM [12].[5] Alice-on-Mozart becomes our third test system. Last but not least, we use the JVM shipped with Sun's Java 2 SDK 1.4.1 (named Java-on-SDK in the following). We compiled the Java benchmarks with `javac` from Sun's SDK. We ran Alice-on-SEAM with our simple run-time compiler, whose quality we deem to be roughly equivalent to optimized byte code. All other systems used interpreters, that is, the JIT-ter was turned off in Java-on-SDK. No other configuration or tuning was done.

We focus on three common cases: recursive computation with simple integer arithmetic, concurrent computation with data-flow synchronization, and symbolic computation. We report the minimum time obtained from eight runs of each benchmark on each system, with a preceeding dry run to hide startup effects such as lazy component resp. type linking. Table 2 shows the results. We emphasize that these are preliminary results.

Recursion. *fib*(31) and *tak*(24, 16, 8) are standard benchmarks. Java-on-SDK performs best: It is 3.8 resp. 1.6 times faster than Alice-on-Mozart. Java-on-SEAM performs worst, taking twice resp. four times as long as Alice-on-Mozart.

The reason for SEAM's bad performance is that these benchmarks actually exhibit a worst-case behaviour of the current implementation. We represent stacks as arrays of references to activation records. Like all data, these are allocated on the heap. In real applications, which perform computations between function applications, the generational garbage collector releases space consumed by obsolete stack frames early enough, sparing us any special treatment of stacks. In contrast, *fib* and *tak* suffer from heavy allocation

[5] Alice on Mozart performs roughly within 30 percent as well as Oz on Mozart.

Table 2: **Benchmark Results (in milliseconds)**

	fib	*tak*	*mkthread*	*concfib*	*nrev*	*deriv*
Alice-on-Mozart	1061	580	451	180	1341	390
Alice-on-SEAM	2203	1522	250	60	1652	551
Java-on-SDK	280	360	24045	10475	1792	1642
Java-on-SEAM	2123	2234	451	100	12307	4807

and frequent garbage collections: The youngest generation is rapidly filled, and this requires either a minor collection or more memory to be made available to the youngest generation. In both cases, the overall performance of the VM suffers.

We see several solutions. The Mozart VM circumvents this problem by using free lists for environments, which enable for explicit deallocation and reuse. Another solution would be to flatten task stacks, that is, represent the activation records inline instead of as separate nodes. To prevent frequent allocation of large blocks this would incur, stacks should then be handled specially in the store.

Concurrency. Languages intended for open programming such as Java and Alice need to support concurrency and synchronization efficiently. *mkthread* sequentially creates 100,000 threads executing the empty computation and waits for their termination. *concfib* computes the Fibonacci number of 20, creating a thread for every recursive call. This differs from *fib* in that a large number of threads are runnable at the same time. The Alice implementation exploits Alice's implicit data-flow synchronization, whereas the Java implementation uses the `Thread` methods `start` and `join`.

Alice-on-SEAM is two to three times faster than Alice-on-Mozart, and outperforms Java-on-SDK by more than one order of magnitude. To the best of our knowledge, Java-on-SDK does no longer support configurability of green and native threads; presumably those threads are heavy-weight system threads. Java-on-SEAM benefits from SEAM's light-weight threads.

Symbolic Computation. *nrev* allocates a list with 5000 elements and naïvely reverses this list using 'append'. *deriv* computes 30 times the 6th symbolic derivation of $(\frac{1}{x})^3$. Both benchmarks actually perform some work on algebraic data types between two recursion steps, thereby producing lots of temporary heap allocation. This can be considered the average case for many applications.

Alice-on-Mozart performs best, closely followed within 20 to 40 percent by Alice-on-SEAM. This result indicates that our design decision to not handle stacks specially is adequate. Moreover, it evidences that our SEAM prototype is competitive with a highly optimized virtual machine specifically designed for a single language.

Java-on-SDK performs worse than Alice-on-SEAM, ranging from 8 percent slower for *nrev* up to 3 times as slow for *deriv*: Object allocation is obviously expensive on the JVM. The bad performance of *deriv* indicates that virtual method invocation, the Java idiom corresponding the Alice's pattern matching, is expensive.

Large Applications. We have tested a large application on Alice-on-SEAM, namely the Alice compiler recompiling itself and the system libraries. Performance is competitive with Alice-on-Mozart.

8. FUTURE WORK

We have presented a the architecture of a generic virtual machine that is extensible to become a specialized runtime environment for many languages. We have demonstrated the validity of the approach by implementing two language layers on top of it, one for the functional concurrent language Alice, the other for a complete Java Virtual Machine.

We have identified a number of areas worthy of future research. For one, we will strive to improve the system's efficiency. Another goal is to improve interoperability. Pairwise matching-up of languages, as is still necessary now, can be improved by proposing a SEAM-based "common language specification", as Microsoft did for .NET. In particular, we would like to suggest a common representation for program components, to obviate the need for implementing language-specific component managers or class loaders, an to provide a generic way to establish references to program components implemented in different languages.

9. ACKNOWLEDGMENTS

SEAM was developed in a follow-up project to Mozart [31], and as such owes much to the numerous developers of and contributors to the Mozart VM. The authors would like to thank Ulrike Becker-Kornstaedt for her comments on this paper. The Alice language layer is a joint work of the whole Alice Project—we want to mention Andreas Rossberg in particular for his work on the compiler frontend and for the numerous discussions. Our thanks go to Christian Schulte for the distinction of abstract and concrete representations in pickling, and for having brought up the name STEAM (Simple Tiny Extensible Abstract Machine). Thanks also go to Guido Tack for his helpful comments on the paper.

10. REFERENCES

[1] The Alice programming language. Web Site at the Programming Systems Lab, Universität des Saarlandes, 2002. `http://www.ps.uni-sb.de/alice/`.

[2] N. Benton, A. Kennedy, and G. Russell. Compiling Standard ML to Java bytecodes. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP)*, pages 129–140, Baltimore, Maryland, Sept. 1999. ACM Press.

[3] P. Bertelsen. Compiling SML to Java bytecode. Master's thesis, Department of Information Technology, Technical University of Denmark, Jan. 1998.

[4] P. Bonzini. *GNU Lightning*, 1.0 edition, 2002. http://www.gnu.org/software/lightning/.

[5] P. Bothner. Kawa: Compiling Scheme to Java. In *Lisp Users Conference ("Lisp in the Mainstream")*, Berkeley, California, Nov. 1998.

[6] T. Dowd, F. Henderson, and P. Ross. Compiling Mercury to the .NET Common Language Runtime. In *Electronic Notes in Theoretical Computer Science*, volume 59.1, pages 70–85. Elsevier Science Publishers, 2001.

[7] R. K. Dybvig, C. Bruggemann, and D. Eby. Guardians in a generation-based garbage collector. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, pages 207–216, Albuquerque, New Mexico, June 1993. ACM Press.

[8] M. A. Ertl, D. Gregg, A. Krall, and B. Paysan. Vmgen—a generator of efficient virtual machine interpreters. *Software–Practice and Experience*, 32(3):265–294, 2002.

[9] J. Gutknecht. Active Oberon for .NET: An exercise in object model mapping. In *Electronic Notes in Theoretical Computer Science*, volume 59.1, pages 120–138. Elsevier Science Publishers, 2001.

[10] T. L. Harris. *Extensible Virtual Machines*. PhD thesis, Churchill College, University of Cambridge, Dec. 2001.

[11] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the programming language Haskell. In *SIGPLAN Notices*, volume 27(5), May 1992.

[12] L. Kornstaedt. Alice in the land of Oz—an interoperability-based implementation of a functional language on top of a relational language. In *Electronic Notes in Theoretical Computer Science*, volume 59.1, pages 18–33. Elsevier Science Publishers, 2001.

[13] T. X. Le. Berlioz: Compiling Oz to Java bytecode. Master's dissertation, National University of Singapore, 2001.

[14] X. Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report RT-0117, INRIA, Feb. 1990.

[15] T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Addison Wesley, 2nd edition, Apr. 1999.

[16] M. Mehl, T. Müller, C. Schulte, and R. Scheidhauer. *Interfacing to C and C++*, 1.2.4 edition, 1998. Mozart Online Documentation.

[17] M. Mehl, R. Scheidhauer, and C. Schulte. An abstract machine for Oz. In *Proceedings of PLILP'95, LNCS*, Utrecht, The Netherlands, 1995. Springer-Verlag.

[18] Microsoft. What is the Common Language Specification? Online Documentation, 2001. http://msdn.microsoft.com/library/en-us/cpguide/html/cpconwhatiscommonlanguagespecification.asp.

[19] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[20] J. Niehren, J. Schwinghammer, and G. Smolka. Concurrent computation in a lambda calculus with futures. Programming Systems Lab, Universität des Saarlandes, June 2002.

[21] I. Piumarta, B. Folliot, L. Seinturier, and C. Baillarguet. Highly configurable operating systems: The VVM approach. In *Proceedings of the 3rd ECOOP Workshop on Object-Orientation and Operating Systems (OOOSWS)*, Cannes, France, June 2000.

[22] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[23] R. Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Doctoral dissertation, Fachbereich Informatik, Universität des Saarlandes, Dec. 1998.

[24] C. Schulte. *Programming Constraint Services*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2000.

[25] D. Simon. An implementation of the programming language DML in Java: Runtime environment. Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, Feb. 2000.

[26] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, Vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[27] T. B. Steel. A first version of UNCOL. In *Proceedings of the Western Joint Computer Conference*, pages 371–377, 1961.

[28] Sun Microsystems, Inc. *Java 2 SDK Documentation: Object Serialization*, 1.4.1 edition, 2002. http://java.sun.com/j2se/1.4.1/docs/guide/serialization/.

[29] TC39/TG2. C# language specification. Technical report, ECMA, 2001. To appear as ECMA-334.

[30] TC39/TG2. Common Language Infrastructure (CLI). Technical report, ECMA, 2001. To appear as ECMA-335.

[31] The Mozart Consortium. Mozart Oz 1.2.4. Web Site, Sept. 2002. http://www.mozart-oz.org/.

[32] B. Venners. *Inside the Java Virtual Machine*. McGraw-Hill, 2nd edition, Jan. 2000.

[33] A. Walter. An implementation of the programming language DML in Java: Compiler. Diplomarbeit, Fachbereich Informatik, Universität des Saarlandes, Feb. 2000.

[34] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo, France, Sept. 1992. Springer-Verlag.