Virtual Machine Based Heterogeneous Checkpointing*

Adnan Agbaria†  Roy Friedman
Computer Science Department
Technion – Israel Institute of Technology
Haifa, 32000
Israel

November 21, 2000

Abstract

Checkpointing an application is the act of saving the application’s state during its execution on stable storage, so that if the application fails, it can be restarted from the last saved state, thereby avoiding loss of the work that was already done. A heterogeneous checkpoint/restart mechanism allows to restart an application from a saved state that was taken in a hardware architecture and/or operating system that can be different from those in the machine on which it is restarted. This paper explores how to construct such a mechanism at the virtual machine level. That is, rather than dumping the entire state of the application process, the mechanism reported here dumps the state of the application w.r.t. a virtual machine. During restart, the saved state is loaded into a new copy of the virtual machine, which continues running from there. The heterogeneous checkpoint/restart mechanism reported here was developed for the OCaml variant of ML. The paper reports on the main issues encountered in building such a mechanism and the design choices made, presents performance evaluations, and discusses some lessons and ideas for extending the work to native code OCaml, and to Java Virtual Machines.

---

*This research is supported by the Bar-Nir Bergreen Software Technology Center of Excellence
†Also supported by Israeli Ministry of Science and Technology, Grant Number 1628-2-00
1 Introduction

The world today is interconnected. That is, computers inside departments and organizations are connected by LANs, and these LANs are all connected to the Internet. Not surprisingly, a large body of research, as well as several commercial products, have addressed the issue of how to use idle cycles of as many connected computers as possible for carrying large computations, utilizing these resources in an efficient way. This is done both at the LAN or cluster level, often referred to as *cluster computing* [5, 7, 16, 26], as well as at the Internet level, e.g., in the form of *meta-computing* [1, 21, 23] and *grid-computing* [14].

One aspect of executing heavy computations, whether in an interconnected settings or on a single local computer is checkpoint and restart [12, 28, 32]. That is, it is common practice to periodically save the state of a long running computation so that if the application fails, it can be restarted from the last checkpoint. This avoids losing the entire computation due to such a failure. Most practical checkpoint/restart (hereafter, C/R) work done so far assumes that the system is *homogeneous*, or in other words, a system in which all computers are of the same architecture and run the same operating system. This is despite the fact that most interconnected systems are *heterogeneous*, i.e., consist of nodes with different hardware architectures and operating systems.

The problem of C/R is much more difficult in heterogeneous systems than it is in homogeneous ones. In the latter, checkpoint can simply be done by dumping the process core. There are optimized implementations that attempt to greatly reduce the amount of saved data. But even they can rely on the fact that the architecture and operating system of the computer in which the failed application is restarted are the same as the ones in the computer in which the state was saved. In particular, such implementations can assume that the data representation, the machine registers, the stack, heap and data segments, as well as machine native instruction sets are all the same. However, in most organizations, and in particular in the Internet, the environment is heterogeneous, and therefore the above assumptions do not hold. Thus, to improve the utilization of cluster, meta, and grid computing systems, it is desirable to construct C/R mechanisms that can operate across multiple platforms and operating systems.

In recent years there is a growing proliferation of virtual machine based programming languages, such as Java, C#, and OCaml, that also implement their own memory management. These languages are translated to byte-code representation, which is independent of a particular architecture’s instruction set. Similarly, the state of the application depends on the virtual machine’s internal registers, stack, heap and data segment representation, which are independent of operating system and hardware. Also, in order to support internal memory management, and in particular, garbage collection, data types are often tagged, which is useful when trying to interpret a block of saved data dumped in one architecture, and use it in another architecture. It is therefore conceivable to construct a heterogeneous C/R mechanism that will checkpoint the application’s state at the virtual machine level, rather than the native machine’s level, making the problem of cross platform and operating system C/R much easier.

This paper reports on our experience in constructing a heterogeneous C/R system for the OCaml virtual machine [4]. The system we developed operates across a wide range of Unix-like operating systems as well as Windows NT, and across multiple hardware platforms such as Intel’s Pentium, Compaq’s Alpha, Sun’s UltraSparc, and IBM’s RS/6000. Additional ports to any other platform should be fairly trivial, given that we already support both little-endian and big-endian formats, and both 32-bits and 64-bits architectures. Our implementation relies on the structure of the OCaml virtual machine, and the fact that it is the same on all these platforms. We utilize the garbage collection features of OCaml and the tagging of data types in dumping and reconstructing
the data across platforms. We also employ the natural optimization that data translation between representations is done only during recovery time, and only if the recovery happens on a different architecture than the one in which the application was saved.

In this paper we report on our experience, design decisions, methods and optimizations. We also present a performance analysis, and provide some thoughts and comments about how this work can be extended to OCaml native code and to Java Virtual Machine. Another (side-effect) contribution of this paper is that, to the best of our knowledge, it includes the most thorough written description of the OCaml virtual machine and (actual) garbage collection mechanism.

The rest of this paper is organized as follows: Section 2 includes a description of OCaml, its virtual machine and garbage collection mechanism. We review the main problems in trying to implement heterogeneous C/R in Section 3. The implementation is presented in Section 4, followed by a qualitative analysis and performance measurements in Section 5. Section 6 talks about related work, and we conclude with a discussion in Section 7.

2 The OCaml Virtual Machine

Objective Caml (OCaml) [4] is a dialect of the ML programming language. The OCaml system supports both native-code and byte-code compilation. For native-code, the compiler generates an optimized machine-dependent executable code. On the other hand, the byte-code compiler of OCaml generates hardware-independent code that can run in any machine architecture or OS for which there is an OCaml runtime environment. OCaml is currently used mainly in research projects, but also in several commercial products.

The OCaml run-time and tools are open source, allowing people to add or modify the system. Moreover, the OCaml Virtual Machine (OCVM) is implemented in C with well documented source code. The implementation of OCVM is fairly simple and small. Additionally, the OCaml virtual machine runtime and tools were ported to most platforms and operating systems. All this makes OCVM a good vehicle for investigating heterogeneous C/R at the virtual machine level.

In this chapter we describe the low level design and implementation of OCVM. We first describe data representation and manipulation. We then describe the main memory components of the system: the heap, the stack, and the byte-code, followed by a brief description of how threads are created and managed by the system. Lastly, we detail the garbage collector and the interpreter mechanisms of the system. The purpose of this description is to obtain better understanding of OCVM, which is required to understand the issues that needs to be dealt with, and the solutions we describe later in the paper. Most of these issues are in fact common to many virtual machines, and thus this section is also useful as a general reference beyond OCVM.

2.1 Main Memory Areas

OCVM has three main memory areas: the heap, the stack and the byte-code. These areas are allocated by OCVM during initialization. We explain below these components and their purpose in the system.

Heap The main purpose of the heap is to maintain application data. OCVM allocates both static and dynamic data in the heap. Memory deallocations in the heap are performed implicitly by the garbage collector (see Section 2.4 for more details).

Initially, OCVM allocates a heap of a predefined size. The heap is expanded if necessary during runtime according to applications’ requirements. OCVM maintains the heap as a
linked list of memory chunks ordered by increasing memory addresses. Each chunk consists of a chunk header, which holds its length and a pointer to the next chunk.

Initially, all the heap space is declared as free and added to the freelist data structure. During the application’s execution, allocations are done by allocating objects from freelist, and a deallocation returns an object to freelist.

Stack OCVM allocates the stack area during initialization. The stack is allocated by using malloc with a default size of 16K. If the stack becomes full, OCVM reallocates a new stack with double the size of the old one.

The main purpose of the stack is, as usual, to save local variables and return addresses of function calls. However, local variables of non-primitive data types are actually dynamically allocated in the heap, and the stack only includes a pointer to them.

Byte-code During initialization, OCVM allocates an area of memory by using malloc for maintaining the OCaml program byte-code. The byte-code is generated by using the byte-code compiler and the same file can be loaded and executed in any architecture type.

2.2 Data Representation

The basic data type of OCaml is a 32-bit word in a 32-bit architecture and a 64-bit word on a 64-bit architecture. Usually, this data type contains a value of either an integer or a pointer. A pointer value in OCaml, which is a 4-byte aligned address value, points into one of the mentioned above main memory areas. In our description below we assume a 32-bit data representation. The discussion for 64-bit architectures is analogous.

The least significant bit of an OCaml handle value determines its representation. If the least significant bit equals to 0, this value is a pointer. Otherwise, it contains an actual value. Thus, given an OCaml value we can determine, in runtime, if the value is a pointer or not.

To be precise, OCVM has two main data type representations: atomic and blocked types. An atomic data type may exist anywhere in the memory, while a blocked data type exists only in the heap. The garbage collector of OCVM must be able to distinguish between these types at runtime for its internal manipulation.

2.2.1 Atomic Types

An atomic data type is a 32-bit size, which is either a small integer value or just a 0-tuple block. Atomic types of 0-tuple block are allocated statically once and for all the program’s life outside the heap. These types are used by OCaml to represent a user-defined abstract type. In fact, there is a global table called atom_table, whose size is 256 entries, that maintains all these atomic data. An atomic type can be an integer (type int), characters (type char), Boolean (type bool), or void (type unit).

2.2.2 Blocked Type

Blocked data types are created in the heap. For each blocked data there is a strictly defined 32-bit header. The block header consists of three fields. The fields are tag, color and size. The first 8 bits of the header contains the block tag, which specifies the block type among various blocked types in OCaml. We list the interesting blocked types below. The next two bits of the header contain the block color, which determines the block’s state according to the garbage collection
mechanism (see Section 2.4 for more details). The last 22-bit field contains the block size, which is the number of the block fields except for the header. The other entries in a block data are also 32-bit each. Moreover, each of those fields is simply an OCaml value that contains either an actual value or a pointer value. Figure 1 shows an example of a blocked data type with four fields.

![Figure 1: A blocked type with its header in OCaml](image)

As mentioned above, the tag field of the header indicates the block type. This is significant for OCVM internal usage. For instance, a tag value less than the constant NoScanTag means that the block is garbage-collected. The various types of general blocked types in OCaml includes:

**A Closure Type** A Closure type represents functions in OCaml. This block has two fields. The first field is a pointer into the byte-code area, and the second contains the function environment which is defined by OCVM. Notice here that this block should be ignored by the GC, as it is a data type that is allocated during system initialization.

**An Object Type** An Object type in OCaml is a blocked type allocated in the heap that contains well-formed values, which can be recursively traversed by the garbage collector. The tag value of a blocked object type should be less than NoScanTag and the color field contains the object status according to the GC algorithm (see the generational algorithm in Section 2.4).

**A String Type** Strings in OCaml are allocated and then represented by a blocked type in the heap. The fields of such blocked type contain the string data in ASCII.

**A Floating-point Type** A floating point in OCaml is allocated in the heap as a blocked type. The fields of the block represent the floating point value in the IEEE double precision standard format.

### 2.3 Threads

From the operating system’s point of view OCVM is a regular process. Thus OCVM has its private data, text and stack memory segments, which are allocated by the operating system. Consequently, the OCVM defines and creates its own stack, heap, registers, etc. in which it provides runtime environment and executes the byte-code application.
OCVM supports both single and multi-threaded applications. In the case of a multi-threaded application, OCVM executes and manages all the threads. Each thread has its private stack and registers in the system. OCVM schedules a ready thread to run according to specific policies defined by the system.

2.4 Garbage Collection

A Garbage Collection (GC) algorithm periodically scans the heap in order to reclaim unreachable objects. In OCaml, garbage-collectible data are blocked types allocated in the heap, as described before. The garbage collector recognizes a garbage-collectible block by using the tag value which is defined in each block header.

OCaml uses an algorithm for garbage collection that combines the generation and mark-sweep algorithms [17]. The heap is divided into two areas, the young generation and the old generation areas [11, 10]. The young generation is a fixed area of memory defined between the young_start and young_end variables. There is a data structure in the system that maintains pointers from the old generation into the young one. On the other hand, the old generation area is simply the heap, which is a linked list of memory chunks ordered by increasing memory addresses. Each chunk consists of a chunk header, which gives its length and a pointer to the next chunk, and an integral number of memory pages of 4 KB each. Chunks are added to the old generation by calling malloc as necessary, and a page table is used to distinguish memory pages belonging to a chunk in the old generation from other memory.

Memory allocation is performed either linearly in the young generation area or in the free list of the old generation area. Initially, an allocation of an OCaml data value is applied in the young generation. If the data remains alive long enough, it is transferred to the old generation area during a garbage collection operation on the young generation [10]. Eventually, if there is no more space for memory allocation in the free list, OCaml extends the heap by calling malloc.

2.4.1 Symbols and Data Structures

In this section we present and define symbols and data structures that are used or have meanings in the garbage collection mechanism. Illustrating and clarifying these values contributes to the understanding of the garbage collection mechanism described below.

The color field of a blocked header specifies the status of the block w.r.t. the garbage collector, as specified follows:

Blue: Means that the block belongs to the free list.

White: Means that the block has not yet been visited by the mark phase.

Gray: Means that the block has been visited by the mark phase, but its children have not been visited yet.

Black: Means that the block has been visited by the mark phase, and immediate children have been visited too.

The notations and data structure labels that are used by the garbage collection mechanism are:

mutator This notation includes all the processes and threads defined by the user code.

roots All the mutator's pointers to the heap.
reftable  All references from the old generation to the young one.

grayvals  Stack pointers to gray values, used for mostly-depth-first marking of the old generation

freelist  It is a linked list of memory blocks reserved for allocation in the old generation.

2.4.2 The Collection Mechanism

We now turn to the (sketch of the) garbage collection mechanism implemented in OCaml [11, 10]. This mechanism is based on the generational algorithm where a copying mechanism is used for copying live objects from the young generation to the old one. On the other hand, Dijkstra’s mark-sweep algorithm [9] is used to reclaim the old generation.

A minor garbage collection is a collection of the young generation. It copies the live values from the young generation into the old generation, using free memory obtained from the freelist of the old generation. The live values are those reachable from the globals, the stacks, the roots, or the refstable. The space used for the young generation is recycled after a minor garbage collection, and the refstable becomes empty.

A major garbage collection works on the old generation. This operation uses the incremental mark-sweep algorithm, in a number of slices. One slice of a major garbage collection is executed after every minor garbage collection operation. There are two kinds of garbage collection slices: mark slices and sweep slices. A sequence of mark slices (called the mark phase) followed by a sequence of sweep slices (the sweep phase) constitute one cycle of a major garbage collection. The amount of marking (resp. sweeping) to do in a mark (resp. sweep) slice is determined by the total size of the live values being promoted from the young generation in the preceding minor collection: the more promotions, the more garbage collection work must be done. This is an attempt to distribute the garbage collection work over computation in a fair manner, ideally ensuring that the mutator never has to wait long for the collector to free memory.

During the mark phase, garbage-collectible blocks are divided into three classes, represented by colors: white, gray and black. A stack called grayvals of references to gray values is used to speed up marking, so that one marking pass over the old generation will usually suffice. The heap is pure if all gray values below the marking pointer markhp are also in the grayvals stack. If the grayvals stack overflows, and cannot be extended, then the heap becomes impure and a second marking pass is needed. The mark phase completes when there are no more gray values in the heap. This is the case when the grayvals stack is empty, the heap pure, and the marking pointer has reached the end of the heap.

During the sweep phase, chunks of the heap are swept sequentially:

- Every white value, i.e., an unreachable value, is made blue and put on the free list.
- Every black value, i.e., has been visited by the collector, is made white.
- Every blue value, i.e., is on the free list, is ignored.

No gray values can remain after the mark phase. After a number of sweep slices, the sweep pointer is at the end of the heap, and a full garbage collection cycle is completed. Then the mark phase is entered again after graying all values reachable from global variables, the stacks, and the roots.

2.4.3 Garbage Collection Frequency

In OCaml there is no dedicated thread for performing garbage collection. Garbage collection can be started in any thread while it is running; garbage collection might start when an allocation
from the young generation fails. The thread, which initiated the memory allocation that triggered garbage collection, delays its normal execution and starts performing garbage collection. First, a minor collection is started, which copies all live objects to the old generation. Then, a major collection starts immediately in the old generation.

2.5 Interpretation

As we mentioned before, the OCaml compiler produces a byte-code file that is executed by OCVM. The interpreter is responsible for reading, decoding and then executing byte-code on the actual machine. OCVM defines three virtual registers for executing byte-code:

PC: The program counter register points into the byte-code memory where the next instruction will be fetched.

SP: The stack pointer register points into the stack head.

ACCU: The accumulator helps in maintaining values for the interpreter.

During program execution, the interpreter fetches the byte-code instruction pointed by PC. Then, it decodes the instruction and set its parameters if needed. Lastly, the interpreter executes the appropriate operations in the actual machine and saves results as well. After each execution of a byte-code instruction, the interpreter checks a set of flags for handling signals if needed.

3 Main Issues in Implementing Heterogeneous C/R

We have implemented a prototype of heterogeneous C/R in OCVM. In this Section we outline the main issues that one has to deal with in implementing heterogeneous C/R, while the actual implementation is presented in Section 4. For clarity, we separate the discussion about checkpoint from the discussion about restart.

3.1 Checkpoint Issues

Here we discuss the issues that one has to deal with during checkpointing in a virtual machine and present solutions to some of these issues. Most of the solutions we propose are implemented in our prototype system.

3.1.1 Performance

Usually, checkpoints are performed periodically during the execution of an application. Hence, checkpoints might occur several times in the course of execution. The overhead imposed by checkpoints should therefore be minimal, otherwise it would not be worth using this mechanism. The main things that can affect the overhead of taking a checkpoint on the run-time of the application is whether the entire application is blocked or not, the size of the data being saved, and the time to calculate what data needs to be stored. In order to avoid blocking the application, it is possible to fork a new process; the child process then saves its state and exists. In saving the data, there is a tradeoff between the overall size of the saved file and the number of write operations. The easiest thing is to simply dump everything to disk. On the other hand, in order to minimize the size of the checkpoint file, more sophisticated rules must be implemented. These rules, however, require scanning parts of the memory, and involve more write operation to the disk, thereby creating additional overhead.
3.1.2 Safe Points

Each byte-code instruction is interpreted by the virtual machine into several native machine instructions. For each byte-code instruction, the OCVM interpreter fetches the instruction, decodes it, and then executes the instruction in the native machine. More precisely, the execution involves reading the next instruction from the byte-code stream, extracting the possible arguments from the operation code, and branching to the part of the interpreter that executes the instruction. This is analogous to the way hardware executes a (native) machine instruction.

A checkpoint operation can be invoked in arbitrary moments during the run of an application, e.g., as a result of a timer expiration. However, since a byte-code instruction is typically executed by several native machine instructions, it is possible that a checkpoint will occur in the middle of the execution of a single byte-code instruction. This might result in saving an inconsistent state of the application. That is, since we are operating in the virtual machine, using byte-code granularity, during restart, the system will either have to reissue the instruction, or start from the next instruction. However, reissuing the instruction might not be legal, if the part that got executed is not idempotent. On the other hand, skipping the instruction at restart is also not desirable, since the instruction did not terminate, e.g., did not update the memory or registers, etc. Hence, special care should be taken to avoid saving the state of the virtual machine in the middle of a byte-code instruction.

We define a safe point to be a point in the execution in which a checkpoint would save a consistent application’s state. A checkpoint taken in a safe point is hereby referred to as a safe checkpoint. A safe point in the byte-code application can be found between every pair of subsequent instructions or during an instruction that does not change the system’s state. Namely, if $E_1$ and $E_2$ are subsequent instructions, then a checkpoint between these instructions is safe. On the other hand, if $E$ is an instruction that does not change the system state, then a checkpoint that occurs while $E$ is executing is also safe. A condition that does not change any flag or value it reads from is an example of such an instruction. Note that in this case, $E$ would have to be reissued during restart.

We claim that user initiated checkpoints are safe. A user initiated checkpoint is performed explicitly by invoking a checkpoint function call in user code. By definition, such a checkpoint is applied between two separate byte-code instructions, and is safe. However, if $C$ is a system initiated checkpoint, it might be unsafe. A system initiated checkpoint could start occasionally at any point during runtime, and in particular, in the middle of the execution of a byte-code instruction. Thus, care needs to be taken to delay such checkpoint until the system reaches a safe point.

In our implementation we ensure safe checkpoints even when initiated by the system. When a checkpoint is invoked, the OCVM sets a specific flag indicating a checkpoint request and continues normal execution. As we explained before, the OCVM interpreter checks the signal and status flags before fetching a new instruction, and if set, invokes the corresponding handler routine. We have modified the interpreter to also examine the checkpoint flag, and if set, to call the checkpoint routine (see Section 4 for details).

3.1.3 Checkpointed Data

The checkpointed data is the data that has to be saved during checkpointing and should be enough for restarting the process. In a homogeneous checkpointing mechanism, the checkpointed data simply consists of the process data segment, heap, stack, and its registers. In a heterogeneous C/R mechanism for virtual machines, however, the checkpointed data has to be defined in the virtual machine equivalents of the above. That is, we do not save the virtual machine process, but
rather save the logical state of the application w.r.t. the virtual machine. We outline the types of
checkpointed data that needs saving:

Global Data We define *global data* to be all data allocated in the data segment of the actual
machine that are defined and accessed by the application. These data should be saved during
checkpoint. For example, in OCaml with C calls, data allocated by malloc are part of global
data that needs to be saved during checkpoint.

The Heap In a virtual machine, usually, the heap contains all the data that are allocated either
statically or dynamically. In OCVM the heap also contains all the stacks of live threads if
the application is multi-threaded. Therefore, all reachable cells in the heap are part of the
checkpointed data.

The Main Stack For every virtual machine there is a main stack managed by the virtual machine
for maintaining the application’s return addresses and local variables. In OCVM, the main
stack is allocated during OCVM initialization by using malloc as explained in Section 2.
The stack contains relevant data of the application and it should be saved as part of the
checkpointed data.

Registers Every virtual machine defines several virtual registers that are used in executing the
application in the same way that actual registers are used by the hardware. These registers
maintain the running application state and they should be checkpointed as part of the
checkpointed data.

In order to access and then save the checkpointed data, the checkpointer should be implemented
inside the virtual machine, which is corresponding to the kernel level of an OS running on an actual
machine. In section 4 we present a heterogeneous C/R mechanism that is integrated to the OCVM
and has access to all checkpointed data.

3.1.4 Thread Synchronization

OCVM, like many other virtual machines, supports multi-threaded applications. Threads are
created and managed by OCVM without any intervention by the actual OS. Checkpointing a multi-
threaded application is one of the traditional restrictions in checkpointing mechanisms [29, 22]. In
fact, many existing homogeneous systems, such as Starfish [5] and Candor [21], do not support
checkpointing of multi-threaded applications.

There are several points that should be taken into account during developing a checkpoint
mechanism that supports multi-threaded applications. First, every checkpoint C in a multi-threaded
application should be *consistent*. That is, every read from a global data that occurs before C
must read a value that was written before C. An easy way to ensure consistent checkpointing for
multi-threaded applications is to stop all threads, take the checkpoint, and then resume normal
multi-threaded execution.

Another problematic point of checkpointing a multi-threaded application is how a checkpointer
would obtain the status of all live threads in the application. In homogeneous checkpointing with
kernel support, this information can be retrieved from the kernel data. On the other hand, in a
user-level threads implementation, this information is saved in the process memory and should be
traced by the checkpointer. In our case of heterogeneous C/R, fortunately, threads are allocated
and managed by a virtual machine and since our implementation of heterogeneous C/R is at the
virtual machine level, we can access all necessary threads’ information and relevant data.
3.1.5 Abstract Registers

A virtual machine has its particular abstract registers for executing an application. The number of abstract registers and their purpose differs from one virtual machine to another according to its implementation. In OCVM, for example, there are four abstract registers: PC, SP, ACCU, and ENV, as discussed in Section 2. These registers are defined as local variables inside the interpreter function, implemented in C. On the other hand, these abstract registers should be saved as part of the checkpointed data because they contain relevant information about the running user application.

In our checkpoint implementation (see Section 4), the checkpointer is applied as a function call from the interpreter. Thus, we pass the abstract registers as actual parameters to the checkpointer and then it saves them as part of the defined checkpointed data.

3.2 Restart Issues

We now turn the discussion to restart issues that are relevant to virtual machines in general, and OCVM in particular.

3.2.1 Data Representation

Nowadays, data representations typically differ in two areas. The first aspect is big-endian vs. little-endian convention. The second aspect is word size of either 32-bit or 64-bit. To make our work general, we should be able to restart an application on any architecture type regardless of the original architecture. For instance, we should be able to checkpoint a running application on Intel 32-bit architecture and then restart it on Alpha 64-bit architecture. Similarly, to checkpoint a running application on Sun Solaris with big-endian and then restart it on a PC with little-endian representation.

We assume that application failures are rare, and therefore, the overhead for checkpointing should be minimal. Thus, we prefer to save data in its native representation. During restart, data is restored according to the machine it is being restarted on. Note that this is similar to the optimization in CORBA in which data sent in GIOP messages is sent in the sending machines representation, and it is the responsibility of the receiver to translate it as necessary [24].

The conversion mechanism converts checkpointed data according to the restarting machine’s data representation if it differs from the one in which it was saved. For example, when the checkpointing machine has little-endian while the restarting one has big-endian or vice versa. Each actual value, which is not a pointer, is then converted accordingly. Also, if the two machines have different word size, then we expand or compress each value as proper. Note, however, that in the transition from 64-bit to 32-bit some data might be lost, e.g., if we had an integer value larger than $2^{31}$. On the other hand, our conversion mechanism takes care to maintain the sign of values.

3.2.2 Pointers Adjustment and Values Distinction

Upon restart, we restore all the checkpointed data and allocate them in memory areas as they were in the original checkpointed process. For example, a global value defined in the data segment should be restored in the new data segment. Also, the heap should be restored with its original image. However, we cannot guarantee that the heap, for instance, will be allocated in the same address as it was in the original machine.

The checkpointed data mostly contains pointer values that point into the original memory address space. Since in heterogeneous C/R the restarted application might be restarted in a
different architecture or OS, there is no guarantee that pointer values of the checkpointed data will be valid in the new machine. Hence, we need to adjust each pointer to point to the relevant new address.

In order to properly adjust pointers in the restarted application, we first need to distinguish pointers from non-pointer values. Then, for each pointer, we need to adjust its contents to point to the corresponding cell in the new machine. In OCVM, for example, we can distinguish a value if it is a pointer or not by the least significant bit. If the bit equals zero, the value is a pointer. Otherwise, it is an actual value. Therefore, during restart we can examine each value and adjust it accordingly. The remaining question here is how to adjust pointers.

Usually, pointers point into the main memory areas of the virtual machine. During checkpointing, we save the memory boundaries of all these areas. Then, during restart, for each value, we first examine if it is a pointer and into which memory area it was pointing. We verify this by comparing the pointer value with all the saved boundaries. Lastly, we adjust the pointer to the new address by adding the offset to the begging of the specified memory area.

We emphasize here that during restart the garbage collector should not work to avoid any access to an incorrect address. Hence, to disable the collector during restart and only enable it afterwards.

3.2.3 Thread Synchronization

Thread synchronization is also required during restart. In restarting a multi-threaded application, we have to recreate all threads as in the original application and then restore their status. Moreover, we need to ensure that the application will only continue running when the entire memory was reconstructed and all its threads were recreated. If the application, however, starts running and at least one of its threads has not been recreated yet, the application could enter a deadlock or perform an error.

Such problems of deadlock during restart could appear when threads use semaphores, monitors, condition variables and other facilities in the application level. For example, if a thread $T_1$ is sleeping on a waiting queue or condition variable $R$, which is held by $T_2$, and $T_2$ starts running before $T_1$ is recreated, then $T_2$ will not be able to wake $T_1$ when it releases $R$. Thus, when $T_1$ is finally recreated, it will remain blocked forever. There are possible solutions to the above problem, but the simplest and safest approach is to enforce the policy that no thread can start running until all threads are fully restored.

3.2.4 File Descriptors

In general, restoring file descriptors is one of the traditional restrictions of C/R (see [21, 22] for more details). Particularly, in a heterogeneous C/R case this problem becomes even more difficult because file systems may differ between two different machines and/or OSs. Thus, we prefer not to deal with this problem individually for each system. Instead, we need to tackle this problem within the virtual machine and use an abstract mechanism for heterogeneous C/R.

In our case, OCVM allocates a particular structure called channel for each opened file descriptor. A channel maintains relevant information about the corresponding opened file, that contains the actual file descriptor, I/O buffers for intercepting I/O operations, etc. Any I/O operation on a file descriptor is intercepted by OCVM and then performed on the corresponding channel, and lastly performed in actual machine level. Therefore, in order to support file descriptors checkpointing, we save all the channels as part of the checkpointed data and then use their information for reopening the files in the restarted application. In section 4 we describe our mechanism for restarting file descriptors on behalf of these channels.
4 Implementing Heterogeneous C/R

In this section, we describe the C/R mechanism that we implemented in the virtual machine of OCaml version 2.02. We start by describing in details our checkpoint mechanism, followed by a description of the restart mechanism. We then evaluate this mechanism by illustrating its advantages and disadvantages.

4.1 The Checkpoint Mechanism

Figure 2: The C/R layer resides in the OCVM for checkpointing the application layer

Figure 2 depicts the location of the C/R mechanism inside OCVM. Checkpoint is performed by invoking the checkpoint mark routine. This routine simply sets the checkpoint flag `chkpt_flag` and then exits immediately. Since there is no general simple way to figure out whether the instruction executed by OCVM in a given moment is idempotent or not, checkpointing the application at this point might generate an unsafe checkpoint, as defined in Section 3.

At the beginning of any byte-code instruction, OCVM reaches a consistent state in which it allows performing any interrupts. Therefore, the interpreter of OCVM checks before executing any new byte-code instruction if there are signals to perform first. At this point, the interpreter also checks the checkpoint flag. If the checkpoint flag has been set, the interpreter performs checkpointing. Figure 3 depicts parts of the interpreter source code responsible for deciding when and how to invoke the checkpoint function.

```
Instruct(CHECK_SIGNALS):
  if (chkptState == Checkpoint) {
    Setup_for_c_call;
    chkptRc = checkpoint(sp, pc, accu, env, extra_args);
    Restore_after_c_call;
  }
  if (something_to_do) goto process_signal;
Next; /* Go to the next instruction */
```

Figure 3: Relevant interpreter code that calls the checkpoint function

The checkpoint routine saves the application’s state w.r.t. OCVM on stable storage. We outline the steps that it takes below.

1. Create a new process for performing copy-on-write checkpointing [6, 28]. Namely, the child process performs checkpoint while the parent process exits the checkpoint routine and continues normal execution. This greatly reduces the impact of checkpointing on the running time
of the application. On the other hand, Windows NT does not support fork. Thus, in NT we use a checkpointing thread, that has to block the rest of the application during checkpoint, so the overhead on NT is higher.

2. Invoke a minor garbage collection for reclaiming garbage in the young generation area of the heap. Recall that this collection copies all live data to the major heap and leaves the minor heap empty. Therefore, there is no need to save the minor heap as part of the checkpointed data.

3. Disable threads scheduling. If the application is multi-threaded, threads are normally scheduled to run according to a timer. This timer should be disabled in the checkpointer process to prevent running other threads during checkpointing.

4. Open a temporary checkpoint file for saving the checkpointed data.

5. Save the machine architecture and the application type, i.e., single-threaded or multi-threaded. Moreover, save the value one. This value is examined by the restarting machine to determine if it has different big-endian/little-endian representation than the saving machine. Also, save a flag indicating whether the architecture is 32-bit or 64-bit.

6. Save relevant boundary addresses of main memory areas. As we mentioned before, we need to remember the original memory areas where pointers of the original machine point to.

7. Save the abstract registers of OCVM. These registers maintain relevant information about the OCVM system state.

8. Dump the major heap into the temporary checkpoint file. This is done by tracing all heap chunks and saving them. To achieve better performance, each chunk is simply dumped to disk. Since these include the information about live objects and the free list, this information will be available after restart.

9. Save global data and the atomic table of OCVM. Global data that we save here do not belong to the application level. They have some relevant information belonging to OCVM. For example, we save all pointers maintaining information about the free list inside the heap. This data is useful for restoring the OCVM state eventually.

10. Save the application stack area. In a multi-threaded case, save the current stack of the running thread.

11. If the application is multi-threaded, save all the other thread stacks and other global data regarding threads.

12. Save information about the opened channels. As we mentioned before in Section 3, channels maintain abstract information about the opened files and this information is important for restoring file descriptors.

13. Write a signature at the end of the temporary checkpoint file, and close it. Finally, delete the old checkpoint file and set the temporary file as the new checkpoint file.

14. Terminate the checkpointer process.
int checkpoint(value sp, code_t pc, value accu, value env, long extra_args)
{
    int rc, chkptFd;
    ...
    gc_minor(Val_unit);
    pid = fork();
    if (pid >0) /* Parent process */
        exit(0);
    if (MULTI_THREADED)
        disableTimer();
    chkptFd = open(tempChkptFileName, 066, 022);
    saveBoundaryAddresses(heap_start, start_code, stack_high);
    saveArchtCharacters();
    CHKPT_WRITE(chkptFd, atom_table, wsize); /* Save atomics */
    rc = saveFreeList(chkptFd);
    /* Save OCVM globals */
    globals.global_data = global_data;
    globals.allocated_words = allocated_words;
    globals.extern_sp = extern_sp;
    CHKPT_WRITE(chkptFd, &globals, sizeof(globals));
    /* Save the OCaml registers */
    registers.sp = sp;
    registers.pc = pc;
    registers.accu = accu;
    registers.env = env;
    registers.extra_args = extra_args;
    CHKPT_WRITE(chkptFd, &registers, sizeof(registers));
    dumpHeap(chkptFd, heap_start);
    saveStack(stack_start, sp);
    if (MULTI_THREADED)
        saveThreadsInfo();
    saveChannels();
    closechkptFd);
    link(tempChkptFileName, chkptFileName);
    unlink(tempChkptFileName);
    exit(0);
}

Figure 4: Relevant source code of the checkpoint function in OCVM

Figure 4 depicts a pseudo code of the checkpoint mechanism in OCVM. This code helps understanding the checkpoint mechanism listed above.

Note that if a failure occurs during checkpoint, the temporary checkpoint file would not contain all the checkpointed data and so it cannot be used for recovery. Hence, we have to ensure that the operation of creating a new checkpoint file is atomic. Indeed, in our implementation we actually save the checkpointed data in a temporarily file and then after completing the checkpoint successfully we set the temporary file to be the new checkpoint file.

As a final comment, in order to enable system initiated checkpointing, we introduced a new environment variable called CHKPT_INTERVAL. This variable is set to the length of the time interval between two consecutive checkpoints, if system initiated checkpointing is needed, or either not defined or set to -1 if the system should not perform checkpoints. Also, the environment variable CHKPT_FILENAME is used to specify the name that the checkpoint file should have.

4.2 The Restart Mechanism

We have modified OCVM such that during initialization, OCVM accepts as a parameter whether the application should be started from the beginning or from a given checkpoint file. A new environment variables CHKPT_STATE is used for this. If the application should be started from
scratch, but perform checkpoints from that point on, CHKPT_STATE is set to enable. Otherwise, to perform a restart from a checkpoint file, CHKPT_STATE is set to restart and CHKPT_FILENAME should be set to the name of the checkpoint file to start from. If CHKPT_STATE is undefined, C/R will not be performed and the application runs normally.

We added the chkpt_init function to be applied during OCVM initialization in order to determine the execution policy by examining the new environment variables and then updating the chkptState variable accordingly. Figure 5 depicts the relevant source code of chkpt_init.

```c
void chkpt_init(char **argv) {
    char *env;
    chkptFileName = getenv("CHKPT_FILENAME");
    env = getenv("CHKPT_STATE");
    if (env && !strcmp(env, "disable")) {
        chkptState = Disable;
    } else if (env && !strcmp(env, "restart")) {
        chkptState = Restart;
    } else {
        chkptState = Enable;
    }
    /* Get temporary name checkpoint files */
    strcpy(tempChkptFileName, chkptFileName);
    strcat(tempChkptFileName, "XXXXXX");
    mktemp(tempChkptFileName);
}
```

**Figure 5**: Relevant source code of the checkpoint initialization function in OCVM

After OCVM initialization, the interpreter is called to execute the byte-code application. If the chkptState variable equals to restart, the first step that the interpreter performs is calling the restart function. Otherwise, it starts the application execution from the beginning. Figure 6 depicts the relevant code of the interpreter where it takes a decision to perform restart.

```c
value interpret(code_t prog, asize_t prog_size) {
    ...
    if (chkptState == Restart) {
        Setup_for_c_call;
        restart(&sp, &pc, &accu, &env, &extra_args);
        Restore_after_c_call;
    }
    Next;
    ...
}
```

**Figure 6**: Part of the interpreter source code where the decision for performing a restart is taken

Below we present the restart mechanism that corresponds to the checkpoint mechanism presented in the previous section. We outline the mechanism step-by-step and later provide its pseudo code in Figure 7.

1. Open the provided checkpoint file and check if it has a valid signature.
2. Read the original machine and application type from the checkpoint file:
- Read the constant one and then compare it to the current machine. If they are different, then the checkpoint was taken in a machine with big-endian representation while restart is occurring in a machine with little-endian representation or vice versa. If data conversion is needed, set an appropriate flag here.

- Read the word bit-length of the original architecture. Compare it with the word bit-length of the current machine. If they are different, then restart is occurring in a machine architecture that differs from the architecture where checkpoint was taken. If there is a difference in the word length, set a flag to indicate whether expansion or reduction of word bit-size is needed.

- Read the application type. This is needed since the application is still not running, so there is no other way for OCVM to know what application it is restarting other than reading it in the file. If the application is multithreaded, perform thread synchronization and read the threads’ information.

3. Read the original boundary addresses. We use these addresses to determine where each pointer was pointing in the memory of the original machine, so we can adjust pointers during restart as needed.

4. Read abstract registers and fix any addresses in them if needed.

5. Restore the heap. In this step we first expand the heap to include the same number of chunks as the original corresponding checkpointing system. (Recall that OCVM allocates heap chunks of constant size.) Then, we overwrite each heap chunk by reading old heap values from the checkpoint file and storing them in the corresponding new chunks.

6. Restore the atomic table values and global values and then fix pointers of global values according to the new addresses.

7. Restore the application stack. If the checkpointed application stack is longer than the initial stack created by the system, we reallocate an appropriate stack for restoring the previous stack. Then we pass over all the stack and adjust pointers to the new addresses.

8. If the application is multi-threaded, restore threads information and local stacks.

9. Adjust pointers in the heap. Here we do not need to visit all the cells in the heap. Instead, we use the garbage collection mechanism to visit only the reachable and live blocks, then inside each block we fix the pointers.

10. Restore channels and adjust pointers there. For the standard file descriptors, we only need to update their information according to the checkpointed data. For other file descriptors, however, we first need to open each file and then update its information as well.

11. Close the checkpoint file and return to the interpreter.

Notice here that if the new machine is a 64-bit architecture while the old one is a 32-bit architecture, we have to expand each cell to 64-bit representation and vice versa. Moreover, we have to deal with integer representation in case of little-endian/big-endian mismatch and fix the values as necessary. These corrections should be done in each relevant step described above.
int restart(value **sp, code_t *pc, value *accu, value *env, long *extra_args) {
    int rc, chkptFd;
    ....
    chkptFd = open(chkptFileName, O_RDONLY, 0);
    restoreArchChars(chkptFd);
    /* Read original address boundaries */
    CHKPT_READ(chkptFd, &baseAddr, sizeof(baseAddr));
    CHKPT_READ(chkptFd, &registers, sizeof(registers));
    fixPointers(registers, baseAddr);
    restoreHeap(chkptFd, heap_start);
    CHKPT_READ(chkptFd, &atom_table, wsize);
    rc = restoreFreeList(chkptFd);
    CHKPT_READ(chkptFd, &globals, sizeof(globals));
    fixPointers(globals, baseAddr);
    CHKPT_READ(chkptFd, &oldStack_sz, sizeof(int));
    if (oldStack_sz > stack_sz)
        realloc_stack(oldStack_sz);
    restoreStack(chkptFd, sp, stack_high);
    fixStackPointers(sp, baseAddr);
    /* Fix pointers in the heap */
    chunk = heap_start;
    while (chunk != NULL) {
        p = (header_t *) chunk;
        chend = chunk + Chunk_size (chunk);
        while (p < chend) {
            hd = &hd[p]; sz = Wosize_hd(hd); tg = Tag_hd(hd);
            if ((I_sblue_hd (hd)) && ((tg < No_scan_tag) || (tg == Final_tag))) {
                vp = p + 1;
                for (i = 0; i < sz; i++, ++vp)
                    if (Is_block(*vp)) fixPointer(*vp, vp, baseAddr);
            }
            p += Whsize_wosize (sz);
        }
        chunk = Chunk_next (chunk);
    }
    restoreChannels(chkptFd);
}

Figure 7: Relevant source code of the restart function in OCVM

5 Evaluating the C/R Implementation

5.1 Advantages, Disadvantages, and Restrictions

In this section we evaluate our C/R mechanism in a qualitative manner. We examine both the advantages and drawbacks of this mechanism, and list the restrictions it imposes on the programming model.

The most important advantage of this C/R mechanism is the ability to do cross platform checkpoint and restart. Yes, another advantage that stems from implementing C/R inside the virtual machine is the ability to overcome some of the typical restrictions of user space C/R. This is because the virtual machine manages and operates the application level in a similar manner to the kernel of an OS. Thus, we can extend the virtual machine to overcome many of these restrictions, such as threads, signals, and files. Finally, since we only dump the heap, stack(s), the used parts of the data segments, and abstract registers, the overall size of the checkpoint file is smaller than in implementations that dump the entire core. However, this latter improvement can also be achieved in C/R mechanisms that do not have access to a virtual machine.

The main shortcomings of our implementation include:

- Currently, our C/R mechanism is only implemented for OCVM. OCaml language is not as widely used as other languages like C++ and Java. Thus, implementing heterogeneous C/R in JVM (Java Virtual Machine) is likely be more useful, although more difficult. Never the
less, we believe many of our ideas can be used for Java as well, and OCaml was a convenient starting point.

- This C/R mechanism is implemented in a virtual machine that executes byte-code. However, byte-code usually executes much slower than native code, due to the need to interpret it in software. Hence, one of most interesting research topic is to provide heterogeneous C/R for native code applications. The main difficulties here are the fact that native code relies on the actual registers and stacks of the native machine, which could be hard to map across platforms. Also, different compilers might generate different native code, so it may not be possible to map every instruction on the checkpointing platform to a corresponding instruction in the restarting platform. Hence, the definition of safe points needs to be refined for this case.

- This C/R mechanism is designed and built for serial applications and does not work for parallel applications. As a future research direction, we intend to provide heterogeneous C/R for parallel message-passing applications, by integrating this work with our Starfish system [5].

We now list the restrictions of our heterogeneous C/R mechanism:

- C/R restricts interleaving of C and OCaml. That is, since we only checkpoint memory allocated inside the virtual machine’s heap, any C part of an OCaml application must only use the OCaml library routines for allocating and deallocating memory, rather than using the OS native ones.

- Outside connections, like sockets, are not supported. If a failed application has an outside connection, our C/R mechanism cannot recover this connection and the application might not run properly.

- We can recover file descriptors, but only if the same file is accessible from the restarting machine. Also, files can only be read from, or written sequentially. When a file is written sequentially, during restart we simply seek the file to the position it had during restart. However, if the file is written randomly, we cannot guarantee correct semantics, unless we also implement a log file, which we have not so far.

5.2 Experimental Results

This section presents experimental results that measure the overhead of checkpoint and restart in our implementation, as described in Section 4. We also present a time breakdown for the significant tasks involved in both checkpoint and restart.

5.2.1 Platforms Supported

We tested our implementation of heterogeneous C/R on several hardware architectures and OSs. To verify heterogeneous C/R, we have performed C/R across these distinct platforms. For example, taking checkpoints on a PC running Linux and restart in several other machines such as Alpha, RS/6000, and an Intel-based PC running Windows NT.

Table 1 lists all the machines that we have used for performing heterogeneous C/R in OCVM. For each machine we describe its architecture and the installed operating system. This represents a wide selection of different platforms on which heterogeneous C/R works. Moreover, we would like to mention the fact that between Unix-like OSs (including Linux), the port only required recompiling
the code, so it is highly likely that we can support any other Unix-like OS by simply recompiling on that system.

5.2.2 C/R Overhead

To offer a C/R mechanism for general usage, we have to show and verify that it is acceptable in terms of its imposed overhead. If this overhead is too high, a user would rather prefer failures than endure the performance penalties of checkpointing [28].

In this section we measure the C/R overhead that is imposed on the application runtime. These measurements show that even though our C/R is implemented at the virtual machine level, the imposed overhead is low and even negligible compared to the application normal runtime. As shown below, the checkpoint overhead is proportional to the size of the application data. There are two reasons for this: On all Unix-like systems, checkpoint is taken in a forked process. On systems that implement copy-on-write fork, each write causes an exception, which adds an overhead, so the more data there is, the more exceptions will occur. On the other hand, in systems that fully duplicate the memory during fork, this duplication takes time that is linear in the memory size. The other reason is that we save the state on stable storage. Thus, even if this is done on a separate process, it consumes systems resources, and slows down the original application process.

We have two test applications for C/R measurements. The applications are matrix multiplication and insertion sort. These applications are simple and are often used for measurements. In addition, these test applications are computational intensive and have the property that their runtime and memory requirements grow proportionally with an input parameter \( n \), which is a size of arrays.

The matrix multiplication test consists of only a few lines of OCaml code. Figure 8 depicts the main function code of this test. The time complexity here is \( O(n^3) \), while the space complexity is \( O(n^2) \), assuming an \( n \times n \) matrices. Thus, by increasing the parameter \( n \), the checkpoint file can be made large proportionally to \( n^2 \). Our second application, insertion sort, is an example application published in the official OCaml user’s guide [4]. Figure 9 shows this test as it appears in the OCaml user’s guide. The time complexity of this test is \( O(n \log(n)) \) and the space complexity is \( O(n^2) \), where \( n \) is the size of an array to be sorted.

The difference between these test applications, however, is the content of checkpointed data. Since the insertion sort application is implemented recursively, the stack grows during runtime due to many recursive calls (see Figure 9). On the other hand, in matrix multiplication, the stack remains the same and the checkpointed data grows as the heap grows proportionally to \( n^2 \).

We report results of C/R from several machine types. We measure checkpoint overhead with different values of \( n \), and then measure restart in different machine types as listed in Table 1.

Figures 10 and 11 show the runtime of the tests with and without checkpointing, when run-
let matmul n =  
  let mat1 = Array.make_matrix n n 0 in  
  let mat2 = Array.make_matrix n n 0 in  
  (* Initialization *)  
  ...  
  for i = 0 to (n-1) do  
    for j = 0 to (n-1) do  
      for k = 0 to (n-1) do  
        mat3.(i).(j) <- mat3.(i).(j) + (mat1.(i).(k) * mat2.(k).(j))  
      done;  
    done;  
  done;  

Figure 8: Relevant source code of the matrix multiplication test

let rec sort lst =  
  match lst with  
  | [] -> []  
  | head :: tail -> insert head (sort tail)  
and insert elt lst =  
  match lst with  
  | [] -> [elt]  
  | head :: tail -> if elt <= head then  
    elt :: lst  
  else  
    head :: insert elt tail

Figure 9: Relevant source code of the insertion sort test

ning on rodrigo (a Linux machine). The X axis represents the checkpoint size in MBs, and the Y axis represents the runtime in seconds. These figures show that the runtime with one checkpoint is mostly equal to the original runtime, namely the checkpoint overhead is almost negligible. The checkpoint overhead is at most one percent, which is quite small compared to many other implementations, e.g., [6].

Figure 10: Comparing the running time of matrix multiplication with and without checkpointing on rodrigo

Figure 12 shows the restart time on various platforms and different checkpoint file sizes, for checkpoints taken on rodrigo (see Table 1) for the matrix multiplication test. We summarize below the machines on which we restarted the application, and the data transformations they required.
Insertion sort

- checkpoint overhead

Figure 11: Comparing the running time of insertion sort with and without checkpointing on **rodrigo**

**rodrigo** This is an Intel Pentium II machine running Linux Redhat 6.1. This is also the original machine where the checkpoint was taken. We have drawn the restart time on this machine for comparison with other platforms.

**PC8** This is an Intel Pentium II machine running Windows NT. We compare here the restart time on the same architecture, but with different OS. Since our C/R is implemented at the virtual machine level, the restart time is roughly the same as in the original machine.

**csd** This is a dual processor Sun Ultra Enterprise machine running Sun Solaris OS. This machine has a big-endian number representation, while the original one has little-endian representation. Therefore, in restart we need to convert each value from little-endian to big-endian representation. Because of this conversion there is a gap in the restart time between **csd** and **rodrigo**.

**sp2148** This is a dual processors Alpha machine running Linux RedHat 6.2. This machine is a 64-bit architecture, while **rodrigo** is a 32-bit architecture. Hence, during restart we need to convert each value of 32-bit size to its equivalent 64-bit representation. Because of this conversion, we see the difference in restart time on **sp2148** vs. the original machine.

Figure 12: Comparing the restart time on different platforms
5.2.3 Timing for Substantial Parts of the C/R Mechanism

In section 4 we presented the C/R mechanism in details. In this section we measure the time spent in the substantial parts of this mechanism in the checkpointer process during checkpoint. By substantial we mean those parts that might contribute noticeable time to the overall checkpoint time. For these measurements, we used the matrix multiplication test on rodrigo running Linux with the ext2 file system. Note that due to the use of fork, these times do not directly affect the checkpoint overhead, which is measured w.r.t. to the original process running time. Yet, by examining these times, we can evaluate our mechanism and look for further optimizations.

We believe that there are two main substantial parts for checkpointing: saving the heap area in stable storage and committing the checkpoint file. We notice here that more than 80 percent of the checkpoint time is spent in saving the heap. In addition, we notice here that the bigger the checkpoint file becomes, so does the time for committing it. The time to commit mainly comes from the commit operation, which deletes the old checkpoint file. In order to delete large files on ext2, it may be necessary to read all the associated inodes, which could span several hierarchies, in order to find and free all blocks belonging to the file.

We notice also in Figure 13 that other parts take less than 5 percent of checkpointing. These parts contain the minor garbage collection, saving the abstract registers, and saving the stack. All these operations are performed either in a small area of memory or on a few distinct values.

![Checkpointing](image)

Figure 13: During Checkpointing - timing the substantial parts versus various checkpointed data size

During restart, the substantial parts are restoring the heap and fixing pointer values inside it. Figure 14 depicts the restart time in seconds. We notice here that these substantial parts take more than 90 percent of restart. Note that there are another two substantial parts of restart that we have already evaluated in the previous section. The first part is the time needed for converting from big-endian to little-endian representation or vice versa. The second part is the time needed for converting from 32-bit to 64-bit word size representation or vice versa. Figure 12 exhibits those times.
Related Work

There are two main approaches for implementing C/R in uniprocessor systems. One approach is to implement C/R in user space. For example, C/R in Starfish [5], Condor [21], CLIP [8], and Libckpt [29], have been implemented in user space. The second approach is to implement C/R in kernel mode, such as in Unicos [18] and Sprite [25]. In this approach, the operating system exports two system calls, `chkpt` and `restart`, to the user (according to the POSIX standard). There is an additional approach that combines both user and kernel spaces.

Usually, implementing C/R in user space imposes a set of restrictions. There are many checkpoint implementations that do not support signals, timers, memory-map files, or shared memory [22]. Other problematic issues include parent-child relationship, inter-process communication, and multi-threading [22, 27, 29]. Implementing C/R in kernel space eliminates most of these restrictions, but requires modifying the Kernel.

A great deal of research has addressed the issue of supporting C/R in distributed systems. There are two main approaches for providing C/R in these systems. The first approach is coordinated checkpointing, where processes coordinate the checkpoint operation, and the latter approach is called uncoordinated checkpointing, where each process takes a checkpoint independently of other processes [12, 28]. Moreover, there are many distributed systems in both industry and academia that support C/R for message-passing applications, e.g., Starfish [5], Condor [21], and LoadLeveler of IBM [2].

Most published work regarding C/R addresses homogeneous systems. To the best of our knowledge, there are only a handful of research papers dealing with C/R mechanisms for heterogeneous systems. Most of them are based on compiler intervention and adding explicit calls in the code. We briefly describe these papers below.

Ferrari et al. have developed an approach to checkpoint the dynamic state of processes in a platform-independent manner called Process Introspection [13]. This approach centers on the semantic-preserving modification of programs by a source code translator that incorporates C/R functionality into processes. Another approach of C/R in heterogeneous systems was proposed by Theimer and Hayes [31]. In their proposed approach, the state of a process is examined and checkpointed using compiler-generated symbol mapping information. Moreover, there are other approaches that deal with process migration on heterogeneous systems. For example, Symphony
has a proposed mechanism for server migration on the Internet [15].

In addition, automatic program-transformation mechanisms have already been used to obtain heterogeneous C/R. These mechanisms based on translating a program’s code to another in which new calls are added and define explicit data structure for performing heterogeneous C/R. Porch is a C-to-C compiler that translates C programs into semantically equivalent C programs that are capable of saving and recovering from portable checkpoints [3]. Moreover, e2ftc is another C-to-C compiler for obtaining portable checkpointing, that is based on the same technique used in Porch [30].

As a platform-independent language, there are some research work for achieving heterogeneous C/R in Java. These work probe C/R at the language level. Lawall and Muller, for example, suggested a checkpoint mechanism in which the state of a Java application can be recovered from the contents of the object fields [20]. They ask that each object invokes particular methods from a predefined interface called Checkpointable. Then, in the implementation level, they perform a serialization for these objects. In addition, there are several Java-based mobile agent systems that use serialization to implement agent migration [19].

7 Conclusions

In this paper we presented a heterogeneous C/R mechanism, that works across a wide range of hardware architectures and operating systems. This mechanism is integrated into the OCaml virtual machine, that runs OCaml byte-code. The main benefits of our approach, aside from simplifying the problem of heterogeneous C/R, is that by working at the virtual machine level we can eliminate many typical restrictions that exist in many other user level C/R approaches, as well as resulting is smaller checkpoint files, as explained in Section 5. The main drawback of our approach is that it relies on byte-code execution, which is slower than native code execution.

Yet, our experimental results have shown that our C/R mechanism achieves heterogeneous C/R with minimum overhead. Moreover, this mechanism provides transparent checkpointing in which there is no need to recompile or modify an application’s source code in order to take advantage of C/R.

We view our work here as a first step towards achieving true heterogeneous C/R. That is, we would like to extend our work in two immediate directions, namely, supporting native code generated from OCaml and byte-code Java and/or C#. In considering native code generated from OCaml, it appears that the main difficulties lie in the fact that native code generated from OCaml is executed directly by the hardware, relying on the native machines registers, heap, stack, etc. Thus, there is a need to translate these concepts between the machines and OSs. Moreover, since the native code generated by different compilers might be different, it might not be possible to map a given instruction on one machine to a corresponding instruction on another machine. Of course, one option is to turn off all optimizations, so it is possible to trace back machine level instructions to their corresponding source code instructions. In this approach, a safe point would be the border between machine level instructions that correspond to a single source code instruction. This solution is problematic, since unoptimized code runs much slower than optimized code, especially on new hardware architectures.

Another idea might be to declare beginnings and endings of functions and loops as safe points. The problems with loops is that many optimizers might reorder things from inside the loop out. Even worse, in many scientific programs the entire code might reside within one loop. Thus, this restriction would prevent taking checkpoints during the entire "interesting" part of the computation. With functional programs, such as OCaml, it is common to place everything within functions, so
normally a loop consists of multiple function calls. However, the OCaml optimizer can in-line functions, which might mean that even for pure functional programs, function boundaries might not always serve as good safe points.

Finally, even if we solve the problem of safe points, this is still not sufficient, since we would have to translate the function/procedure calling stack to reflect the new representation of the return addresses and parameters. Nevertheless, the fact that OCaml relies on its garbage collection and is a strongly typed language would greatly simplify these tasks.

When considering Java, our work is highly relevant, since any work that attempts to do similar things in JVM would have to tackle the same issues that we did. Also, since Java is also a strongly typed language that relies on a garbage collector, our solutions can probably be adapted to Java as well. On the other hand, there are several different JVMs, which might have different internal representations and each might manage its memory segments differently. Thus, to support cross JVM C/R we might need to be able translate these concepts, which starts to resemble heterogeneous native code C/R. Moreover, the use of Just-In-Time compilation is now common practice in most JVMs. This complicates the task since we need to support both native code and byte-code. Moreover, parts of the same application might run inside the JVM while other parts might run as native code, with cross dependencies between them. A heterogeneous C/R mechanism for Java needs to be able to support all this. Finally, another complication with Java is the fact that many JVMs allow using native threads when available, unlike OCaml that always uses its own threads.

Acknowledgements: We would like to thank Mike Factor and Jim Plank for insightful discussions. Also, we wish to thank Compaq\textsuperscript{TM} for allowing us free access to an Alpha machine.
References


