Alcom-FT Technical Report Series
ALCOMFT-TR-03-1

A Portable Virtual Machine for Program Debugging and Directing*

Camil Demetrescu ‡ Irene Finocchi §

Abstract

Directors are reactive systems that monitor the run-time environment and react to the emitted events. Typical examples of directors are debuggers and tools for program analysis and software visualization. In this paper we describe a cross-platform virtual machine that provides advanced facilities for implementing directors with low effort.

Keywords: virtual machines, debugging, directors, reversible computing.

1 Introduction

The intrinsically dynamic nature of running programs, opposed to static implementation code, makes programming, debugging, and performance monitoring difficult and time-consuming. Development environments typically provide source-level debuggers that help see things in action, clarifying how the code is interpreted by a computer, but do not help much with portraying the behavior of complex programs at a more abstract level neither do provide advanced tools for fault localization. Devising powerful methodologies and flexible tools for assuring software reliability still remains a much difficult problem [16]. Platforms for software development should provide not only editors and compilers, but also a flexible collection of specialized tools for debugging, analyzing, checking, and visualizing software. Such tools are usually referred to as *directors* [21], i.e, reactive systems that monitor the run-time environment and react to the events emitted by the processes. Writing directing applications typically requires low-level primitives for:

- Monitoring and controlling events such as variable referencing, memory allocation, program interruption, function call and return.
- Detecting memory leaks and wrong memory accesses.
- Providing access to old execution states, so that execution must not be restarted when a fault is discovered, but it is possible to backtrack quickly to a state where a wrong statement did occur.

Directors may be quite difficult to implement, especially if they must provide a continuous (and not query-driven) monitoring of the ongoing state of the computation. In this paper we describe a cross-platform virtual machine that provides advanced facilities for implementing directors with low effort. Before detailing our contributions, we review the main approaches to program directing proposed in the literature.

1.1 Related work. In case of compiled programs, which are directly executed by hardware, directing applications are usually implemented by taking quite invasive approaches, which might invoke the use of *operating system primitives* that offer special-purpose functionalities [11]. The heterogeneous nature of computing environments, however, limits the portability of such tools on different platforms: in general, one cannot rely on the availability of hardware or OS monitors that provide access to process execution states, and the situation may be even worse in networked environments [21]. A different approach hinges upon *program instrumentation*, i.e., on augmenting the source/object/executable code to emit events that contain monitoring information [2, 3, 14, 23]. Instrumenting source code has the advantage of referring to variable names more directly and treating data more "semantically", but typically requires implementing an auxiliary parser or compiler. Moreover, not all platforms provide high-level primitives for accessing every aspect of the execution state, and thus flexibility and power of the director may be limited. On the other hand, assembly code can be instrumented almost automatically, but is strongly platform-dependent, and thus portability becomes a major issue.

In view of the considerations above, it is natural to implement directors in environments that provide *interpreted execution* [13, 19, 20]: the underlying interpreter can indeed be extended with special support for debugging and performance monitoring, and events like variable referencing, statement execution, program interruption, function call and return, execution run-time errors can be easily detected [12]. However, even using interpreted execution, common limitations of previous

*Work supported in part by the IST Programme of the EU under contract n. IST-1999-14.186 (ALCOM-FT) and by the Italian Ministry of University and Research (Project "ALINWEB: Algorithmics for Internet and the Web").

‡Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Rome, Italy. Email: `demetres@dis.uniroma1.it`. URL: `http://www.dis.uniroma1.it/~demetres`.

§Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via di Tor Vergata 110, 00133 Rome, Italy. Email: `finocchi@disp.uniroma2.it`. URL: `http://www.disp.uniroma2.it/users/finocchi/`.

environments seem to be lack of portability, generality, and availability. Furthermore, interpretation slows down the running time. However, this may not be highly critical in applications like debuggers and software visualizers, which typically require a large amount of human interaction. Advances in hardware technologies, caching strategies, and use of just-in-time compilers may also cope with this slowdown. Not last, intermediate-level languages (such as Java bytecodes) can be interpreted much more efficiently than high-level ones. Hence, a popular trend is to compile programs into an intermediate form based on an abstract machine definition. This also guarantees portability of programs, once the *virtual machine* is made available on different platforms. A well-known success story on portability is the JVM [18], released at the beginning of 90's and now in widespread use. A recent competitor is the .NET platform [5], announced by Microsoft in mid-2000.

The JVM has been especially designed for supporting the Java language [8]. As a consequence, Java bytecodes are not well suited as target for other languages: implementing even subsets of languages other than Java has posed many difficulties and lead to deteriorate performances [9, 10]. The .NET platform is somewhat more general and is claimed to be designed with multiple language support in mind [5]. However, its instruction set does not include instructions that manipulate addresses, and this may be a strong limitation for targeting languages such as C or C++, for which directors and debuggers would be mostly useful.

Another desired feature within debugging and monitoring environments is *reversible execution*, which provides access to old execution states (see, e.g., [4, 6, 15, 17, 25]). The location of a program fault is indeed the first point of investigation during the debugging process. If the faulty statement is reached or, more in general, a desired state is passed over, one typically must restart the execution. This can be avoided using step-by-step execution, which however may be long and boring. Even the traditional breakpoint mechanism may be inadequate for programs with many loops and branches, since too many breakpoints should be set to hit the actual execution path up to the faulty statement. Reversibility allows the programmer to reach easily previous states of execution, thus overcoming such difficulties. In addition to fault localization, reversible execution also provides a useful context for supporting *patching*, which is the possibility of inserting corrective code when a fault has been recognized and resuming the program execution using the modified code without re-entering the edit-compile-execute cycle from scratch [19].

Due to implementation difficulties, runtime overhead, and storage requirements for storing "historic" data, reversibility is rarely supported. The very first attempts date back to the late 60's: EXDAMS [27], for instance, provided an execution backtrack facility for FORTRAN programs, whose complete execution history could be saved and then replayed in a "post-mortem" fashion. Zelkowitz incorporated an execution backtrack statement in the PL\I programming language [27]. Similar language-based approaches were taken in [7, 26]. Speedup techniques for reversible execution have been shown in [1], where a structured-backtracking approach is introduced: reversibility is allowed only over complete program statements, such as while loop or conditional statements, by recording the value of a subset of variables in a suitably identified change set. Some of the limitations of structured backtracking, e.g., backtracking into a loop from outside or into a function call, are overcome in [3], where an efficient reversible debugging mechanism is described. However, the approach in [3], based on program instrumentation at assembly code level, is very platform dependent and does not support reversibility of system calls nor different levels of history logging granularity, which may be very useful to reduce the storage overhead.

1.2 Our contributions. This paper presents the Leonardo Virtual Machine (LVM), a cross-platform abstract machine that provides advanced facilities for program monitoring, controlling, and debugging. Directors written within the LVM framework are independent of the underlying computer architecture and operating system, can be realized with relatively low effort, and are able monitor the state of the ongoing computation continuously, and not just on demand. A comprehensive set of system calls for memory, process, file, and event handling provides access to the LVM facilities.

The LVM can run multiple processes in time sharing and has primitives for both synchronous and asynchronous event handling. It offers fine-grained control over memory accesses to detect several kinds of memory faults, and integrates reversible computing capabilities by maintaining a history of previous computation states. In order to reduce time and space overhead for storing historic data, history logging can be supported at different levels of granularity.

Currently, executable files for the LVM can be generated using the Leonardo C Compiler, a portable ISO C89 compiler originally developed as part of a previous project, Leonardo IDE [6]. Similarly to Java "`.class`" files, LVM "`.leo`" executable files are fully cross-platform, i.e., they can be run on any machine on which the LVM is available. The LVM is based on a cross-platform open architecture and is currently being developed and tested in different operating systems (including Windows, Linux, Solaris, and MacOS X) as part of the Leonardo Computing Environment project [24].

2 The Leonardo Virtual Machine

The LVM is an abstract machine that is able to execute programs written in an assembly-like language. Differ-

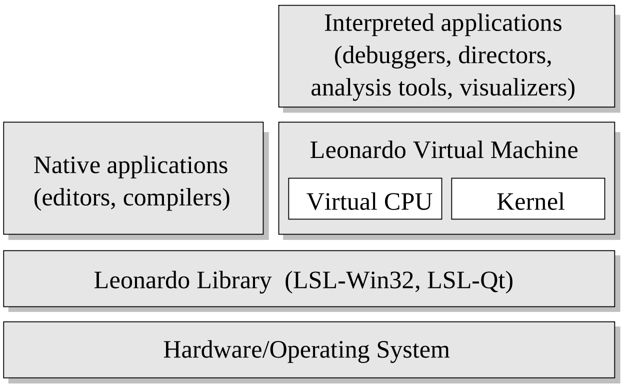

Figure 1: Architectural layers of the LVM.

ently from other modern virtual machines such as the JVM and .NET, the LVM instruction set supports unrestricted memory accesses and a flexible parameter passing approach that make it possible to deal with source languages such as C and C++. The LVM is equipped with a comprehensive set of system calls for memory, process, file, and event handling.

The LVM is made of two main modules (see Figure 1): a virtual CPU (VCPU), which is the LVM runtime engine, and a Kernel, which offers operating system facilities for process and resource management. To guarantee portability and platform-independence, the LVM is implemented on top of the Leonardo Library (LL), a cross-platform general-purpose collection of software components that bridge the LVM and other tools to the underlying operating system [24].

2.1 Virtual CPU. The VCPU is a stack-based execution machine with only 5 registers: a program counter (PC), a stack pointer (SP), and three segment registers (R0, R1, and R2) for accessing memory. Its instruction set covers about 100 elementary control flow, data flow, and arithmetic/logic operations. In the current version, the VCPU follows a simple interpretative approach that helped us easily recompile and test the LVM on different platforms. The actual implementation of the full VCPU instruction set is just about 180 lines of C code. As an example, in Figure 2 we show our code for the `muls` instruction, which multiplies the two signed 32-bit words on top of the evaluation stack. The corresponding x86 and PowerPC assembly blocks have been produced by the Metrowerks CodeWarrior IDE 4.1 compiler with maximum optimization level. To support time-sharing process scheduling without using platform-dependent alarm mechanisms, the fetch-execute loop is simply exited every time the special `leave` instruction is encountered in the code: in this way, the control is passed back to the Kernel, which may schedule another process and re-enter the fetch-execute loop. We remark that, although no real-time performances can be guaranteed with this technique, if `leave` instructions are uniformly distributed in the code, this simple method can be quite effective in practice. The VCPU performs almost no runtime checks: the problem is solved by allowing external clients (e.g., the Kernel) to install callbacks that are executed to perform the desired verifications whenever the VCPU encounters "critical" instructions such as load/store (`ld`/`st`), jump to subroutine (`jsr`), return from subroutine (`rts`), and the like. With this technique, the VCPU is completely independent of the rest of the LVM, and thus it may be easily reused in other projects.

```
#define Exec_muls_  { --SP_; ++PC_; \
   *(i4*)(SP_-1) = *(i4*)(SP_-1) * *(i4*)(SP_); }
```

x86

```
sub   ebx,0x4
add   ebp,0x4
lea   eax,dword ptr [ebx-0x4]
mov   edi,dword ptr [eax]
imul  edi,dword ptr [ebx]
lea   eax,dword ptr [ebx-0x4]
mov   dword ptr [eax],edi
jmp   CCPU_Exec+0x61)
```

PowerPC

```
subi   r25,r25,4
addi   r28,r28,4
lwz    r3,-4(r25)
lwz    r0,0(r25)
mullw  r0,r3,r0
stw    r0,-4(r25)
b      *-4864
```

Figure 2: Implementation of the VCPU `muls` instruction and corresponding x86 and PowerPC code.

2.2 Kernel. The Kernel includes a collection of *managers* that deal with different aspects of process, event, and resource management. The *Process Manager* is responsible of creating, suspending, resuming, killing, and scheduling processes on the VCPU. Processes are scheduled according to a simple round-robin strategy. The *Memory Manager* maintains a memory segment for each process, which maps its logical address space. Block allocation requests made by the process are served using a first-fit strategy, and the memory segment is dynamically resized if no free hole can be found. The *Event Manager* is responsible of event handling, maintaining for each process a queue of "interrupt" requests made by other processes or by Kernel managers. Finally, a special manager, called *Log Manager* provides support for maintaining a history of previous states of a process computation. More details about memory, event, and history log management will be discussed in Section 4, Section 5, and Section 6, respectively.

3 Portability

Portability across different platforms is one of the main design goals of the Leonardo Virtual Machine. Experienced programmers know that this goal is fraught with many pitfalls, caused by both software and hardware differences. An obvious problem arises when functionalities that are not covered by standard libraries are needed: for instance, creating a semaphore in Linux and in Win32 requires substantially different system calls.

There are, however, much more subtle sources of problems. For instance, there are features in standard languages such as C that are intentionally left platform-dependent for efficiency reasons (see, e.g., the size of intrinsic types such as `int`). Furthermore, you may discover that a program that runs correctly on an x86-based machine causes a bus error when recompiled on a SPARC-based architecture because of an unaligned memory access. Last, but not least, numeric format differences might create serious problems in data exchange: for example, an executable program compiled on a big-endian machine will not be read on a little-endian platform unless proper conversions are made.

While the LVM is carefully implemented according to tight guidelines, most portability issues are addressed in the underlying Leonardo Library, which is designed to abstract and hide platform differences.

3.1 Leonardo Library. The Leonardo Library (LL) is a cross-platform application development toolkit. It lets application developers target different operating systems with a single application source code, providing a well-documented collection of C *components* with a clean and modular interface [24]. An LL component is a coherent collection of functions, types, and constants targeted to a specific task. The library, which features about 500 C functions, covers a large number of tasks including graphic user interface, thread, synchronization, I/O and memory management, and contains components with fundamental data structures and general-purpose utility functions. The LL is currently implemented in two versions, sharing the same common application programming interface: LL-Win32, based on the Microsoft Win32 library, and LL-Qt, based on the Qt library developed by Trolltech.

4 Protection

Debugging deals with unsafe, incorrect programs, and often bugs lead to unpredictable program behaviors. It is therefore crucial for a debugging runtime environment to be prepared to any kind of unexpected event. This implies that a large amount of checks is to be performed, and this might be extremely time-consuming. To alleviate this problem, system designers typically manage to replace run-time checking with load-time checking, whenever possible. For instance, most verifications in the Java Virtual Machine happen at load time, taking advantage of the "safe" nature of the Java programming language. Unfortunately, if different languages are to be considered, a larger amount of run-time checks might be required. For instance, a C program has the freedom to access any memory location and divert the control flow to any code address, and those addresses may be available only at run time. The LVM is prepared to any kind of abnormal event, while still effectively exploiting load-time checking. In the remainder of this section we describe how protection is implemented in the LVM.

4.1 Memory protection. A typical source of abnormal events comes from incorrect memory accesses. Operating systems typically prevent processes from getting outside their logical address space, but usually allow them to make any kind of damages inside it. For instance, a simple unbounded scan of a dynamically allocated array might cause the program to overwrite data used by the Memory Manager to keep blocks and holes inside the heap: the error may not be noticed immediately, but it might cause successive allocation operations to fail with no apparent reason. Overwriting code sections may lead to even more unpredictable effects. To prevent this kind of events, the LVM keeps a *memory mask* for each process. The memory mask maps every 32-bit memory word in the process address space into an 8-bit control word that specifies whether that word can be legally accessed by the program. With this technique, identifying fine-grained memory violations inside a process address space can be done at run time with just one additional memory lookup.

4.2 Stack protection. A process in the LVM spends most of its execution time accessing the top of the runtime stack[1]. Therefore, checking for stack overflow/underflow at instruction level may be extremely time-consuming. The solution adopted by the LVM is to combine both load-time and run-time checking. At load-time the code is analyzed statically, visiting the graph of basic blocks and computing the maximum stack usage over all possible paths. This information allows us to perform stack overflow checking only when entering a function call.

To prevent stack underflow, i.e., popping an empty stack, the LVM uses again static analysis to check if the code handles the stack correctly. Nevertheless, underflow may still happen if a function is entered at an incorrect point, so that pop without push operations are performed. This is solved by ensuring that `jsr` instructions target the correct function entry points.

4.3 Code protection. The correctness of jumps in the code is mostly checked statically in the LVM. The only runtime verification is concerned with calls to subroutine: to check if the target code address is valid, a special bit in the memory mask is used. This bit, set at load time, indicates whether a memory word is a valid function entry point. This simple solution yields a very efficient run-time code protection strategy.

5 Event handling

Processes running in the LVM emit several kinds of events. For instance, events are raised in case of ab-

[1] 101 out of 107 VCPU instructions use the top of the stack.

normal operations such as division by zero, stack overflow, and memory protection faults. Other events correspond to low-level actions such as jump to subroutine, return from subroutine, and memory load/store, as well as higher-level actions performed by the Kernel, including memory allocation and deallocation, process start and termination, and the like. Events can also be generated explicitly by the programmer using the `KRaiseEvent` system call.

A process can listen to events raised by another process (possibly itself) using the system calls `KNewAsyncEvtH` or `KWaitSyncEvtH`. Both of them require to specify the PID of the process raising the events of interest, and a handler that is to be executed when the specified event does occur. Differently from `KWaitSyncEvtH`, which suspends the process until the specified event is raised, using `KNewAsyncEvtH` events are processed asynchronously without having to wait for the event.

Events might also carry some information about the way they were generated: for instance, a handler for a memory store will receive in its parameters the base address and the size of the memory segment actually written. We observe that, differently from signals in UNIX, processes raising an event in the LVM do not have to specify the recipient of the event, and a single event might have several listeners. As we will see in Section 7, this flexible design makes it easy to write programs that monitor/control the actions of other programs by listening to the events they generate.

6 Reversible computing

The Log Manager maintains a history of a process computation, supporting rollback to previous execution states. Informally, a *state* is identified by the values of the registers of the VCPU and by the content of the memory image of the running program. The approach followed by the Log Manager consists of keeping an *incremental log* of state changes. Special points in the executable code are treated as *checkpoints*, while the execution of any sequence of instructions between two consecutive checkpoints is regarded as a *transaction*. During forward execution, at each checkpoint a new *log record* is stored on a *history stack*: the log record maintains information on the state changes occurred during the transaction. In more detail, it contains addresses and old values of the memory words that did change. To perform a backward step, the log record on the top of the history stack is popped and its content is used to restore the state corresponding to the previous checkpoint. This rolls the transaction back as if it was an atomic step.

The incremental approach is very popular and it has been widely used in quite different contexts, including distributed computing and databases. Natural questions in our setting are: how large does the history stack grow for a running program? What is the runtime overhead of keeping a history stack? Roughly, if a program spends time $T(n)$ on an input of size n, the size of the history stack for reversing the entire program will be proportional to $T(n)$: this is clearly a worst-case space bound. Nevertheless, many optimizations can be added to reduce in practice the time and space overhead of maintaining a history stack. In addition to a careful design and implementation of data structures, our solution is based on a suitable choice of "checkpointable" states, and exploits effectively the stack-based nature of the LVM.

6.1 Basic optimizations. Building a log record during a transaction and pushing it on the history stack may be extremely time-critical. Since the history stack may run arbitrarily large, we keep it in external memory, using standard *buffering techniques* to reduce the number of I/Os. To limit space overhead, it is also crucial not to save *multiple changes* of the same memory word that take place during the same transaction. We do this efficiently by storing a *dirty bit* for each memory word: the bit is set to 1 at the first word change and is reset at the end of the transaction. With this approach, we expect that the longer are the transactions, the smaller is the resulting history stack. Thus, controlling the granularity of checkpoints can yield substantial space savings (see Section 8).

6.2 Stack optimization. The runtime stack of a process is maintained as a block in the process address space. At any time, the actual stack may occupy only a fraction of that block, while the rest is free. A key observation is that the data in the free part of the stack block is not being used by the program, and therefore it can be regarded as *not* belonging to the state of the process. As a consequence, if a transaction places temporary data on the top of the stack leaving the final stack height unchanged, the flow of temporary data can be ignored by the Log Manager without affecting the correctness of the history logging strategy. Using this kind of transactions, which we call *stack-preserving transactions*, one can obtain substantial space and time savings.

An interesting observation is that in stack-based computation models, the bursts of low-level machine instructions corresponding to statements of high-level languages typically yield stack-preserving transactions. Thus, since intermediate states generated by those bursts are uninteresting for debugging purposes, it is quite natural to consider a burst as the shortest possible transaction. In LVM executable programs, bursts of instructions corresponding to high-level statements are delimited by the `leave` instructions (see Section 2.1). Moreover, LVM executable programs satisfy the following relevant property, which is checked statically by the Process Manager at load time:

Stack invariant: transactions between consecutive `leave` instructions within the same subroutine are stack-preserving.

Since most VCPU instructions modify the top of the runtime stack by manipulating temporary data, exploiting the stack invariant yields substantial space and time improvements. To this aim, the Log Manager considers the states corresponding to `leave` instructions as the only "checkpointable" states. As a consequence, the maximum level of history log granularity corresponds to a checkpoint per `leave` instruction (granularity 1).

6.3 Controlling checkpoint granularity. To support different levels of granularity, a subset of `leave` instructions can be selected as checkpoints, obtaining longer atomic transactions. The granularity can be chosen either based on the program structure (e.g., by regarding the execution of a code block or of an entire subroutine as a transaction), or by simply grouping together a user-defined number of consecutive `leave` instructions.

6.4 Undoing Kernel system calls. Tools that support reversible computing typically do not address the issue of reversing system calls (see, e.g., [3]). This is a major limitation for a debugging system. The LVM is designed to support reversible computing over *both* program statements and Kernel system calls. We use a memory-mapped approach, keeping information used by the Kernel managers in the process address space. Thus, the state of a process includes program variables and dynamically allocated blocks, as well as Kernel data. A more detailed description of these aspects is beyond the scope of this paper. We remark that, in the current version, the LVM does *not* address the problem of maintaining global consistency in reversing cooperating processes, which would require causality tracking. This will be a topic of further research.

6.5 Using the Log Manager. Maintaining the history a process computation in the LVM is very easy. A debugger can let the Log Manager start recording the history of another process by just calling the `KStartRecording` system call, which makes an initial checkpoint for that process. To make successive checkpoints at the desired level of granularity, the debugger can invoke the `KLogState` system call. Restoring the state corresponding to the previous checkpoint simply requires a call to `KRollBack`.

7 Writing a director: cache simulator

In this section we show that using LVM system calls we can write a director with just a few lines of C code: Figure 3 shows the implementation of a simple toy cache simulator, which counts the number of cache misses that a program would generate using a memory system with a single-line 4KB cache. To accomplish this task, the simulator first loads the process to be analyzed using the `KNewProc` system call, and then registers itself as a listener of memory access events emitted by that process, specifying a suitable handler (`_MemAccessH`). The monitored process, which is initially idle, is then started using `KStartProc`, and the simulator is put to sleep with `KWaitSyncEvtH` until the process is run to completion. The LVM event handling mechanism guarantees that the `_MemAccessH` handler is activated every time the monitored process makes a memory access in its logical address space. Parameters passed to `_MemAccessH` by the Event Manager specify address and size of the accessed segment. The handler maintains the base address of the currently cached memory frame (`sB`), and a count of memory accesses (`sMems`) and cache misses (`sMiss`). As a final step, the simulator dumps the final counting results. We remark that even a simple application like this one may be extremely difficult to realize in conventional computing environments.

```
#include <lvm.h>

#define PROGRAM     "queen.leo"
#define LINE        4096

static ui4 sMems;   /* counter of mem accesses */
static ui4 sMiss;   /* counter of cache misses */
static ui4 sB;      /* current line address    */

void _MemAccessH(ui4 inBase, ui4 inSize) {
    ui4 theB1 = inBase/LINE;
    ui4 theB2 = (inBase+inSize-1)/LINE;
    if (sB != theB1) { sB = theB1; ++sMiss; }
    if (sB != theB2) { sB = theB2; ++sMiss; }
    ++sMems;
}

void main(){

    /* create new process */
    ui4 thePID = KNewProc(PROGRAM, 0, 0);
    if (thePID == KINVALID_PID) {
        KPuts("[CacheSim:] Can't load program\n");
        return;
    }

    /* install mem access handlers */
    KNewAsyncEvtH(thePID, KMEM_READ_EVENT,
                  0, (KEvtHT)_MemAccessH);
    KNewAsyncEvtH(thePID, KMEM_WRITE_EVENT,
                  0, (KEvtHT)_MemAccessH);

    /* start process and wait for termination */
    KStartProc(thePID);
    KWaitSyncEvtH(thePID, KHALT_EVENT, 0, NULL);

    /* output results */
    KPuts("[CacheSim:] # Mem accesses = ");
    KPrintlnUI4(sMems);
    KPuts("[CacheSim:] # Cache misses = ");
    KPrintlnUI4(sMiss);
}
```

Figure 3: Code of a simple cache simulator.

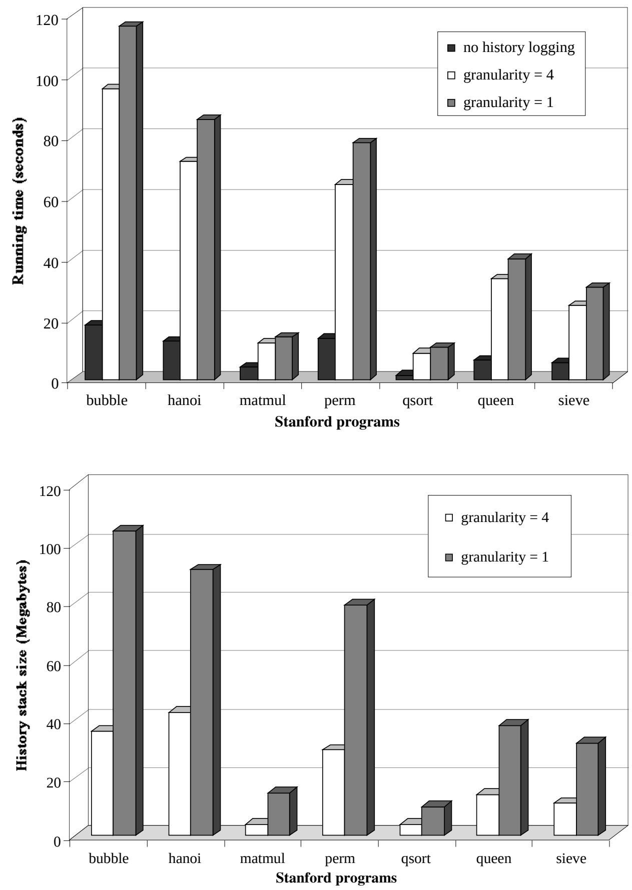

Figure 4: Time and space overhead of history log creation on the Stanford integer benchmark suite for granularities 1 and 4.

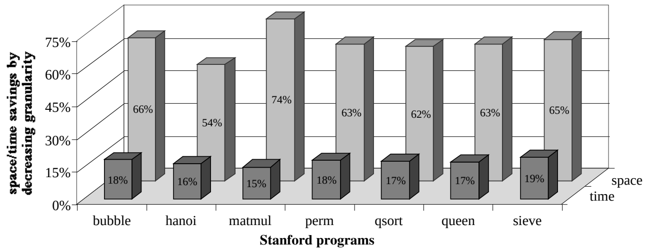

Figure 5: Reducing granularity from 1 to 4 saves approximately 60% of the history stack size and 18% of the running time for the Stanford programs.

8 Experimental evaluation

In this section we report the results of a preliminary experimental investigation aimed to evaluate the time/space overhead of supporting rollback to previous execution states and of running programs in the LVM under the control of a director. In both cases substantially more time than normal execution may be required. This is especially true when profiling memory accesses, whose trace data may run into gigabytes even for small programs. For instance, programs monitored with the `cache` analysis tool produced by ATOM [22] may be slowed down by a factor of 11. A similar slowdown is observed in [3] for supporting reversibility using program instrumentation at assembly code level.

We first analyze the time and space overhead of history log creation. We conducted our preliminary experiments on a few toy programs from the Stanford integer benchmark suite, which provide a variety of different programming patterns. Parameters settings for the Stanford programs are as follows: `bubble` on 1000 items, `hanoi` on 18 discs and 3 towers, `matmul` on 64×64 matrices, `perm` on 8 numbers, `qsort` on 5000 items, `queen` with 10 queens, `sieve` on primes less than 100000. Times were measured using the `getrusage()` system call on a Macintosh Powerbook Titanium with a PowerPC G4 at 500 MHz and 384 MB RAM running MacOS X. The LVM was compiled using `gcc` with optimization level $O4$. Figure 4 shows the time and space overhead for log creation with granularities 1 and 4 (i.e., a checkpoint every `leave` / every four `leave` instructions). The worst case clearly corresponds to granularity 1: programs are slowed down approximately by a factor of 6 and the history log size may be quite large even on medium-size instances. Both time and space are substantially reduced when granularity is decreased from 1 to 4: as shown in Figure 5, the savings are around 18% and 60%, respectively.

Interestingly, programs that exhibit more temporal locality in memory write operations can benefit at a larger extent of granularity reduction. The chart in Figure 6 is related to two very simple programs that compute the sum of the elements in the same column of a 256×256 matrix and store the results in an array: `LoopA256` scans the matrix by row, while `LoopB256` scans the matrix by column. In `LoopB256`, array items are repeatedly updated, while in `LoopA256` an array item is written only every 256 iterations. Thus, `LoopB256` exhibits much more writing temporal locality than `LoopA256`, yielding substantial space reductions.

To evaluate the overhead of event handling, we considered memory accesses, which are by far the most frequent events. Preliminary experiments show that the execution of a program whose memory accesses are monitored by a director may be as high as 8 times slower than normal execution.

9 Conclusions and work in progress

We have presented the Leonardo Virtual Machine, an abstract machine with advanced facilities for writing program directors. Our work is foundational to the rapid development of tools for program debugging, directing, and visualizing. In the LVM, the controlling/monitoring portion of a director, which may be

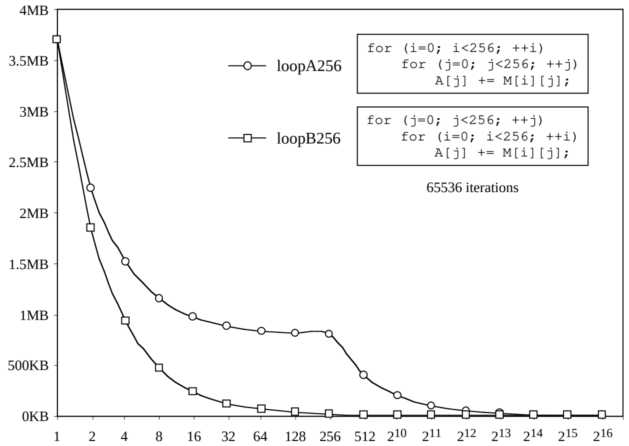

Figure 6: History stack size for test programs `LoopA256` and `LoopB256` and log granularity ranging from 1 to 65536.

extremely difficult to realize in conventional computing environments, can be written with just a few lines of C code. The LVM is fully platform-independent, supports a flexible event-handling scheme, and features built-in reversible computing capabilities.

We are currently extending the LVM Kernel in several directions. A relevant extension is a GUI Manager that supports advanced system calls for graphic user interface handling. We also believe that further optimizations could be added, yielding substantial time/space improvements in the LVM performances. The interested reader can find up to date information about the LVM at [24].

Acknowledgments. We wish to thank Francesco Quaglia for many useful discussions.

References

[1] H. Agrawal, R. DeMillo, and E. Spafford. An execution-backtracking approach to debugging. *IEEE Software*, pages 21–26, 1991.

[2] M.H. Brown. Exploring algorithms using Balsa-II. *Computer*, 21(5):14–36, 1988.

[3] S. Chen, K. Fuchs, and J. Chung. Reversible debugging using program instrumentation. *IEEE Transactions on Software Engineering*, 27(8):715–727, 2001.

[4] J. Choi and H. Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the ACM SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 48–59, 1998.

[5] Microsoft Corp. `http://www.microsoft.com/ net/`.

[6] P. Crescenzi, C. Demetrescu, I. Finocchi, and R. Petreschi. Reversible execution and visualization of programs with Leonardo. *Journal of Visual Languages and Computing*, 11(2):125–150, 2000. The system is available at the URL: `http://www.dis.uniroma1.it/~demetres/Leonardo/`.

[7] M. Golan and D. Hanson. DUEL: a very high-level debugging language. In *Proc. Winter USENIX Technical Conference*, pages 107–117, 1993.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading MA, 1997.

[9] K.J. Gough. Stacking them up: a comparison of Virtual Machines. In *Proceedings 6th Australasian Computer Systems Architecture Conference*, pages 55–61. IEEE Computer Society Press, 2001.

[10] K.J. Gough and D. Corney. Evaluating the Java Virtual Machine as a target for languages other than Java. In *Proceedings Joint Modula Languages Conference*, LNCS 1897, 2000.

[11] S.L. Graham, P.B. Kessler, and M.K. McKusick. Gprof: a call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, 1982.

[12] D.R. Hanson. Event association in SNOBOL4 for program debugging. *Software - Practice and Experience*, 8(2):115–129, 1978.

[13] C.L. Jeffery and R.E. Griswold. A framework for execution monitoring in Icon. *Software - Practice and Experience*, 24(11):1025–1049, 1994.

[14] J.R. Larus and T. Ball. Rewriting executables files to measure program behavior. *Software - Practice and Experience*, 24(2):197–218, 1994.

[15] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, 36(4):471–481, 1987.

[16] H. Lieberman. The Debugging Scandal and What to Do About It. *Comm. of the ACM*, 40:27–29, 1997.

[17] H. Lieberman and C. Fry. ZStep95: A Reversible, Animated Source Code Stepper. In J.T. Stasko, J. Domingue, M.H. Brown, and B.A. Price, editors, *Software Visualization: Programming as a Multimedia Experience*, pages 277–292. MIT Press, 1997.

[18] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading MA, 1997.

[19] T.G. Moher. PROVIDE: a process visualization and debugging environment. *IEEE Transactions on Software Engineering*, 14(6):849–857, 1988.

[20] B.A. Myers. Incense: a system for displaying data structures. *Computer graphics*, 17(3):115–125, 1983.

[21] R. Sosič. The Dynascope directing server: design and implementation. *Comp. Systems*, 8(2):107–134, 1995.

[22] A. Srivastava and A. Eustace. ATOM: a system for building customized program analysis tools. In *Proceedings of the ACM Symposium on Programming Languages Design and Implementation (PLDI'94)*, pages 196–205, 1994.

[23] J.T. Stasko. TANGO: A Framework and System for Algorithm Animation. *IEEE Comp.*, 23:27–39, 1990.

[24] The Leonardo Computing Environment. WWW site: `http://www.leonardo-vm.org`.

[25] A. Tolmach and A.W. Appel. A debugger for standard ML. *Journal of Functional Programming*, 5(2):155–200, 1995.

[26] J.S. Vitter. US&R: a new framework for redoing. In *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symp. Practical Software Development Evironments*, pages 168–176, 1984.

[27] M.V. Zelkowitz. Reversible execution. *Communications of the ACM*, 16, 1973.