VMAD: a Virtual Machine for Advanced Dynamic Analysis of Programs

Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, Philippe Clauss

▶ To cite this version:

Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner, Philippe Clauss. VMAD: a Virtual Machine for Advanced Dynamic Analysis of Programs. [Research Report] 2010, pp.10. inria-00534748

HAL Id: inria-00534748
https://inria.hal.science/inria-00534748

Submitted on 24 Feb 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

VMAD: a Virtual Machine for Advanced Dynamic Analysis of Programs

Alexandra Jimborean, Matthieu Herrmann, Vincent Loechner and Philippe Clauss
INRIA Nancy-Grand Est (CAMUS), LSIIT Lab., CNRS
University of Strasbourg, France
{alexandra.jimborean, vincent.loechner, philippe.clauss}@inria.fr, maherrmann@unistra.fr

Abstract

In this paper, we present a virtual machine, VMAD (Virtual Machine for Advanced Dynamic analysis), enabling an efficient implementation of advanced profiling and analysis of programs. VMAD is organized as a sequence of basic operations where external modules associated to specific profiling strategies are dynamically loaded when required. The program binary files handled by VMAD are previously instrumented at compile time to include necessary data, instrumentation instructions and callbacks to the VM. Dynamic information, such as memory locations of launched modules, are patched at startup in the binary file. The LLVM compiler has been extended to automatically instrument programs according to both VMAD and the handled profiling strategies. VMAD's potential is illustrated by presenting a profiling strategy dedicated to loop nests. It collects all memory addresses that are accessed during a selected number of successive iterations of each loop. The collected addresses are consumed by an analysis process trying to interpolate addresses successively accessed through each memory reference as a linear function of some virtual loop indices. This profiling strategy using VMAD has been run on some of the SPEC2006 and Pointer Intensive benchmark suites, showing a very low time overhead, in most cases.

1. Introduction

Runtime code analysis and optimization is becoming a main strategy used to face the ever extending and changing variety of existing processor architectures and execution environments that an application can meet. Unlike static compilers, that have to take conservative decisions from restricted available information extracted from the source code, runtime profilers and optimizers lie on information extracted at execution time. While nowadays processors are providing more and more computing resources at the price of more and more usage complexity – particularly with the advent of multi/many-core processors – efficient program optimizations request accurate and advanced runtime analyses. However, such analyses inevitably incurs time overhead that has to be minimized.

Many tools for the instrumentation of programs exist: Valgrind [10], DynamoRIO [3], PIN [9], Dyninst [4], PEBIL [6], Strata [5], but have strong limitations in the kind of possible instrumentations and analyses that can be implemented without reaching a huge runtime overhead. This is mainly due to the fact that all these tools, excepting PEBIL, use software dynamic translation (SDT) to handle the insertion of instrumenting instructions. This approach necessarily introduces an inevitable runtime overhead. These tools are discussed further in the related work section.

In this paper, we present VMAD, a virtual machine (VM) handling x86_64 binary files especially tailored at compile time to include instructions and data related to the VM functioning. VMAD provides the following features:

- instrumentation management and analysis phases are separated modules that are loaded only if requested;
- these modules can be either generated offline or at runtime thanks to a dedicated just-in-time compiler module;
- they can be activated or deactivated at any time during the target application run;
- several instances of the same module can be run simultaneously while targeting different parts of the application code;
- instrumentations and analyses can include any kind of tasks and their time and space scopes can be managed accurately;
- the VM is able to switch between several versions of the same code extract in order to avoid any over-

head due to unnecessary instrumentation instruction runs.

For the binary code to be tailored for the VM, the compiler handles specific pragmas that have to be inserted in the source code to delimit the code regions of interest and to specify the required analysis. We have extended the LLVM compiler [7] to handle such pragmas. This extension allows the developper to initiate dynamic low-level analyses that focus on specific parts of the source code in order to explain runtime bottlenecks. To our knowledge, VMAD is the first proposal providing low-level instrumentation initiated from the source code with almost negligible runtime overhead.

To show VMAD's potential, we present one advanced instrumentation and one analysis processes that have been implemented. The instrumentation, which is dedicated to loop nests, consists in collecting all memory addresses that are accessed during a selected number of successive iterations of each loop of the nest. This instrumentation is quite specific since it has to occur only on some non contiguous phases of the loop nest execution, *i.e.*, only when the memory accesses are occurring while the enclosing loops are all executing iterations that have been selected to be instrumented.

The analysis process makes use of the accessed addresses collected through the instrumentation. It tries to interpolate addresses successively accessed through each memory reference as a linear function. For performance reasons, it runs while the instrumentation occurs by consuming each new collected address and verifying that it fits a linear function of some virtual loop indices.

The implementation of such instrumentation and analysis processes involves to extend the LLVM compiler to handle a new pragma, by developing a number of dedicated program transformation passes. Obviously it also involves developing the modules that will be loaded by the VM and mostly defining the control and management of an instrumentation, or defining the computations of some behaviour modelling strategy when the module implements an analysis.

The remainder of the paper is organized as follows. In section 2, we present an overview of our static-dynamic framework. VMAD is detailed in section 3, while the static preparation phase of the code is presented in section 4. The instrumentation and analysis processes used to show our system abilities are described in section 5. Finally, we give some benchmark results in section 6, related work in section 7 and we conclude in section 8.

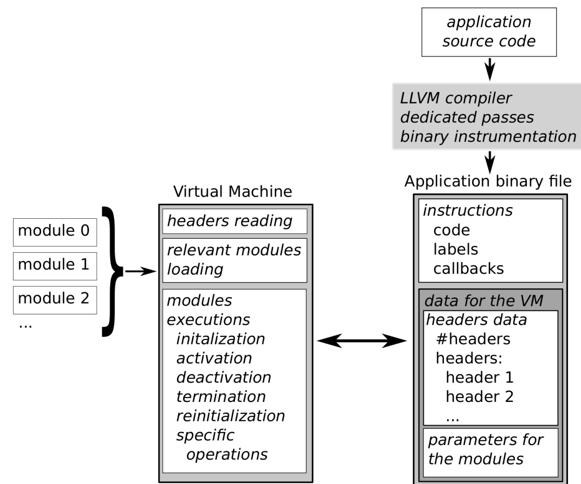

Figure 1: Framework overview.

2. Framework overview

We consider as an "instrumentation process" a process attached to a particular kind of original instructions in order to monitor their run – as for instance a process dedicated to collect all addresses accessed through any memory instruction of a code extract. An "analysis process" is devoted to use information collected by an instrumentation process for some computations – as for instance to model the program behavior as a graph. In the following, we use the term "profiling process" to denote either an instrumentation or an analysis process.

VMAD has been built by taking great care of performance and runtime overhead. Hence it does not use any software dynamic translation that would delay the run of the input program as it is done in Valgrind [10], DynamoRIO [3], Pin [9], Dyninst [4] or Strata [5]. Further, instrumentation instructions are not inserted on-the-fly by replacing some NOP instructions that have been previously inserted at compile time, as it is done with PEBIL [6]. Rather, several copies of the same code extracts are built at compile time, each copy corresponding to a phase in the whole program run in which the profiling processes will either operate fully, partially or be completely deactivated. The price to pay with this approach is the larger size of the program binary file. However, great care can also be taken of minimizing the size of the copies by inserting branches to the original code whenever possible. Beside performance, another noticeable benefit with this approach is the opportunity to create any advanced instrumentation for which the related instrumented copy can be far different than the original code.

The static-dynamic collaborative framework is depicted in figure 1. At compile time, the C/C++ source

code with some dedicated pragmas is translated into the LLVM intermediate representation (IR) with additional specific metadata. A LLVM pass creates copies of targeted code extracts where instrumentation instructions are inserted only if convenient information can be found in the IR. This is not always possible. For example, in the LLVM IR, all data accesses are made from an infinite set of registers, and memory accesses can only be actually identified in the final assembly/binary code. Further, the number and kind of memory accesses are obviously dependent on the optimizations applied by the compiler – particularly register allocation. Hence some instrumentations can only be inserted in the final assembly code during a devoted phase of the compiler backend. During this phase, some additional code and data are inserted which will allow the VM to take control of the profiling processes and also to pass the requested parameters to the relevant modules.

Besides instrumentation instructions and labels attached to some targeted instructions and control structures – *e.g.*, memory accesses, loops – additional code can consist in some *decision blocks* providing a way to toggle between instrumented or non-instrumented code snippets, following some conditionals. Such blocks can also contain some updates of variables used by the current profiling process, as for instance a counter providing the number of times an instrumentation has already occurred. Some callbacks are also inserted in the code in order to invoke VMAD and its related modules when necessary.

Inserted data, organized as headers, will inform the VM about the kind of profiling process for which the input program has to be managed. It also provides all necessary information as addresses in the code where instrumentations occur, values and addresses to be patched, pointers to the relevant parameters, ... Common symbols are used by both the VM and the binary file in order for the VM to recover all necessary entry points in the binary file thanks to the dynamic linker.

At runtime, we use `LD_PRELOAD` to load VMAD's dynamic shared library at the startup of the handled program, to provide its own version of the C-library entry point *__libc_start_main*. It allows VMAD to first be set up by reading all relevant information in the input program binary file.

VMAD first scans all headers found in the input binary file. Then VMAD loads the required modules and makes some updates by patching the binary code. Each module grants write permissions to itself before patching. At this step, the input program starts its run. When needed, VMAD is activated through callbacks that have been inserted in the code at compile time.

```
// backup of the stack red zone:
sub     $0x80,%rsp
// backup of the scratch registers:
...
// stack adjustement (x86_64 convention):
mov     %rsp,%rbp
mov     $0xfffffffffffffff0,%rsi
and     %rsi,%rsp
// move 0 to %rax (amd x86_64 convention)
mov     $0x0,%rax
// registers for the call
// $0x0 will be patched:
mov     $0x0,%rdi // address of the module
mov     $0x0,%rsi // address of the operation
// function call:
// 1st parameter = rdi (convention)
// 2nd parameter = rsi
callq   *%rsi
mov     %rbp,%rsp // stack readjustement
// restoration of the scratch registers:
...
// restoration of the stack red zone:
add     $0x80,%rsp
```

Figure 2: callback in x86_64 assembly code.

3. The virtual machine VMAD

As described in the previous section, VMAD makes use of three kinds of specific information that is inserted at compile time in the program binary file:

- instrumentation code that has been inserted before or after original instructions in order to monitor their run;
- blocks of instructions dedicated to control the global behavior of the profiling process, organized as *decision blocks* and containing callbacks to invoke some VM operations defined in modules;
- some data corresponding to module parameters, pointers and memory addresses.

Each instrumentation collaborates with a dynamic module. Both share information using a fixed size header previously inserted in the binary file. At startup, as soon as headers and associated parameters have been read from the input binary file, the VM loads the relevant modules and instantiates them. Parameters fetched at this point are *static data*, and are accessed either by the VM – for instance to know which modules are required – or by modules that have been loaded by the VM – for instance to get memory addresses that have to be patched – or by instrumenting instructions – for instance to allocate memory for register savings. On the other hand, *dynamic data* is accessed through pointers

set up by the relevant VM modules which allocate the memory they require.

Each module is structured at least with five main entry points: *init* which role is to instantiate a profiling process, *quit* to kill such an instance, *on*, *off* and *reset* to activate, deactivate and reset a profiling process. Additional operations can also be provided by a module. These operations are invoked thanks to callbacks previously inserted in the input binary code and patched by *init* in order to point to the relevant instance of the module and operation. These callbacks are inserted at some control points in the program binary file and have the common form showed figure 2.

As soon as the instrumented program has been launched, VMAD performs the following operations.

1. Thanks to a commonly defined symbol, it reads all the headers in the program binary file. For each header:
   (a) VMAD gets the type of instrumentation it will have to manage;
   (b) it gets all the necessary parameters;
   (c) it loads the module related to the type of instrumentation;
   (d) it updates the module pointer inside the header.
2. Each loaded module performs the following operations:
   (a) it allocates the memory it requires and updates the related pointer inside the header;
   (b) since by default, the program is able to run without the VM, it patches the code in order to branch to instrumented code;
   (c) it patches all the callbacks in the code to point to the convenient functions of the module.
3. VMAD calls the original *libc_start_main*.

After this last point, the program runs and VM operations are only invoked through callbacks. Hence deactivation – branching to non-instrumented code – and profiling ending are decided from decision blocks and invoked VM operations. Reactivation can be decided from a delayed signal, whose associated handler patches the code in order to branch to instrumented code.

4. Preparing the code at compile time using LLVM

Code analysis and optimization starts, in our approach, at the level of the original source code. The programmer may guard regions of code with specific pragmas, aimed to identify the code sections to be processed.

Our work relies on the LLVM compiler and Clang front-end, which has to be extended to handle the newly introduced pragmas. Hence, the steps required for the clang front-end to be aware of a new pragma imply first to augment the parser, by defining a new class for the pragma. The class contains a specification for the pragma handler, which describes the action to be taken, once the compiler encounters the pragma in the source code. Shortly, the handler verifies the syntax of the pragma and calls the corresponding action. This has to be available in the list of actions taken by the compiler and to contain the semantics of the pragma.

For instance, a pragma may be attached to a specific structure such as a compound statement, a function or a loop. In this case, the action means marking this certain structure with a symbol indicating the existence of the pragma. In this respect, we have defined a new data structure *PragmaCollector* which keeps track of all the associated pragmas. The next step is code generation. As we are extending the LLVM compiler, the source code is converted into LLVM IR. The decision to be taken at this point is whether to create a new instruction for the pragma, to use LLVM intrinsics, annotated attributes or the LLVM metadata[1]. Due to various disadvantages presented by the first three options, we selected the last one: metadata information is inserted in the generated code to mark the presence of the pragma.

In what follows, we add a LLVM pass taking the corresponding actions, based on the metadata. In the case of instrumentation, we want to create different versions of the code, namely original and instrumented. The approach is to select the code carrying the metadata information and to duplicate it. One of the versions remains in the original format, with minimal alterations, while the second version is enhanced with instrumentation code. Depending on the type of instrumentation, the LLVM IR may not contain sufficient information. Should one aim to detect dependencies between data structures or to model the code, such as to interchange loops, the LLVM IR offers great support and malleability. Processing the code represented in LLVM IR is advisable when higher level information is required.

Although the LLVM IR provides interesting information, low level instrumentation remains impossible for several performance related mechanisms, such as tracing memory behaviour. This is due to the fact that LLVM IR uses an infinite number of virtual registers, which will be later mapped either to physical registers or to memory locations. As registers are not yet allo-

[1]Metada information can be used since LLVM version 2.6

cated in the LLVM IR, one cannot distinguish whether the LLVM “load” and “store” instructions represent memory or register accesses. Additionally, LLVM IR is in static single assignment (SSA) form, which simplifies the analysis of the control flow graph, but introduces a number of unnecessary “load” and “copy” instructions. These are eliminated when generating the code for a specific target architecture. Also, SSA PHI instructions are not supported by traditional instruction sets, hence the compiler replaces them with instructions preserving their semantics, but which are not present in the LLVM bytecode [8]. Under these circumstances, one needs to convert the LLVM bytecode to be instrumented into assembly code.

The code which is not instrumented is represented in the LLVM IR, conserving the higher level information for further compiler passes. On the other hand, regions marked for instrumentation are cloned and extracted into new functions. They are analyzed instruction-wise and a specific macro is inserted before each of the tackled instructions. Macros will be expanded in instrumenting code.

Once various versions of the code have been generated, a mechanism to choose one or another for execution is compulsory. In this respect, the code has to be prepared to interact with the VM, as the execution flow is dynamically guided. Versions of code are preceded by a decision block, containing calls to the VM with parameters describing the code structure. At runtime, the VM selects one code version, based on the current values of the parameters, and patches the code accordingly. During the execution, additional calls to the VM are performed to provide information about the current status. These are specific to the targeted type of instrumentation and are strategically placed at the key points of the program. The callbacks to the VM represent a compromise between performance and generality, as pointers to the adequate module functions are patched at start-up. On the other hand, it ensures the genericness of our framework, allowing the VM to handle a multitude of actions.

As the code is patched dynamically, at compile time one has to mark points of high importance such as beginning and end of cloned and instrumented regions, or decision blocks. Also, the original flow is modified statically to include newly inserted structures and calls. Our strategy is to insert these code snippets in x86_64 assembly code, to ensure that no modifications appear in the last phases of code generation. Furthermore, specific instructions are coded in hexadecimal representation, on the precise number of bits required for patching. In figure 2, the two lines marked as *address of the module* and *address of the operation* are illustrated, for

```
# number of headers.
global vmad_headers_nb
vmad_headers_nb:
        .long   1

#list of headers
. global vmad_headers
vmad_headers:

. global vmad_0_entry_lb
vmad_0_entry_lb:
        .quad   vmad_0_entry
        .quad   vmad_loop_entry
        .quad   vmad_0_param

#list of parameters
. global vmad_0_param
vmad_0_param:
       .long   3
       .quad   vmad_0_end_original
       .quad   vmad_0_loop_reinstru
       .quad   vmad_0_instru_call
```

Figure 3: Headers and parameters.

clarity purposes, as written in x86_64 assembly code, whereas they are actually inserted in the semantically equivalent hexadecimal form, as shown in figure **??**. The latter representation allows an accurate control of the exact number of generated bits, which is crucial for dynamic code patching. In our example, both instructions `Mov $0, %rdi` and `Mov $0, %rsi`, would each represent $0 on only 8 bits. However, VMAD requires 64 bits to be allocated for patching the addresses, since they can be 64-bits pointers. Therefore, the compiler must insert these instructions in hexadecimal form.

In addition to preparing the code, a number of headers and parameters are annexed to the generated binary code. The list of headers is specific to the type of instrumentation, as they determine the modules to be loaded in the VM. Moreover, they are linked to the corresponding parameters, containing higher level information statically available, but which would be prohibitively time-expensive to identify in the binary representation. At the same time, the compiler transmits as parameters instrumentation specific information, such as addresses of the code snippets added to the original code.

As illustrated in figure 3, the list of headers is introduced by its length, given by *vmad_headers_nb*. Each header has a fixed size containing three entries: the unique ID of the instrumentation, the type of the instrumentation, the address of associated parameters.

The list of parameters may have a varying length and include information such as labels indicating sections of code which are patched by the VM, or specific details, for instance characteristics of a loop, its depth or parent loop. The parameters may differ depending on the instrumentation type.

All in all, any compiler can be adapted to interact with the VM, following the steps presented above, to

provide the binary code in a specific format, accompanied by the list of headers and parameters.

5. Instrumenting loop nests and analyzing memory accesses

We underline VMAD's features by creating a framework tailored to emphasize its complexity. It consists in instrumenting and profiling the memory accesses of critical pieces of code –loop nests– in a minimal amount of time overhead. We focus on loop nests, as they represent the major part of the whole execution time of compute-intensive applications. The goal of our instrumentation framework is to collect the memory addresses that are accessed during samples of the run iterations and, if possible, compute a linear function interpolating their values. Additionally, the loop trip counts of each loop, except the outermost, are also collected for a few runs to be linearly interpolated. Such a profiling is particularly useful for nested *while*-loops accessing memory through indirect references or pointers. If linear modelling of the memory accesses and of the loop trip counts is obtained, such loop nests can then be optimized and parallelized using *for*-loops dedicated approaches as the polyhedral model [1, 2].

5.1. Loop nest instrumentation

Efficient and advanced loop nest instrumentation consists in profiling only a subset of the executed iterations of each loop. The complexity of the method is outlined in the case of nested loops, as instrumentation depends not only on the iteration of the current loop, but also of the parent loops. For a thorough understanding, consider the loop nest in figure 4. The first three iterations of each loop are instrumented. One may easily notice that instrumented and non-instrumented iterations alternate, hence the execution has to switch from one code version to another at runtime. Once the outermost loop profile has been completed, the execution can continue with a non-profiled version of the loop nest, thus inducing no overhead for the remaining iterations.

For linear interpolation of memory accesses we have to profile the first three iterations of each loop. The first two are dedicated to collecting data and computing the affine function coefficients, while the third iteration provides data to verify the interpolation correctness. As an extension, one may consider beneficial to set the instrumentation on/off instruction-wise. As in the case of a loop containing an if-then-else structure, sufficient information is acquired in three iterations concerning a subset of the monitored instructions, but this might be not enough for some conditionally executed instructions. Therefore, one may consider instrumenting only these instructions in subsequent iterations, avoiding unnecessary overhead. Similarly, once the instrumentation phase ended, it may be later restarted by the VM, if required.

Statically, our LLVM pass creates copies of the critical loop nests, extracts them in new functions and converts to x86_64 assembly code. A second pass analyses the functions and precedes each instruction accessing memory with instrumentation code that computes the actual memory location being accessed, and makes a call to the VM to transmit the data collected. Figure 5 illustrates the structure of the code of figure 2 and the links between different versions. Blocks O_i, O_j and O_k represent the original version of the code, while I_i, I_j and I_k represent the instrumented bodies of each loop. The instrumented and original versions are connected together at their entry point, where a choice is made at runtime deciding which version to run, based on the values of the virtual iterators. One decision block is associated to each loop, represented by D_i, D_j and D_k, correspondingly. More precisely, if $i = 1$, the value of iterator i allows instrumentation, therefore its body will be instrumented, block I_i. But if $j = 3$, the body of the second loop will be non-instrumented (O_j) as well as the body of its subloop (O_k). The same strategy is applied to the other iterations as well. As a general rule, if a parent loop is non-instrumented, all its subloops will be non-instrumented. On the other hand, if the parent loop is instrumented, its subloops may or may not be instrumented, depending on the value of their own iterators. To handle the trip counts of the considered loops unitary, either *while*-loops or *for*-loops with more than one exit point, we introduce "virtual iterators". They are maintained to mirror the actual number of executed iterations.

At compile time, we mark the beginning and the end of original and instrumented versions of the loop nests with labels, which are appended in the list of parameters given to the VM. Additionally, callbacks to the VM are performed:

- in each decision block – to decide the version to execute;
- at the end of each instrumented iteration – to send the acquired data to the VM for processing;
- at the end of the loop nest – to inform the VM that its execution has finished.

Lastly, a list of headers and parameters is prepared, notifying the VM regarding the modules required for this instrumentation: module *vmad_loop*, *vmad_gather_memory_addresses* and

```
loop i          i = 0, 1, 2
  ...               j = 0, 1, 2
                        k = 0, 1, 2   --> instrumented
  loop j                k = 3, ...    --> non-instrumented
    ...
                    j = 3, ...
    loop k              k = 0, ...    --> non-instrumented
      ...
    ...         i = 3, ...
                    j = 0, ...
                        k = 0, ...    --> non-instrumented
```

Figure 4: Loop nest instrumentation.

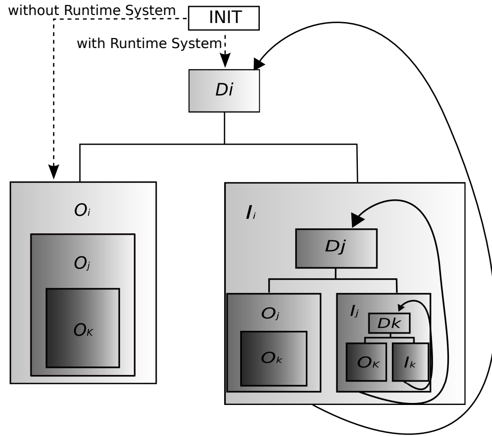

Figure 5: Code structure.

vmad_interpolation. One instance of each of these modules is created per loop, at runtime. The first module encloses the mechanisms necessary for handling loops, the second one collects the memory accesses performed inside the loops, while the last module performs the interpolation. Each module depends on the previous one to complete its task, however, their implementation is scalable in the same time. The profiling process can easily be extended with various other instrumentations, as the loops framework may be used with new purposes, independent on the other modules. As well, the memory locations acquired in the *vmad_gather_memory_addresses* may be used as foundation of new types of analysis.

The list of parameters contains specific information, such as addresses of the code to be patched at startup, the structure of the loop nest or memory locations accessed from the loop body.

At runtime, the VM parses the list of headers, loads the solicited modules and patches the code to enable instrumentation. During execution, it is triggered by the callbacks to select among versions of code and to process the information collected from instrumentation.

5.2. Analyzing memory accesses

Our instrumentation tracks the memory accesses performed inside loops and computes associated affine functions of the loop bounds, interpolating the accessed addresses, when possible.

Since the number of accessed locations can be very high, considering a memory intensive loop nest, it is recommended that the acquired data is processed immediately by the interpolation process, rather than stored for a later utilization.

For each instrumented loop, a buffer is created at compile time, to (re)store the state of the machine before the interpolation process. At runtime, the VM allocates space to be populated dynamically with the accessed memory locations and, for each loop, it patches the corresponding address to store collected information. Additionally, it creates a stack per loop nest and pushes a structure to accommodate the function coefficients together with the depth of the embedding loop, for each of the instructions accessing the memory and for the loop bounds. Hence, each loop pushes on the stack sufficient space for:

- coefficients of the functions interpolating on the subloops upper bounds,
- coefficients of the interpolation functions on the memory locations accessed by the instructions contained in the loop nest; for each instruction, these coefficients correspond to the indices of the enclosing loops, plus a constant.

As the instrumented iterations of a subloop are executed, the VM reads the values of the memory locations from the designated buffer and the corresponding function coefficients are computed and stored in the associated positions. As soon as the execution of the subloop ends, its structure is popped and the values of the coefficients, as well as the total number of iterations of the subloop, are propagated in the structure of the parent loop.

As a new iteration of the parent loop begins, the VM pushes a new structure on the stack and computes the new memory access function. Communication with the VM is achieved by means of a dirty flag, which indicates that a new memory location is available in the buffer.

In the same manner, the process is repeated until all loops finish their execution. At this time, all the coefficients have been computed and the functions verified for linearity.

```
# pragma instrument_mem_add{
   for (mrA = groupA.head, mrPrevA = NULL;{
                   ....
       for (mrB = groupB.head, mrPrevB = NULL;
            mrB != NULL;
             mrPrevB = mrB, mrB = (*mrB).next) {
                    ...
    gp = D[(*mrA).module] + D[(*mrB).module]
                              - CAiBj(mrA, mrB);
                    ....
                    if (gp > gpMax) {
                       gpMax = gp;
                       maxA = mrA; maxPrevA = mrPrevA;
                       maxB = mrB; maxPrevB = mrPrevB;
                    }
```

```
  0 |  0 | 140736985202616 |
  0 | 64 | 24881880 |
  0 | 140736985202632 |
64 | 0 | 24881848 |
  4 | 0 | 6345856 |
  4 | 6345896 |
  0 | 0 | 140736985202244 |
  0 | 0 | 4234836 |
  0 | 0 | 140736985202244 |
  0 | 0 | 140736985202572 |
  0 | 0 | 140736985202568 |
  0 | 64 | 24881872 |
  0 | 0 | 140736985202616 |
```

Figure 6: Code extract from `ks` and its corresponding interpolation functions.

6. Experiments

In this section, we present the results of our experiments running the interpolation of memory accesses in loops. We targeted all C codes from the SPEC CPU 2006 benchmark suite [12] and four codes from the Pointer-Intensive benchmarks [11]. LLVM with the Clang front-end cannot handle Fortran codes and all C++ codes are Objective-C codes that we failed to compile with LLVM. We also failed in compiling the `gcc` benchmark using our own `Makefile`, since SPEC requires some specific flags to be set. We added our pragma in the source codes for some loop nests in some of the most time consuming functions [13]. We ran the benchmarks using the `ref` input files to compute VMAD's runtime overhead, and using the `test` input files to get output files with the interpolation results, since runs using the `ref` files would have produced a huge amount of data for this instrumentation. The execution platform is a 3.4 Ghz AMD Phenom II X4 965 micro-processor with 4GB of RAM running Linux 2.6.32.

Our measurements are shown in table 1. We ran each program in its original form and in its instrumented form to check the runtime overhead induced by using VMAD. For each instrumented loop nest, the dynamic profiling is activated each time its enclosing function is invoked. For each program, the second column shows VMAD's runtime overhead, the third column shows the functions in which loop nests were instrumented, the fourth column shows the total number of instrumented loops per function, the fifth column shows the total number of instrumented memory instructions in the program, the sixth column shows the total number of times instrumented memory accesses ran effectively and finally, the last column shows the number of memory accesses that were identified to be linear.

For most programs, VMAD induces a very low runtime overhead which is even negligible for `perlbench`, `bzip2`, `milc`, `hmmer`, `h264ref` and `lbm`. For the programs `sjeng` and `sphinx3`, the significant overheads are mainly due to the fact that the instrumented loops execute only few iterations, but they are enclosed by functions that are called many times. Thus all iterations are run while being fully instrumented since each call represents a very low execution time. However, the profiling strategy could be improved in order to consider specifically such cases by deactivating the instrumentation after a few calls. Program `milc` is showing an opposite behavior since a few memory instructions are run a lot of times. In such a case the runtime overhead is quite low. For the Pointer-Intensive benchmarks, the execution times are too small – the order of milliseconds – to get relevant overhead measures: either a large runtime overhead is obtained since VMAD inevitably induces a fixed minimum overhead (`bc`), or even a speedup is obtained (`ft`).

We also noticed that this particular instrumentation process makes any program's binary file larger than the original version with 400 more bytes per instrumented memory instruction, on average. For instance, program `milc` instrumented version is about 267 kbytes versus 191 kbytes for the non-instrumented version. In figure 6, it is shown an extract of program `ks` and some interpolation functions that were computed by our profiling process. For instance, one of the memory accesses can be modeled as $64i + 24,881,848$ where i denotes a virtual outer loop index.

7. Related work

Most of the tools providing dynamic profiling, as Pin [9], Dyninst [4], Strata [5], DynamoRio [3] or Valgrind [10], are using software dynamic translation (SDT) to handle the insertion of instrumenting instructions. This approach necessarily introduces an inevitable runtime overhead. To our knowledge, VMAD is the first proposal providing low-level instrumentation initiated from the source code with negligible runtime overhead in most of the cases.

One of the most popular tool is Pin [9], which is a software system that performs runtime binary instrumentation of Linux and Windows applications. Pin's aim is to provide an instrumentation platform for building a wide variety of program analysis tools, called pintools. A pintool consists of instrumentation, analysis, and callback routines. A just-in-time (JIT) compiler is used to insert instrumentation into a running application. The JIT compiler recompiles and instruments small chunks of binary instructions immediately prior to executing them. Pin overwrites the entry point of procedures with jumps to dynamically generated instrumentation.

Although Pin is similar to VMAD in the sense that it provides a way to implement some advanced profiling

Program	Runtime overhead	# instrumented functions	# instrumented loops	# instrumented instructions	# instrumented memory accesses	# linear memory accesses
perlbench	0.073%	S_regmatch S_find_byclass	17 36	3,873	404,388	8,420
bzip2	0.24%	mainsort fallbackSort	7 18	502	1,053	608
mcf	20.76%	primal_bea_mpp replace_weaker_arc refresh_potential	2 1 3	138	4,054,863	2,848,589
milc	0.081%	mult_su3_na mult_su3_nn main gaugefix imp_gauge_action setup_layout	2 2 3 1 4 4	195	1,988,256,195	1,988,256,000
hmmer	0.062%	P7Viterbi P7SmallViterbi P7ParsingViterbi ShadowTrace P7Forward ReadSELEX P7Fastmodelmaker printivec seqdecode	1 4 1 3 3 5 2 1 2	742	845	0
sjeng	182%	std_eval setup_attackers	2 5	662	1,155,459,440	1,032,148,267
libquantum	3.88%	quantum_toffoli quantum_sigma_x quantum_cnot	1 1 1	42	203,581	203,078
h264ref	0.49%	SetupFastFullPelSearch FastFullPelBlockMotionSearch FullPelBlockMotionSearch BPredPartitionCost	1 1 4 2	349	32,452,013	30,707,102
lbm	0%	LBM_compareVelocityField LBM_storeVelocityField storeValue	3 3 1	136	358	0
sphinx3	172%	mgau_eval vector_gautbl_eval_logs3	5	194	78,437,958	51,566,707
anagram	-5.37%	BuildMask	3	53	159	134
bc	183%	bc_divide YY_DECL	3 1	142	302,034	243,785
ft	-8.46%	MST	4	36	36	22
ks	29.7%	SwapSubsetandReset FindMaxandwap UpdateDS	1 2 2	102	42,298	29,524

Table 1: Measures made on some of the C programs of the SPEC CPU 2006 (first part) and Pointer-Intensive (second part) benchmark suites.

strategies thanks to the pintools, it focuses exclusively on runtime instrumentation of binaries. Thus it does not require the source code to be available. On the other hand, some advanced dynamic analyses, like the interpolation of memory accesses in loops presented in this paper, would be very difficult to implement with Pin. Of course, the compile-time phase of our framework plays an important role in providing such a wider scope of analysis opportunities. This phase is also important in the runtime overhead minimization, while Pin needs at runtime to parse the binary code, insert instructions and compile some code snippets. VMAD can be seen as a high level version of Pin, where low level instrumentations are initiated from the source code.

The PEBIL toolkit [6] is closer to VMAD since it does not use SDT but static binary instrumentation. However, the instrumentation strategy is different since PEBIL uses function relocation to acquire enough space at instrumentation points to insert correct full-length branch instructions at runtime. We use two different strategies to transfer control from the application to the instrumentation code: at compile time, we insert branch instructions branching initially to their following instructions and that are patched at runtime by VMAD; we also insert callbacks in the instrumented code snippets that are correctly patched at runtime to point to the proper loaded module functions of VMAD.

8. Conclusion

In this paper, we presented VMAD, a dynamic profiling infrastructure where advanced analyses can be implemented with almost negligible runtime overhead, since it does not use software dynamic translation unlike most of the dynamic profiling tools. We extended the LLVM compiler to handle specific pragmas allowing the developer to initiate low-level instrumentation from specific parts of the source code. Some LLVM dedicated passes duplicate the targeted code snippets into instrumented and non-instrumented versions. In the instrumented versions, instrumentation instructions, data, and callbacks are inserted. At runtime, the virtual machine VMAD loads the necessary profiling modules and patches the application code such that all address references to VMAD's functions and data are correct. VMAD's potential has been shown by implementing a profiling strategy interpolating memory accesses in loops. Almost negligible runtime overhead has been obtained by running VMAD with some SPEC2006 and Pointer-Intensive benchmark programs. To our knowledge, VMAD is the first proposal allowing developers to initiate low-level analyses from selected parts of the source code.

For our next developments, we plan to focus on modules that perform code transformations and extend VMAD accordingly. For instance, memory access interpolation can be followed by modules performing data dependence analysis and loop parallelization. Another goal is to generate analysis and transformation modules on-the-fly, since VMAD has also been tailored to support this feature.

References

[1] U. Banerjee. *Loop Transformations for Restructuring Compilers - The Foundations*. Kluwer Academic Publishers, 1993. ISBN 0-7923-9318-X.

[2] C Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[3] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *CGO '03: Proc. of the int. symposium on Code generation and optimization*, pages 265–275, 2003.

[4] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4):317–329, 2000.

[5] N. Kumar, B. R. Childers, and M. L. Soffa. Low overhead program monitoring and profiling. In *PASTE '05: Proc. of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 28–34. ACM, 2005.

[6] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient static binary instrumentation for linux. In *ISPASS-2010 2010 IEEE Int. Symposium on Performance Analysis of Systems and Software*, 2010.

[7] LLVM compiler infrastructure. `http://llvm.org`.

[8] Register allocation in the LLVM compiler. `http://llvm.org/docs/CodeGenerator.html#regalloc`.

[9] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proc. of the ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[10] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2007.

[11] Pointer intensive benchmark suite. `http://pages.cs.wisc.edu/~austin/ptr-dist.html`.

[12] SPEC CPU2006. `http://www.spec.org/cpu2006`.

[13] R. P. Weicker and J. L. Henning. Subroutine profiling results for the CPU2006 benchmarks. *SIGARCH Comput. Archit. News*, 35(1):102–111, 2007.