VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework

Alexandra Jimborean, Luis Mastrangelo,
Vincent Loechner, and Philippe Clauss

INRIA Nancy-Grand Est (CAMUS), LSIIT,
University of Strasbourg, CNRS, France
{firstname.lastname}@inria.fr

Abstract. VMAD (Virtual Machine for Advanced Dynamic analysis) is a platform for advanced profiling and analysis of programs, consisting in a static component and a runtime system.

The runtime system is organized as a set of decoupled modules, dedicated to specific instrumenting or optimizing operations, dynamically loaded when required. The program binary files handled by VMAD are previously processed at compile time to include all necessary data, instrumentation instructions and callbacks to the runtime system. For this purpose, the LLVM compiler has been extended to automatically generate multiple versions of the code, each of them tailored for the targeted instrumentation or optimization strategies. The compiler chooses the most suitable intermediate representation for each version, depending on the information to be acquired and on the optimizations to be applied. The control flow graph is adapted to include the new versions and to transfer the control to and from the runtime system, which is in charge of the execution flow orchestration.

The strength of our system resides in its extensibility, as one can add support for various new profiling or optimization strategies, independently of the existing modules. VMAD's potential is illustrated by presenting several analysis and optimization applications dedicated to loop nests: instrumentation by sampling, dynamic dependence analysis, adaptive version selection.

1 Introduction

Runtime code analysis and optimization becomes the main strategy for facing the ever extending and changing variety of processor architectures and execution environments that an application can meet. Unlike static compilers, that have to take conservative decisions from restricted information extracted from the source code, runtime profilers and optimizers rely on information captured at execution time. While today's processors provide more and more computing resources at the price of increasing usage complexity, particularly with the advent of multicore processors, efficient program optimizations such as adaptive and speculative parallelism, require accurate and advanced runtime analyses. However, such analyses inevitably incur time overhead that has to be minimized.

M. O'Boyle (Ed.): CC 2012, LNCS 7210, pp. 220–239, 2012.

In this paper, we present a framework called VMAD, that includes a virtual machine handling x86_64 binary files, tailored at compile time thanks to dedicated passes developed in the LLVM compiler [22]. One of VMAD's specific feature is that low level profiling is initiated from the source code by the programmer through the insertion of a dedicated *pragma*. The inserted pragmas delimit the regions of the source code of interest and specifies the type of analysis to be performed at runtime. This approach provides the programmer a direct view of the actual execution behavior of his source code. We have extended the LLVM compiler and the Clang front-end to handle such pragmas.

VMAD has the ability of dynamically loading separated modules that implement various analysis strategies. The modules are loaded and unloaded on demand, and several instances of the same module can be loaded simultaneously to handle different code regions. Profiling by sampling, as well as code transformations, are achieved using the same mechanism of multi-versioning and chunking. At compile-time, several versions of the targeted code are automatically generated depending on the semantics of the pragma. For instrumenting by sampling, original and instrumented versions are prepared, while for optimizations, a pattern that can support various code transformations is built. Optimizations are usually preceded by a profiling phase. In this respect, the multiple versions are prepared such that they can be launched successively in chunks of code of various sizes. For example one chunk might represent a subset of iterations of a loop, or a number of function calls. The runtime system then takes the decision concerning the version to be run and adjusts the chunk size correspondingly.

To show VMAD's potential, we present some advanced analysis processes. The first one consists in collecting all memory addresses that are accessed during a selected number of successive iterations of each loop of a loop nest. This instrumentation is rather specific since it occurs on non-contiguous phases of the loop nest execution. The analysis process tries to interpolate addresses successively accessed through each memory reference as a linear function. The second application extends the previous one by performing a dynamic dependence analysis: when all accessed memory addresses can be represented as linear functions, these are transmitted to a dependence analysis module which determines if the loop nest may be parallelized. Finally, the third application is a runtime version selector handling distinct versions of loop nests generated by applying different optimizations. Samples of each version are launched successively and the performance of each version is evaluated by accessing the CPU time stamp counter. The best version is then run for the remaining iterations.

The remainder of the paper is organized as follows. In Sect. 2, we present an overview of our static-dynamic framework. The runtime component is detailed in Sect. 3, and the static preparation of the code is presented in Sect. 4. The loop analysis processes implemented in VMAD are further described in Sect. 5. Finally, we summarize related work in Sect. 6 and conclude in Sect. 7.

2 Framework Overview

VMAD has been built by taking great care of its performance and its runtime overhead. Hence, we avoided the use of software dynamic translation that would delay the execution of the input program. Further, instrumentation instructions are not inserted on-the-fly by replacing some NOP instructions that have been previously inserted at compile time, as done with PEBIL [21]. Rather, we use multi-versioning: several copies are built from the targeted code extracts at compile time. The price to pay with this approach is the larger size of the program binary file. However, great care can also be taken to minimize the size of the copies by inserting branches to the original code whenever possible. Besides performance, another noticeable benefit is the opportunity of implementing advanced analyses which can use versions far different from the original code.

The static-dynamic collaborative framework is depicted in Fig. 1. At compile time, the C/C++ source code, annotated with dedicated pragmas, is translated into the LLVM intermediate representation (IR) with additional specific metadata. An LLVM pass creates copies of the targeted code extracts, sometimes customized with instrumentation instructions. Depending on the type of information intended to be captured during the analysis phase, instrumentation instructions may be inserted either in the LLVM IR, for high-level information, or in the final assembly code, for low-level information. As an example, tracing memory accesses in LLVM IR is not possible, as register allocation is not yet done at this level.

Besides instrumentation instructions, we insert *decision blocks*, providing the means to toggle between versions. We also insert callbacks, in order to invoke VMAD and its related modules when necessary. To be generic, callbacks are inserted as indirect calls and the address of the function to be called is patched at start-up by the virtual machine.

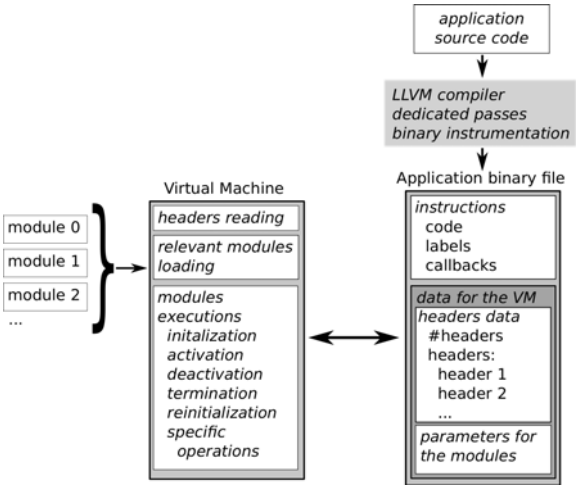


Fig. 1. Framework overview

Moreover, the compiler inserts data in the final binary file to inform the VM regarding the analysis process to be performed and to provide the necessary static information.

At runtime, we use `LD_PRELOAD` to load VMAD's dynamic shared library at startup. `LD_PRELOAD` provides its own version of the C-library entry point *__libc_start_main*, allowing VMAD to read the information statically prepared, to load the required modules and to patch the binary file. Next, control is given to the input program. When necessary, VMAD is reactivated through the callbacks.

3 The Virtual Machine VMAD

The virtual machine makes use of three kinds of information 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; decision blocks to control the selection of the versions; static data corresponding to module parameters, and pointers to the addresses of the inserted code and the callbacks.

Each analysis collaborates with a dynamic module. They share information using a fixed size header which resides inside the binary file. At startup, as soon as the headers and their associated parameters have been read, the VM loads the relevant modules and instantiates them. Parameters fetched at this point are *static data*, and are accessed either by the VM, in order to know which modules are required, or by the modules loaded by VMAD, for instance for obtaining the addresses of the code snippets that have to be patched, or by instrumenting instructions, for allocating memory to backup registers. On the other hand, *dynamic data* is accessed through pointers set up by the corresponding VM modules which allocate the memory they require.

Each module is structured with at least five main entry points: *init* to instantiate an analysis process, *quit* to kill such an instance, *on*, *off* and *reset* to activate, deactivate and reset an analysis process. Additional operations can also be provided by a module. They are invoked thanks to callbacks patched initially 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, as detailed in Sect. 4 and Sect. 5, and have the common form shown in Fig. 2.

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, where the programmer guards interesting regions of code with a specific pragma. The code is then statically shaped to enable the analysis phase, by performing the following steps, detailed below: code tracking, multi-versioning, customizing the versions for instrumentation or optimization, inserting callbacks to the VM, and appending static information required by the VM. An extra challenge is taken when a high optimization level is applied, such as -O3, due to the aggressive code transformations being performed. We first optimize the code

```
sub     $0x80,%rsp      // backup the stack red zone
// backup the scratch registers here...
mov     %rsp,%rbp       // stack adjustement (x86_64 convention)
mov     $0xfffffffffffffff0 ,%rsi
and     %rsi,%rsp
mov     $0x0,%rax       // x86_64 convention
mov     $0x0,%rdi       // address of the module ($0x0 will be patched)
mov     $0x0,%rsi       // address of the operation ($0x0 will be patched)
callq   *%rsi           // function call
mov     %rbp,%rsp       // stack readjustement
// restore the scratch registers here...
add     $0x80,%rsp      // restore the stack red zone
```

Fig. 2. callback in x86_64 assembly code

(O3), then we generate multiple versions and run a few optimization passes to optimize them, without altering the code which requires patching.

Code tracking. Our work relies on the LLVM compiler and on the Clang front-end, which must be extended in order to handle the newly defined pragma. The semantics of the pragma and the delimited region must be preserved from the source code, through the internal phases of the compiler, until the code generation. The original source code is converted into the LLVM IR, annotated with metadata information to mark the code enclosed in the pragma scope. The difficulties of tracking code throughout optimization phases is that metadata information is not entirely preserved, and that code suffers significant transformations. For instance, if one marks the instructions building up a loop and performs loop optimizations, additional code is included (*e.g.* due to loop fusion) or excluded (*e.g.* loop invariants, loop split) from its original body. Therefore, identifying all original instructions is not always possible. Focusing on loops, the conservative solution we propose is to consider that the original loop is transformed into the code region containing

- all loops that include ...
  - at least one basic block containing ...
    - at least one instruction that carries metadata

The consequence is that more code than the one originally marked for multi-versioning might be considered. However, in this manner, we ensure that all instructions of the targeted code region are safely enclosed.

Multi-versioning. Once the region is identified, several clones are generated. We designed an LLVM pass which creates clones of these regions and builds a selection mechanism. The steps to follow for building multiple versions are described in Fig. 3. Using the LLVM copying utilities, we build clones of the instructions found in the region. By default, the clones represent identical copies of the original values. To restore the control flow graph between the copies, a map of the original values and the corresponding clones needs to be built. Based on it, we replace all uses of the original values inside the cloned region, with the corresponding cloned value, thus obtaining a clone of the entire region, as

depicted in Fig. 3B. Finally, each version is extracted into a new function and the selection mechanism is inserted.

Each code version is converted into a suitable intermediate representation, depending on its objectives. Versions created for high level code analysis and optimizations are preserved in the LLVM IR, while versions targeting low-level information are transformed into x86_64 assembly code. For instance, tracing memory behaviour is a difficult task in LLVM IR, because register allocation is not available at this compilation step, and because LLVM IR is in SSA form, containing Φ-nodes. The number of "load" and "store" instructions from LLVM IR is greatly reduced by the optimizers during the code generation, hence they do not represent actual memory accesses. A lower level representation, such as x86_64 assembly, is required for this type of instrumentation. On the other hand, for performing dependence analysis, it suffices to track the "load" and "store" LLVM instructions since it is not relevant for this purpose whether a register or a memory location from the internal memory is accessed.

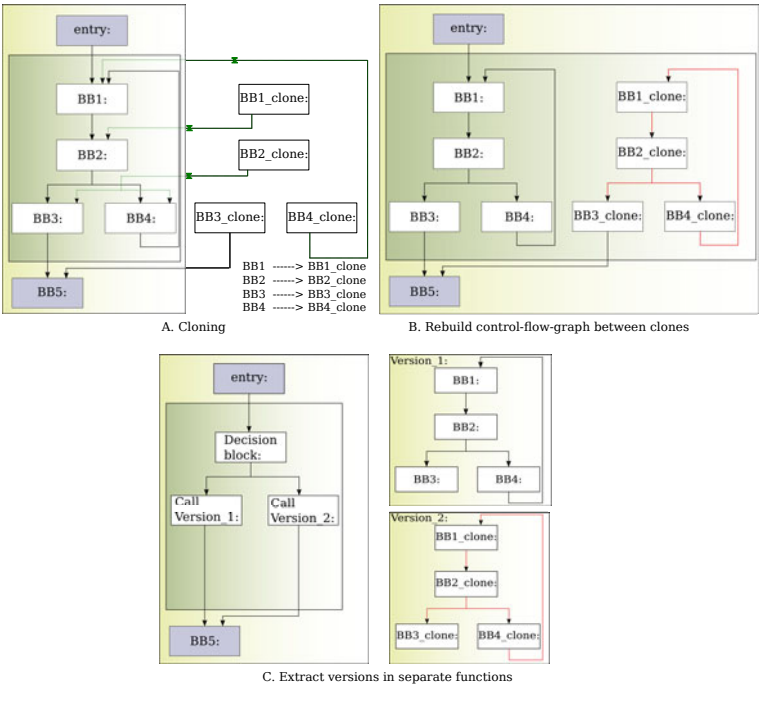


Fig. 3. Multi-versioning

Converting regions of code into different intermediate representations justifies the necessity of extracting the versions in separate functions, further compiled and processed independently. Also, in this manner, we have a clean separation between the regions marked for multi-versioning and the embedding code, from the source level to the LLVM IR and till the x86_64 assembly form. The clones

represented in the LLVM IR are inlined in the original code once they have been customized, to reduce the runtime overhead, but this is not possible for the versions in x86_64 assembly representation, due to register allocation. Nevertheless, the overhead incurred by extracting the versions in separate functions is negligible in most situations.

Chunking. The applications we currently propose for evaluating our framework regard analysis and instrumentation of loop nests, as well as dynamic selection between different optimized versions of loop nests. For these purposes, we build chunks of successive iterations by inserting a virtual iterator *vi* and new conditions $lowerBound \leq vi < upperBound$ in each loop, in order to manage the number and the position of the executed iterations, relatively to the preceding executed chunk. Under these circumstances, sampling is achieved by executing a chunk of the instrumented version, and then switching to the original version. More generally, our strategy provides the means to switch between different versions of a loop nest while completing its execution. After the execution of each chunk, based on the information registered from instrumentation, the VM is invoked to perform additional computations and to guide the decision concerning the next chunked version to be executed. By adjusting the values of the *lowerBound* and *upperBound* dynamically, one can control the sampling rate, restart instrumentation when necessary, and, more generally, can vary the chunk size and the occurring frequency of any version.

Invoking the VM. The selection mechanism consists in preceding each set of versions by a *decision block*, which invokes the VM to decide upon the version to be executed. Additional callbacks to the VM (Fig. 2) are performed at runtime to transmit the data collected via instrumentation, as presented in section 3. The callbacks are placed statically at the beginning and end of each version, and for loops, in addition, they mark the beginning and the end of each instrumented iteration.

Following a generic approach, all callbacks have a standard form, using indirect calls. This approach enables the compiler to generate multiple versions in a generic form, and relies on the VM to patch the address of the corresponding function, at runtime.

Since we patch the code dynamically, our strategy is to inline these code snippets in x86_64 assembly code, ensuring that the size of the code to be patched is fixed. The inline code contains new labels and jumps and the callbacks to the VM. First, this ensures that the VM can track the position of the callbacks in the binary file by using the addresses of the labels. And second, it prevents the modification of the code snippets in the last phases of the code generation, as all jumps and callbacks are inserted in fixed size hexadecimal representation. What we obtain is a partial control flow graph managed as x86_64 assembly code, inlined in the LLVM IR. On the other hand, the inline code is not accessible to the LLVM compiler in this compilation phase. As a consequence, to preserve the validity of the LLVM IR code, we have to maintain both the original CFG, represented as LLVM branches, as well as the jumps inserted in the inline assembly code. The CFG expressed in inline code is the one actually executed, however,

the LLVM branches are preserved to avoid compile time errors, or false cases of dead-code elimination.

Finally, handling inline assembly code throughout the optimizations requires considerable efforts to protect the inlined code against duplications, relocations or dead code eliminations, and to minimally perturb the optimizing passes.

The steps presented above, for statically processing the code, are independent of the type of profiling or optimization to be performed. Next, the clones are modified following the individual specifications and a list of parameters is appended to transmit relevant static information to the VM.

Customizing the Versions. Each function containing a code version is customized according to the type of instrumentation or optimization, either by inserting snippets of instrumenting code, or by performing various code transformations. This is the step when the most suitable intermediate representation is selected. For the examples we address in this paper, various representations are preferred. For tracking the memory locations and building interpolating linear functions, we insert the instrumenting instructions in the final assembly code, after the register allocation. For performing dependence analysis, we instrument the memory accessing instructions at the LLVM IR level and verify pair-wise dependences of load and store instructions. Finally, to perform runtime version selection, for the purpose of this example, the versions are embedded in the source code. The application benefits on the possibility of executing samples of each version and evaluate them using processor counters. The code snippets for evaluating the performance of each version are automatically inserted in the LLVM IR.

Inserting Static Information. In addition to preparing the code, a set of headers and parameters is annexed to the generated binary code (Fig. 4). The list of headers is specific to the type of instrumentation, as they determine the modules to be loaded in the VM (*vmad_0_entry*, *vmad_loop_entry*). Headers are linked to the corresponding parameters (*vmad_0_param*), containing higher level information statically available, but which would be time-expensive to identify in the binary representation (for example, the loop depth). Furthermore, the compiler transmits as parameters instrumentation specific information, for instance the addresses of the code snippets inserted in the original code (*vmad_0_loop_reinstru*, *vmad_0_instru_call*).

5 Illustrating Applications

5.1 Analyzing Memory Accesses in Loop Nests

The first application we propose for evaluating VMAD consists in instrumenting and profiling the memory accesses of critical pieces of code, with a minimal time overhead. In our examples, the target is non-statically analyzable code. We focus on loop nests, as they represent a significant part of the total execution of compute-intensive applications. The goal of the instrumentation framework

```
# 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           # loop depth
        .quad   vmad_0_end_original
        .quad   vmad_0_loop_reinstru
        .quad   vmad_0_instru_call
```

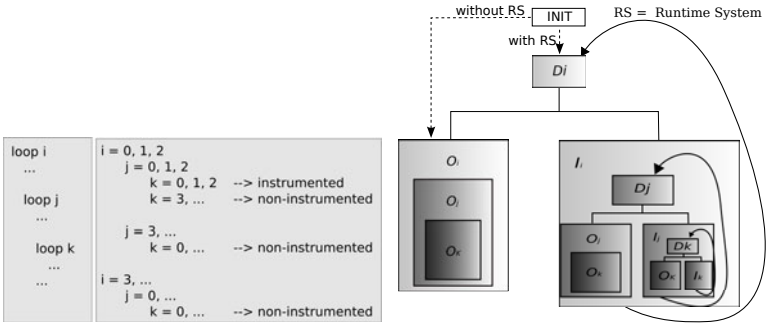
Fig. 4. Headers and parameters

is to collect the memory addresses accessed during samples of iterations and, if possible, to compute linear functions interpolating their values. Additionally, the loop trip counts of each loop, except the outermost ones, are collected for a few runs to be linearly interpolated. Such a profiling is particularly useful for nested loops, either *while*-, *for*- or *goto*-loops, accessing memory through indirect references or pointers. If the memory accesses and the loop bounds can be expressed by means of linear functions, the enclosing loop nests can be optimized and parallelized using *for*-loop dedicated approaches, such as the polyhedral model [4,5]. To handle all loop types in the same manner, we introduce "virtual" iterators, which are maintained to mirror the number of executed iterations.

Loop Nest Instrumentation. Efficient loop nest instrumentation by sampling 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 on the parent loops. For a thorough understanding, consider the loop nest in Fig. 5(a). In this example, 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, each memory instruction should be profiled during the execution of at least three iterations, in order to get sufficient address values. However, since some memory instructions can be guarded by conditional branches, it is required to profile such instructions for more iterations, to increase the chances of collecting enough, *i.e.*, at least three, address values. This contributes to the accuracy of the computed interpolating functions. In our experiments, we fixed the number of instrumented iterations to 10, which was a good trade-off between overhead and accuracy. The sampling rate can be set by a parameter. The first two collected values are dedicated to computing the affine function coefficients, while the remaining values are used to verify the interpolation correctness.

Statically, our LLVM pass creates copies of the loop nests, extracts them in new functions and converts them to x86_64 assembly code. A second pass analyzes the functions and precedes each instruction accessing memory with instrumentation code that computes the actual memory location being accessed,



(a) Loop nest instrumentation. (b) Code structure.

Fig. 5. Instrumenting loop nests

and makes a call to the VM to transmit the collected data. Fig. 5(b) illustrates the structure of the code from Fig. 5(a) and the links between different versions. Blocks O_i, O_j and O_k represent the original versions, 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, containing a callback to the VM. The VM is also invoked when entering or exiting a version of the loop, to retrieve dynamic information. At compile time, we mark the beginning and end of the original and instrumented versions with labels, and append them to the list of parameters given to the VM.

Lastly, a list of headers and parameters is prepared, notifying the VM which are the modules required for this instrumentation: module *vmad_handle_loop*, *vmad_gather_memory_addresses* and *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 uses the information from the previous one to complete its task, however, they are decoupled, hence each module may be employed in performing new types of analysis. The list of parameters contains specific information, such as addresses of the code to be patched at startup, or the structure of the loop nest. At startup, the VM parses the list of headers, loads the solicited modules and patches the code to enable instrumentation.

Analyzing Memory Accesses. Since the number of memory locations accessed inside loops 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 to store the coefficients of the linear functions.

As the instrumented iterations of a loop 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. Subsequent instrumented iterations are used to verify the linearity of these functions.

Communication with the VM is achieved by means of a dirty flag, which indicates that a new memory location is available in the buffer.

Experiments. For our experiments, we targeted the C codes from the SPEC CPU 2006 benchmark suite [29] and four codes from the Pointer-Intensive benchmarks [27]. We inserted a dedicated pragma in the source codes, marking the loop nests (`#pragma instrument_mem_add { // loop nest}`) in the most time consuming functions [31]. 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 produce an amount of data too large to be stored on the disk, but suitable for online consuming. We have carried out the experiments using the O0 and the O3 optimization levels. The execution platform is a 3.4 Ghz AMD Phenom II X4 965 micro-processor with 4GB of RAM running Linux 2.6.32. We ran each program in its original form and in its instrumented form to compute the runtime overhead introduced by using VMAD. For each instrumented loop nest, the dynamic profiling is activated each time its enclosing function is invoked, for the experiments using O0 optimization level. In the experiments with a higher optimization level (O3) we instrument the first eight calls of each function.

Our measurements are shown in Tab. 1. The columns show for each program: the program name (first part of the table: SPEC CPU 2006, second part: Pointer-Intensive); VMAD's runtime overhead, both with O0 and with O3; the code size increase; the number of instructions performing linear memory accesses; the number of instrumented memory instructions; the percentage of memory accesses that were identified to be linear.

For most programs, VMAD induces a very low runtime overhead, which is even negligible for `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 a few iterations, but they are enclosed by functions that are called many times (with O0). Thus, all iterations are run while being fully instrumented. However, the profiling strategy is improved in order to manage such cases by deactivating the instrumentation after a few calls (with O3). Program `milc` shows an opposite behavior since a few memory instructions are executed many times. In such a case the runtime overhead is very low. For the Pointer-Intensive benchmarks, the execution times are too small – of the order of milliseconds – to get relevant overhead measurements: either a large runtime overhead is obtained since VMAD inevitably induces a fixed minimum overhead (`bc`), or even a speedup is obtained (`ft`), which may be explained by cache locality, new alignments or new optimization opportunities. The overhead is higher when instrumenting optimized code (-O3), since modifying optimized code impacts its performance, still, the execution time is better than with O0.

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

Program	Runtime overhead (-O0)	Runtime overhead (-O3)	code size increase	# linear m.a.	# instrumented m.a.	Percentage of linear m.a.
bzip2	0.24%	12.31%	218%	608	1,053	57.74%
mcf	20.76%	17.23%	213%	2,848,598	4,054,863	70.25%
milc	0.081%	3.61%	44%	1,988,256,000	1,988,256,195	99.99%
hmmer	0.062%	0.76%	63%	845	0	0%
sjeng	182%	11.13%	80%	1,032,148,267	1,155,459,440	89.32%
libquantum	3.88%	2.76%	21%	203,078	203,581	99.75%
h264ref	0.49%	4.59%	0.44%	30,707,102	32,452,013	94.62%
lbm	0%	0.93%	170%	358	0	0%
sphinx3	172%	27.62%	20%	51,566,707	78,473,958	65.71%
anagram	-5.37%	34.88%	73%	134	159	84.27%
bc	183%	36.79%	11%	243,785	302,034	80.71%
ft	-8.46%	176%	86%	22	36	61.11%
ks	29.7%	2.98%	268%	29,524	42,298	69.79%

We also noticed that this particular instrumentation process increases the size of a program's binary file by 400 bytes per instrumented memory instruction, on average. However, the code size variation strongly depends on the depth of the loop nests and on the percentage of code selected for instrumentation.

5.2 Dynamic Dependence Analysis

The second application is an extension of the previous one. It adds a module to determine, for a loop nest, which are the loop levels that might be parallelized, according to the memory behavior observed during profiling. Such information can be a useful indication for a developer in order to identify and further analyze such loops, to decide whether they can be effectively parallelized. Our framework identifies the candidate loops by speculatively analyzing dependences between iterations, based on the linear functions interpolating the memory addresses accessed during profiling. The module considers each couple of memory instructions and their associated linear functions, where at least one of them is a write.

We use a simple value range analysis method to determine if the two referenced address spaces can overlap, using the linear functions to compute the minimal and the maximal values of the memory addresses accessed by each instruction. Each write instruction is also considered solely since it can carry an output self-dependence. A loop level not carrying any dependence is then identified as a candidate for parallelization. We used the OmpSCR benchmark suite [26] for our experiments, a set of scientific kernels that are already manually parallelized by the programmer using OpenMP pragmas. Even if these have been deactivated for our runs, they indicate loops being effectively parallel. Loops inside these

Table 2. Dynamic dependence analysis and parallel loop detection in the OmpSCR benchmark suite

Benchmark	#OMP pragmas	#Linear loop nests	#Detected as parallel	Parallel loop levels
FFT	2	2	0	
FFT6	3	10	4	1 / 3 / 1,2 / 1,2
Jacobi	2	4	1	1,2
LUreduction	1	2	2	1,2 / 2,3
Mandelbrot	1	2	1	1
Md	2	2	1	1,2
Pi	1	1	0	
QuickSort	1	2	1	1

kernels contain memory references through pointers, through parameterized array accesses and references to dynamically allocated arrays. Such memory references cannot be handled statically by a compiler. Results are shown in table 2. For two benchmarks, FFT6 and LUreduction, more loop nests than the ones with OpenMP pragmas where detected as parallel. When less parallel loop nests are detected, it is due to dependences induced by reductions.

5.3 Dynamic Version Selection

A loop nest can be optimized using different kinds of transformations such as loop fusion/fission, interchange, skewing, tiling, unrolling, etc. A subset of those transformations can be applied, in different order, or with different parameters (unrolling factor, tile size, ...) to generate distinct versions. Hence many versions can be obtained in this way, and each of them may be the best performing one in some execution contexts, while being slower in some others. Such a phenomenon can occur, for example, when the amount of accessed data generates a lot of cache misses if the computation size exceeds a given threshold. Another case is when the locality of the data accesses depends on some input parameters, or when the control flow traverses costly branches in some circumstances depending on intermediate computations. More exactly, it is a combination of such phenomena that impacts the global performance. Hence, it is in general impossible to predict in advance which version would yield the lowest execution time.

The implemented runtime mechanism consists in first measuring the time per iteration when executing a small chunk of each version, and then running the fastest one for the remaining iterations. Different versions are provided in the source code, delimited by dedicated pragmas. Each version includes an additional condition in the outermost loop, constraining the iterator between a lower and an upper bound, which is required for the chunking mechanism.

At compile-time, the multiple versions are identified and a callback to the dedicated runtime selector module is added, as well as the mechanism to switch between the versions.

The runtime module performs the following operations: for each version, one by one, it sets the chunk bounds such that each new chunk will continue the execution of the previous one, it gets the processor's time stamp counter using the RDTSC instruction, launches the version, gets the new CPU time information, computes the execution time per iteration and stores a reference to the fastest version so far. Finally, when all versions have been evaluated, the fastest version is launched to complete the execution. This naive approach already selects the best version in most cases, but the algorithm can be further refined. Similarly to the sampling rate in the first example, the size of the instrumented chunk can be set by a parameter.

The benchmark programs contain 12 loop nests. The code `2mm` consists of two matrix multiply ($D = A \times B \times C$), `adi` is the ADI kernel provided as an example with the automatic optimizer Pluto [7], `covariance` is a covariance matrix computation, `gemm` is taken from BLAS [6], `jacobi-1d` and `jacobi-2d` are the 1D and 2D versions of the Jacobi kernel, `lu` is a LU decomposition kernel, `matmul` is a simple matrix multiply, `matmul-init` is a matrix multiply combined with the initialization of the result matrix, `mgrid` is a kernel extracted from the `mgrid` code in SPECOMP [3] and `seidel` is a Gauss-Seidel kernel also provided with Pluto.

Such loops are good candidates for loop optimizations such as skewing, loop interchange or tiling. We generated 6 or 7 different versions for each benchmark, either using Pluto or manually. Some versions are tiled, some others are tiled two times in two levels, some others are just skewed or their loops have been interchanged, and finally some are the result of a combination of these transformations. All versions, as well as VMAD's code selector, have been run on a Intel Xeon W3520 at 2.67Ghz under Linux 2.6.38. Results are shown in table 3. For each benchmark, it shows the execution time of the best and of the worst version, the average execution time of all versions, the time when executing with VMAD, and finally a comparison between VMAD and the average execution time.

In most cases, VMAD selects the best version and its execution time is close to the best execution times, and very far from the worst ones. Although it does not select the best version in all cases, it still selects one of the best ones. The overhead is higher when some versions are very slow compared to others.

5.4 Other Possible Applications

In addition to the examples presented above, the VMAD platform can find its applications in debugging or instrumentation, distributed among multiple users. Thanks to the sampling approach and the multiple versions, the selection mechanism can be adjusted such that the version chosen for execution differs from one user to another. Moreover, each version contains only a subpart of the instrumenting or debugging instructions, which ensures a very low overhead, but together, the instrumentation inserted in all versions cover the entire targeted code. Distributed debugging or instrumentation becomes attractive when there is a high number of testers, as each version is executed at least by one user. Moreover, since the overhead is negligible, users are not hindered from executing the versions multiple times. On the other hand, when overhead is not a concern,

Table 3. Dynamic code selection with VMAD

Benchmark	#Versions	Best exec. time	Worst exec. time	Average exec. time	VMAD exec. time	Gap to the average version
2mm	6	2.68	19	8.29	4.80	-42.09%
adi	7	32.99	34.17	33.24	33.10	-0.42%
covariance	6	9.71	145.55	55.81	17.54	-68.5%
gemm	6	7.21	57.10	15.79	9.94	-37.04%
jacobi-1d	6	8.34	11.05	9.70	9.72	0.2%
jacobi-2d	6	2.74	5.24	4.12	4.22	2.42%
lu	6	3.94	51.26	12.11	6.31	-47.89%
matmul	7	4.96	31.49	16.90	6.96	-58.81%
matmul-init	6	3.29	27.04	7.38	4.72	-36.04%
mgrid	6	11.58	16.50	13.45	13.03	-3.12%
seidel	6	76.59	87.71	85.07	86.66	1.86%

the framework can be employed for fully tracing the behavior of the code. This can be achieved by setting the chunk size to a maximal value and selecting the instrumented version.

6 Related Work

VMAD's goal is to be a generic platform running advanced low-level analyses of programs that are initiated from the source code. To our knowledge, there are no previous works directly comparable. However, VMAD can still be related to frameworks that are similar in some important aspects: code instrumentation, code tracking, code cloning and multi-versioning. We also reference a few proposals related to our illustrating applications.

Code Instrumentation. Most of the noticeable code instrumentation tools apply on binary codes. One of the most popular is Pin [23], a software system that performs runtime binary instrumentation. It enables the user to build a wide variety of program analysis tools, known as pintools. A pintool consists of instrumentation, analysis, and callback routines. The insertion of instrumenting instructions is based on software dynamic translation (SDT): a just-in-time compiler recompiles small chunks of binary instructions immediately prior to executing them. Dynamic instrumentation, such as the interpolation of memory accesses in loops presented in this paper, would be impossible to be implemented efficiently with Pin. Of course, the compile-time phase of our framework plays an important role in providing a wider scope of analysis opportunities and in the runtime overhead minimization.

The PEBIL toolkit [21] is more similar to VMAD since it does not use SDT, but static binary instrumentation. PEBIL performs function relocation to acquire enough space at instrumentation points to insert 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 the next instruction and that are patched at runtime; we also insert callbacks in the instrumented code snippets that are patched at start-up with the address of the corresponding functions of VMAD.

The above mentioned tools are designed for instrumenting and profiling the code, nevertheless the goal of VMAD goes beyond code analysis. We aim code instrumentation followed by optimization *on the fly*. This emphasizes the need of a mechanism for creating multiple versions of code and switching between them at runtime (multi-versioning). On the contrary, PIN and PEBIL are tailored to instrument the code for the whole execution time. Their advantage is that they provide easy-to-use APIs allowing the programmer to develop new instrumentation tools, with the cost of an increased overhead at runtime. VMAD requires a new LLVM pass and a new module to support additional instrumentation types. In favour of VMAD comes the fact that it is more flexible in handling multiple instrumented or optimized versions simultaneously. It also allows sampling, by enabling/disabling instrumentation at any time. Moreover, the target code delegates instrumentation related tasks, such as processing the acquired information, to the virtual machine.

Code Tracking. Tracking code has always been a necessary technique, evolving from the simple strategies employed in the early debuggers, to complex approaches meant to correlate the original source code with dynamically optimized code. In our framework, tracking the code through the optimization phases plays a key role, both for identifying the region marked for instrumentation in the source code, and the code that must be patched.

Tracking the suite of code transformations performed in the optimization phase has early been identified as an impractical solution, since compilers reorder, replicate, delete, merge, transform the code, eliminate variables or synthesize new ones. A viable alternative is presented by Brooks *et al.* [8] as a method for acquiring extended debugging information, communicated from one optimization phase to another.

More recent and daring work tackling debugging of dynamically optimized code has been reported [17,19]. The challenge consists in discerning between the optimized code and the optimizers dynamically, and to map it back with the source code, which is no longer available at runtime.

In the gcc compiler [12], generating debug information is possible via the option -g. Also, one can control the amount of information transmitted to the debugger by specifying the level, from -g0 to -g3. This option has been implemented in LLVM [22] and in the Clang front-end [10] and the result consists in populating the code represented in LLVM IR with a significant amount of metadata information, which is then transformed into debug information.

We have adopted a similar approach in tracking code from the source level to the intermediate representation, by marking interesting code regions with metadata information.

The next step in performing multi-versioning is cloning, associated with the construction of a selection mechanism.

Cloning, Multi-versioning, Instrumentation by Sampling. Multi-versioning is a widely adopted strategy to reduce the cost of code instrumentation by sampling. A selection mechanism periodically switches execution between a number of versions embedding instrumentation code and the original version. Chilimbi and Hirzel [14,9] add finer control on the sampling rate and eliminate redundant checks to decrement the overhead. They operate directly on the x86 assembly code using Vulcan [11] for capturing sequences of data references (dynamic executions of loads or stores).

An interesting use of sampling is presented by Chilimbi and Hauswirth [13] for checking program correctness. They develop an adaptive profiling where the sampling rate is the inverse of the frequency of execution of each code region. They adapt the framework introduced by Arnold and Ryder [2] to detect memory leaks. Marino *et al.* [24] extend this solution to multi-threaded programs to find data races.

Our goal is to create a static-dynamic framework that supports multi-versioning and sampling, by means of a generic runtime system that patches the code to enable various types of profiling, instrumentations and code optimizations. We plan to extend our work to accommodate all frameworks described above.

Similarly, ICI [16] has been developed with the aim of providing access to the internal functionalities of compilers. Extensions to ICI [15] provide generic function cloning, program instrumentation, pass reordering and control of individual optimizations. Patching is used to insert an event call before and after the execution of each version, either only for transferring information for further processing, or to change the selection decision of the compiler. In these regards, we have a very similar approach, as we insert callbacks to a runtime system to guard the execution of each code version. However, ICI makes multi-versioning available at function call level only, while we target more precise control for example to enable/disable instrumentation at loop level.

Runtime Code Selection. Several studies proposed a runtime selection between various algorithms, or code extracts, or versions of a function. PetaBricks [1] provides a language and a compiler where having multiple implementations of multiple algorithms to solve a problem is the natural way of programming. Mars and Hundt's static/dynamic SBO framework [25] consists in generating at compile-time several versions of a function that are related to different dynamic scenarios. The STAPL adaptive selection framework [30] runs a profiling execution at install time to extract architectural dependent information. In [28], Pradelle et al. propose a framework to select between versions of loop nests resulting from various polyhedral transformations.

Dynamic Dependence Analysis. The analyzer *pp* [20] is one of the earliest work that proposed hierarchical dependence testing to estimate the parallelism in loop nests. Some recent works are Alchemist [32] and SD3 [18] where runtime and memory overhead is reduced through the use of parallelization and compression.

7 Conclusion

In this paper, we presented VMAD, an infrastructure for dynamic profiling, where advanced analyses can be implemented with almost negligible runtime overhead, since it does not use software dynamic translation like most of the dynamic profiling tools. We extended the LLVM compiler to handle specific pragmas allowing the developer to initiate low-level analyses from selected parts of the source code. Dedicated LLVM passes duplicate the targeted code regions into various versions: instrumentation instructions, version selection mechanism, and callbacks are inserted. At runtime, and when activated, the virtual machine of VMAD loads the necessary analysis modules and patches the callback addresses in the application code. To our knowledge, VMAD is the first proposal allowing developers to initiate low-level analyses from the source code.

Regarding the API, there are two scenarios to be emphasized. If one chooses already defined code analyses, the instrumentation process is totally invisible. The only task is to mark the regions of code of interest with a pragma. We underline that the programmer is not required to annotate the source code with callbacks to the VM, nor to write the decision blocks. These code transformations are handled automatically by our framework. On the other hand, for the compilation expert to develop new types of analysis, it is necessary to write an LLVM pass and to add a module in the virtual machine, containing analysis specific operations. Both the pass and the module are programmed in C/C++ and may include inline assembly code.

VMAD's potential has been shown by implementing the following analyses. First, we instrumented memory accesses in a targeted loop nest, by using sampling. The dedicated LLVM passes duplicate each loop into instrumented and non-instrumented versions, and the control switches from instrumented to non-instrumented code until having collected enough data. Then, we implemented an analysis strategy interpolating those memory accesses as linear functions. Using the results of the interpolation, the next application was to perform dependence analysis and offer hints to the programmer regarding the loops which are good candidates for parallel execution. The last application that we implemented is a runtime adaptive version selector, that takes as input several differently optimized code versions, and selects the best performing one. In this respect, a sample of each version is executed and evaluated based on the processor counters, and the best one is selected to execute until the end of the computations.

Our experiments for interpolating memory accesses as linear functions have been conducted on the SPEC CPU 2006 and on the Pointer Intensive benchmark suites. They reveal almost negligible overhead in most cases, of less than 4%, with -O0 optimization level, and varying between 0,5% and 27% with -O3 optimization level. The two other experiments, on different benchmark suites, also show good results.

We plan to extend the framework to support new types of code instrumentation and optimization. For instance, using the results of the data dependence analysis, we target speculative parallelism by generating code on-the-fly.

References

1. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: a language and compiler for algorithmic choice. In: PLDI 2009, pp. 38–49. ACM (2009)
2. Arnold, M., Ryder, B.G.: A framework for reducing the cost of instrumented code. SIGPLAN Notices 36(5), 168–179 (2001)
3. Aslot, V., Domeika, M.J., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 1–10. Springer, Heidelberg (2001)
4. Banerjee, U.: Loop Transformations for Restructuring Compilers - The Foundations. Kluwer Academic Publishers (1993) ISBN 0-7923-9318-X
5. Bastoul, C.: Code generation in the polyhedral model is easier than you think. In: PACT 2004: Proc. of IEEE Int. Conf. on Parallel Architectures and Compilation Techniques (2004)
6. Blackford, L.S., Demmel, J., Dongarra, J., Duff, I., Hammarling, S., Henry, G., Heroux, M., Kaufman, L., Lumsdaine, A., Petitet, A., Pozo, R., Remington, K., Whaley, R.C.: An updated set of basic linear algebra subprograms (blas). ACM Transactions on Mathematical Software 28, 135–151 (2001)
7. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI (2008)
8. Brooks, G., Hansen, G.J., Simmons, S.: A new approach to debugging optimized code. In: ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI (1992)
9. Chilimbi, T.M., Hirzel, M.: Dynamic hot data stream prefetching for general-purpose programs. In: PLDI 2002: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (2002)
10. Official website of clang: a C language family frontend for LLVM, `http://clang.llvm.org`
11. Edwards, A., Vo, H., Srivastava, A.: Vulcan binary transformation in a distributed environment. Tech. rep. (2001)
12. The GNU Compiler Collection, `http://gcc.gnu.org`
13. Hauswirth, M., Chilimbi, T.M.: Low-overhead memory leak detection using adaptive statistical profiling. In: 11th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, ASPLOS-XI. ACM (2004)
14. Hirzel, M., Chilimbi, T.: Bursty tracing: A framework for low-overhead temporal profiling. In: 4th ACM Workshop on Feedback Directed and Dynamic Optimization FDDO4 (2001)
15. Huang, Y., Peng, L., Wu, C., Kashnikov, Y., Rennecke, J., Fursin, G.: Transforming GCC into a research-friendly environment: plugins for optimization tuning and reordering, function cloning and program instrumentation. In: 2nd Int. Workshop on GCC Research Opportunities (GROW 2010), Pisa Italy (2010), Google Summer of Code 2009 (2010)
16. Interactive Compilation Interface, `http://ctuning.org/ici`
17. Jaramillo, C., Gupta, R., Soffa, M.L.: FULLDOC: A Full Reporting Debugger for Optimized Code. In: SAS 2000. LNCS, vol. 1824, pp. 240–260. Springer, Heidelberg (2000)

18. Kim, M., Kim, H., Luk, C.K.: Sd3: A scalable approach to dynamic data-dependence profiling. In: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, pp. 535–546. IEEE Computer Society, Atlanta (2010)
19. Kumar, N., Childers, B., Soffa, M.L.: Transparent debugging of dynamically optimized code. In: Int. Symp. on Code Generation and Optimization, CGO 2009. IEEE Computer Society (2009)
20. Larus, J.R.: Loop-level parallelism in numeric and symbolic programs. IEEE Trans. Parallel Distrib. Syst. 4, 812–826 (1993)
21. Laurenzano, M., Tikir, M., Carrington, L., Snavely, A.: PEBIL: Efficient static binary instrumentation for linux. In: ISPASS-2010: IEEE Int. Symp. on Performance Analysis of Systems and Software (2010)
22. LLVM compiler infrastructure, `http://llvm.org`
23. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: PLDI 2005: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (2005)
24. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: PLDI 2009: Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation (2009)
25. Mars, J., Hundt, R.: Scenario based optimization: A framework for statically enabling online optimizations. In: CGO 2009, pp. 169–179. IEEE Computer Society
26. OmpSCR: OpenMP source code repository, `http://sourceforge.net/projects/ompscr`
27. Pointer-intensive benchmark suite, `http://pages.cs.wisc.edu/~austin/ptr-dist.html`
28. Pradelle, B., Clauss, P., Loechner, V.: Adaptive runtime selection of parallel schedules in the polytope model. In: ACM/SIGSIM High Performance Computing Symposium (HPC 2011). ACM (April 2011)
29. SPEC CPU (2006), `http://www.spec.org/cpu2006`
30. Thomas, N., Tanase, G., Tkachyshyn, O., Perdue, J., Amato, N.M., Rauchwerger, L.: A framework for adaptive algorithm selection in stapl. In: PPoPP 2005, pp. 277–288. ACM (2005)
31. Weicker, R.P., Henning, J.L.: Subroutine profiling results for the CPU2006 benchmarks. SIGARCH Comput. Archit. News 35(1) (2007)
32. Zhang, X., Navabi, A., Jagannathan, S.: Alchemist: A transparent dependence distance profiling infrastructure. In: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2009, pp. 47–58. IEEE Computer Society, Washington, DC (2009)