MIL: A language to build program analysis tools through static binary instrumentation

Andres Charif-Rubial, Denis Barthou, Cédric Valensi, Shende Sameer, Allen Malony, William Jalby

MIL : A language to build program analysis tools through static binary instrumentation

Andres S. Charif-Rubial*, Denis Barthou†, Cédric Valensi*, Sameer Shende‡, Allen Malony‡, William Jalby*
* Exascale Computing Research Laboratory, FR
Email: {achar,cedric.valensi,william.jalby}@exascale-computing.eu
† Laboratoire LaBRI, University of Bordeaux, Bordeaux, FR
Email: denis.barthou@labri.fr
‡ Department of Computer and Information Science
University of Oregon, Eugene, OR, USA
Email: {sameer,malony}@cs.uoregon.edu

***Abstract*—As software complexity increases, the analysis of code behavior during its execution is becoming more important. Instrumentation techniques, through the insertion of code directly into binaries, are essential to program analyses such as performance evaluation and profiling. In the context of high-performance parallel applications, building an instrumentation framework is quite challenging. One of the difficulties is due to the necessity to capture coarse grain behavior, such as the execution time of different functions, as well as finer-grain behavior in order to pinpoint performance issues.**

In this paper, we propose a language, MIL, for the development of program analysis tools based on static binary instrumentation. The key feature of MIL is to ease the integration of static, global program analysis with instrumentation. We will show how this enables both a precise targeting of the code regions to analyze, and a better understanding of the optimized program behavior.

I. Introduction

As software complexity increases with the development of multicore architectures, high performance parallel applications are increasingly difficult to tune for performance, to debug and profile. Due to compiler optimizations, runtime interactions and complex shared memory hierarchies, capturing the runtime behavior of the code is an essential step in code analysis. The purpose of binary instrumentation is to insert new code into an executable in order to collect and analyze information concerning an execution. Tools offering binary instrumentation such as Dyninst [1], MAQAO [2], Pebil [3], Pin [4] or Valgrind [5], are at the heart of code analysis tools used today. For performance analysis, a first coarse grain analysis is usually achieved in order to identify hotspots, and on these hotspots, a finer grain analysis capturing more details, follows. In order to adapt to the level of details required and to avoid the cost of an indiscriminate and expensive fine-grain instrumentation, several instrumentation languages have been proposed [6], [7], [8]. Instrumentation languages help to define where to insert the instrumentation probes, based on the structure of the binary code in terms of functions, loops, and sometimes blocks or instructions. However, compiler optimizations may change deeply the structure of the code, from the source to the binary, and this limits the effectiveness of such approaches. Indeed, if the compiler generates two versions of a loop, one vectorized and the other not, the binary instrumentation techniques proposed only report performance analysis for each loop, independently of the other. Similarly, when a function "foo" has been inlined or cloned by the compiler as it occurs for OpenMP codes, current instrumentation techniques cannot instrument the "foo" inlined versions (no longer functions), nor relate instrumentation results of cloned versions to the "foo" function. For instance, Figure 1 shows the output of the TAU profiler [9] using Dyninst [1] on the `bt.A` NAS OpenMP benchmark for the thread 1 (out of twelve), the three most time-consuming functions have names that do not match those of the source code, `x_solve`, `y_solve` and `z_solve`. The reason is that the compiler has generated new functions, associated to the `parallel for` constructs and the profiler has not found the correspondance with the source code functions. For optimized codes, the main challenge for instrumentation languages is therefore not only to enable an efficient description of the code fragments to instrument but also to report information relevant for users.

%Time	Exclusive msec	Inclusive total msec	#Call	#Subrs	Inclusive usec/call	Name
100.0	7,845	9:30.284	1	1012	570284826	.TAU application
32.6	3:04.814	3:05.867	201	155905	924715	void targ4161f9()
32.4	3:03.927	3:04.840	201	136524	919605	void targ419ff9()
32.4	3:03.162	3:04.547	201	207576	918145	void targ4146f8()
0.7	3,357	4,058	2	100001	2029312	void targ402c52()
0.3	1,763	1,763	202	0	8728	void targ40be37()

Fig. 1. `pprof` tool output for thread 1 on NAS OMP bt.A benchmark using Dyninst with TAU

In this paper we propose a binary rewriting framework and a domain specific instrumentation language, MIL, for the development of code analysis tools. This language is built on top of MAQAO, a static performance analysis tool[10]. The original contributions of our approach are:

- MIL, a versatile language for instrumentation: the language MIL can be used to gather information on large variety of events, from functions to loops, blocks and instructions for control-flow profiling or value-profiling.

The probes inserted in the binary code can be user-defined, enabling for instance hardware counter profiles, and written either in MIL (MILRT runtime), in C (through an external library) or in assembly. Besides the precise location of these probes, their parameters can be defined by scripts, using a rich API of static analysis. In particular, probes can have parameters dependent on some static property of their insertion location.

- A framework for optimized multi-threaded code analysis: compiler optimizations can generate functions for OpenMP codes with complex control-flow. MIL enables the developer of code analysis tools to focus on functions appearing in the source code, independently of any name mangling, inlining or transformation due to OpenMP directives.
- A low-overhead instrumentation: We combine techniques presented in Dyninst[1] with more aggressive techniques for adding instrumentation code. In particular, we introduce in MIL interpreter new techniques for low-overhead instrumentation of OpenMP optimized codes.

The paper is organized as follows: Sections II and III present the instrumentation language and describe how the binary code is restructured using different algorithms. Section IV shows how this tool is integrated into the TAU parallel performance system, along with an example of a simple function profiler, fully written in MIL for multi-threaded codes. Finally, section V presents the evaluation of the overhead due to instrumentation and a comparison with other existing tools.

II. Instrumentation Language

MIL is a scripting language to define binary code instrumentation. The interpreter of MIL scripts is detailed in Section III and is built on top of MAQAO [2]. Running a MIL script with an input executable produces a new instrumented executable. The possible locations where instrumentation can be inserted are called events and the description of what to instrument corresponds to event filters. The definition of the *probes*, i.e. the code to insert for a given event, is also given in MIL. We present thereafter how to express both in MIL.

A. Abstract Code Structures and Filters

To define instrumentation points, the following structural abstractions can be manipulated: the program itself, its functions, loops, basic blocks, instructions and a particular case of instructions, the call sites. These notions are usual structurations of programs and correspond here to structural abstractions found in the binary code. Loops correspond to only natural loops, and functions may have multiple entries. To define precisely where to insert some instrumentation, we define the notion of *event* as being a particular location in these structures, as for instance the entries and exits of a loop. The events associated to a particular structure are summarized in Table I.

The structures defining the events are described in a hierarchical way, reflecting the nesting of functions, loops, basic blocks and instructions.

	`program, blocks`	`functions`	`callsites, instructions`	`loops`
Event names	`entry, exit`	`entries, exits`	`before, after`	`entries, exits, backedge`

TABLE I. Structural abstractions and associated event names.

	`program, functions`	`callsites`	`loops`	`blocks`	`instructions`
Whitelist, blacklist	name	target name	id	id	address
Depth			integer `inner, outer`		
User	user-defined function				

TABLE II. Structural abstractions and associated filter mechanisms.

MIL is an object-oriented and event-directed language. It extends the Lua syntax[11] providing additional classes and constructs to specifically manipulate all the concepts of our DSL (binary instrumentation), namely, structural objects (functions, loops, ...), events, filters and probes. For a complete set of examples showing all the language capabilities and an exhaustive description of the syntax, please refer to the MAQAO tool project page [10]. We also provide language files for the vim editor. An Eclipse plugin is envisioned.

Figure 2 depicts the rationale of a MIL script. We select a binary and target structural objects. For this example, "Object" is used as a generic container for Function, Loop, Block, Instruction or Callsite (Binary being a special one). Then we can specify filters and events related to objects. The final step is to create probes for each selected event. More details on each stage is given in the next subsections.

Instrumentation overhead is one of the dominant concerns when considering how a binary rewriting tool is used to enable performance measurement. There are two ways to reduce the overhead of instrumentation at binary level: reduce the time taken by the probes themselves or reduce the number of structures instrumented by applying filters. In order to be able to restrict the field of objects to be processed, a filtering mechanism is mandatory. The filtering mechanisms associated to a particular structure are summarized in Table II. A set of filters may be defined for every structure and can be either a list, a built-in filter or a user-defined filter. A filter using lists can define whitelists and blacklists for any structure. Depending upon the given structural object, its main attribute is used to apply the filtering. For instance, the whitelist filter defined by the `"^calc"` followed by blacklist filter `"_test$"` selects all functions with name beginning with `calc` that does not end in `_test`. For callsites, the whitelist filter `"^calc"` selects all instructions calling functions beginning with `calc`. It is also possible to select a set of structures based on their structural attributes (built-in). So far, only one built-in attribute, `depth` is defined for loops. It is possible to select loops with this filter at a given depth ({`depth = 3`}) or using the key words `inner`, `outer` for relative depths.

Figure 3 presents an example of multiple (here two) events,

```
--the main object is name "this"
binary = this:addMainBinary() --to select our binary;
--Define target objects
obj = binary:addObject()
obj_nested  = obj:addObject(); --nested object
--Apply filters to narrow down the set objects targeted
obj:addFilter() --Different types of filters available
--Define events on objects
event = obj:newEvent("event_name"):
probe = event:AddProbe()
--Then fill the probe: specify the target and parameters
```

Fig. 2. MIL script rationale.

each using a nested definition. The first event nest counts the number of innermost loops, of function ”foo1”, that only has one basic block. Hence using a nest of function, loop and basic block events. The second event nest counts the total number of iterations passed in outermost loops of function ”foo2”.

```
function isinner_oneblockloop(loop)
 if(loop:get_nblocks() == 1) then return true; end
 return false;
end

this:setRunDir("/PATH_TO_OUTPUT_FOLDER/");
mb:newEvent("at_entry"):newProbeExt("init_data","my.so");
mb:newEvent("at_exit"):newProbeExt("dump_rslt","my.so");
mb = this:addBinaryMain("PATH_TO_BINARY");
--First function event
fct1  = mb:addFunction():addFilterWL("^foo1$");
loop1 = fct1:addLoop();
loop1:addFilterBI({depth = "inner"});
loop1:addFilterUser(isinner_oneblockloop);
block = loop:addBlock();
block:newEvent("entry"):newProbeExt("count_blocks","my.so");
--Second function event
fct2  = mb:addFunction():addFilterWL("^foo2$");
loop2 = fct2:addLoop():addFilterBI({depth = "outer"});
loop2:newEvent("backedges"):newProbeExt("count_iters","my.so");
```

Fig. 3. Combining multiple events

It also illustrates the usage of user-defined filters, here used to detect innermost loops (in the first event nest) that are composed of only one basic block. A special attribute, `user`, exists for all structures. This attribute corresponds to a user-defined boolean function that evaluates to true only if the structure should be considered for instrumentation. User-defined filters provide more flexibility when simple filters fail to identify precisely the code fragment to instrument. These functions are written in Lua and are meant for more advanced filtering which can manipulate the structure through MAQAO API (in Lua).

B. Instrumentation Probes

After having selected target instrumentation location through events, it is possible to describe the probes to insert into the binary. It is possible to define the probes either in Lua, or to provide the name of the probe function with the name of a shared library containing it, or to define a string with inlined assembly code. With the first method, defining probes in Lua, the complete instrumentation configuration, including filters, probes, can be defined in one single file. We believe this can ease the use of MIL and help developers to define new profiling analyses. In this case, a call to this function, through the Lua interpreter is inserted in the binary and the script of the function is appended to the binary. As many calls to an interpreter may generate large overheads, the Lua Just-In-Time compiler [12] is added to the binary instead of the interpreter.

External calls to precompiled libraries containing the probes have been used by other tools, such as Dyninst [1]. Concerning the insertion of assembly text, we propose a gcc-like inline assembly that handles loops and global variables. Note that both external calls and inline assembly can be used at the same time for the same instrumentation point. When inserting a call to an external function, it is possible to disable context saving. This may be useful when the inserted function already saves and restores all the registers it will be using. Even if most of the users will only use the insertion to external calls or Lua functions, it is important to be able to insert assembly code because it enables significant optimization opportunities.

Given an event, any number of probes can be inserted. For each event, the following attributes can be specified:

- Assembly code to insert before/after the current probe,
- `nowrap`, avoid saving the current probe context,
- Library containing the function to be called,
- Name of the function, for functions defined in libraries or in Lua (in the MIL script)
- Parameters of the called function if any.

The available parameters types are:

- Immediate
- String
- Global variable: Global variables are declared at the beginning of the specification file and can have default values (immediate or string). They are usually used to store the return values of inserted functions and then passed to others.
- Memory: the default behavior is to return the target address of the instruction (of a jump, a load or a store for instance). This enables to capture memory references (accesses) or the value pointed by the memory reference (specified through an additional option). It can also help capturing complex control flow in case of indirections (computed destination). This type of parameter is only available for instruction-level events.
- User defined function.

User defined functions allow to pass to the probes any value computed statically from the analysis of the binary. Note that while the probe is executed during the execution of the application, the evaluation of the parameters of these probes is at the instrumentation time. These functions receive the object (pointer) of the structure instrumented to perform a variety of queries and operations. Given a structure object, MIL is able to provide an access to the MAQAO Framework API (in Lua) . It is beyond the scope of this paper to describe how this occurs internally. Detailed documentation is available on the MAQAO website [10]. Such user-defined parameters can depend on the instrumentation site and can be used to pass the information to the probe that the current loop is vectorized or unrolled for instance.

C. Using MIL to reduce instrumentation overhead

The first step to reducing overhead is to limit the number of code fragments to instrument. The filtering mechanism proposed in Section II-A proposes a simple way to instrument only some parts of the code. Other analyses may require more elaborated filters. Mussler *et al.* [8] use a predefined group of structural properties on the code as filters. Predefined filters may apply to limited class of applications, but selecting *a priori* which parts of the code are of interest is in general intractable. We propose user-defined filters, introduced as a more generic and complementary approach to predefined filters.

Optimizing instrumentation time also concerns the way the instrumentation is inserted into the code. Inserting a function call in a binary application has a cost, namely, the call instruction itself and the instructions to save the context before the call and restore it after. Inserting assembly instructions instead of calling a function removes this overhead. It may seem an extreme optimization, but it could be effective in cases where an instrumented routine contain loops that themselves call other functions. Although this kind of optimization requires architecture-specific considerations, it can reduce significantly the cost of inserted instrumentation calls.

III. MIL Interpreter

The instrumentation language interpreter is developed as a new module for MAQAO [10]. MAQAO is a framework for analyzing and optimizing binary codes and it combines binary disassembly, rewriting, and assembly with analysis to identify code semantics and reconstruct control flow. The framework relies on usual compiler algorithms to detect functions, loops and basic blocks. MIL interpreter complements this information from the binary code with additional static analyses for handling optimized OpenMP codes. For the generation of the instrumented code, the interpreter drives a module of MAQAO, named MADRAS, that proposes a very low level API for instrumentation of x86_64 codes. MADRAS only considers instructions (no loops, functions, or blocks). This same module is used for disassembling the binary code.

Figure 4 shows the components of the instrumentation language and its integration in the MAQAO framework. Blacks arrows describe the components involved in the basic workflow of MIL. Gray arrows depict the additional possible interactions with the MAQAO framework.

A. Static Binary Instrumentation

Several approaches have been considered for instrumentation. In contrast to source code instrumentation (such as proposed in OPARI[13] for instance), or instrumentation that operates at an intermediate language level, binary analysis and instrumentation starts with the program code in its final executable form. Source instrumentation, while flexible, has the disadvantage of requiring recompilation of the application. Besides, the modification of the code can alter the effects of compiler optimizations. Working with the binary code avoids recompilation and preserves any optimization performed by

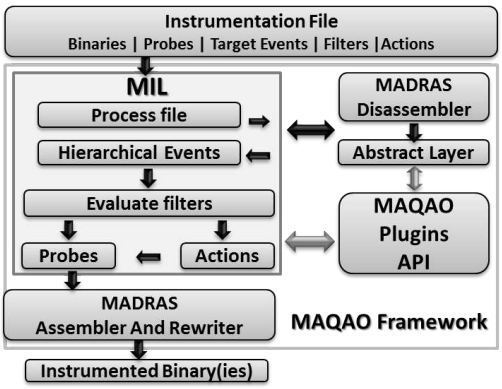

Fig. 4. MIL : MAQAO Instrumentation Language and its integration in the MAQAO framework.

the compiler. However, it does present additional challenges that need to be overcome to deliver a robust instrumentation solution. Below we discuss the issues that arise, the different alternatives, and our approach.

In general, static binary rewriting has two important advantages compared to the dynamic instrumentation. First, because the whole executable code is available when inserting instrumentation, static rewriting is more robust, able to perform more instrumentation requests, and can implement optimization methods more easily. Second, instrumentation occurs only once and before execution. Subsequent runs of the program will include the instrumentation.

For MIL, we use the MAQAO module, MADRAS, to disassemble binary. MADRAS can insert function calls or assembly instructions, delete instructions, or modify them by changing their opcode or operands. Inserted function calls can be wrapped with instructions for saving the context (contents of registers and stack frame) and restoring it after the inserted call, thus ensuring that the execution of the inserted function is transparent for the executable. MADRAS is also able to insert global variables and reference them in inserted or modified code. The MIL interpreter drives MADRAS in order to handle optimized OpenMP codes, as described in the following section.

B. Advanced Static Analysis

Multithreaded and optimized binary codes can present some specificities that introduce challenging binary analysis problems. Four of the major issues we encountered when analyzing codes, as far as code structure is concerned, were handling indirect branches, interleaved functions, inlining and probe insertion in some (difficult) cases. Our solutions to these issues are discussed below. To illustrate our explanations, we will be using *bt* and *dc* (class A) benchmarks from NPB-OMP3.3 [14], and 312.*swim* benchmark (Medium) from SPEC OMP 2001 [15] , all compiled by the Intel Fortran compiler (ifort) with -O3 optimization.

1) Indirect Branch Resolution: There is a major issue with indirect branches if not handled because they can hide exits

(of a function). In order to obtain the complete set of exits of a function, we need to resolve indirect branches within functions. We introduce the concept of conditional probes. It consists of a regular probe combined with a set of conditions. The core idea is to set a condition on the target of the indirect branch once resolved. When considering function boundaries, the condition holds on the set of intervals that describes the limits of the function in terms of addresses. If the target is outside these intervals, then it is an exit and the probe would be executed. This algorithm is implemented internally inside MIL interpreter and requires no input from the users. If desired, users can disable indirect branch resolution in the configuration part of their script.

2) Interleaved Functions: At source level, it is relatively straightforward to identify the structure of a function, and functions have one entry and multiple exits (returns). While exits can be a little tricky to instrument in source, when we consider the general problem at binary level, optimizations achieved by the compiler may produce a more complex code structure. In binary, functions are only labels and it is even possible for two functions to share common blocks (due to compiler optimizations). These *interleaved* functions make the abstraction of the code more complex to handle and are generated for instance by the Intel compiler for OpenMP codes. When it comes to instrument a function, specific measures have to be taken.

To detect interleaved functions, we apply a connected component search on the control flow graph (CFG) of a given function, in our static analysis phase, and make the interleaved functions appear as separate components. If we consider the *bt* benchmark, the multi-threaded part of the code in functions containing OpenMP directives (i.e., the part of code that will be called by the OpenMP runtime) is inlined. Figure 5 reveals a part of the control flow graph of one of the most time-consuming functions of the *bt* benchmark after MIL advanced static analyses. We can observe that it has successfully separated each component of the CFG. By default, MIL default behavior is to consider each component as a regular function. The name of the function is the same as the container function concatenated with a unique suffix and may be different when, for instance, inlining is detected. It is possible to disable this behavior in the configuration section (properties) of a MIL script file.

Let us consider a more complex example with the *swim* benchmark. This application contains four main (most consuming time) functions called from the program entry function. Taking a closer look to the code, we observe that three of these functions are actually inlined in the main routine. The inlined functions are called from the OpenMP runtime and entry/exit points are merged with the ones of the main routine. If only the main routine entry and exits points were instrumented, we would miss accounting for the three inlined routines. In fact, basic time profiling methods show only main and the routine not inlined as the two dominant time-consuming functions. The connected component analysis of MIL can discover the inlined functions and correctly apply function level instrumentation. The most important point of this approach is solving the problem using a static analysis, which is essential to reduce the instrumentation runtime overhead.

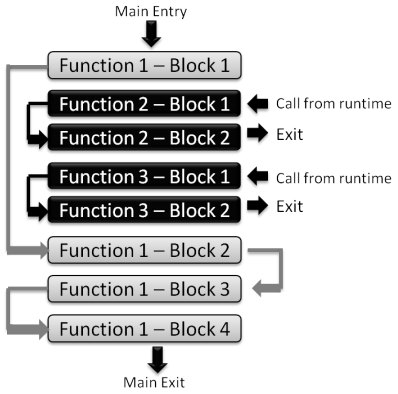

Fig. 5. Part of the CFG of the main function (MAIN__), from the bt benchmark (using Intel fortran compiler with -O3), revealing an example of interleaved functions

3) Inlined Functions: In the previous paragraph, we mentioned the *swim* benchmark example where functions where inlined. In the OpenMP class codes, that is what usually happens because the multi-threaded part of the code is actually called by the OpenMP runtime. As far as Intel compilers are concerned, the starting address of the multi-threaded code to execute is a pointer passed as a parameter of the function of the runtime which is responsible for calling that code. In general, detecting inlining is at least a challenging task and may be impossible at binary level. In MIL, we added a new heuristic that uses debug symbols, when available, to detect inlined functions. The instructions of functions that are inlined have a specific source line, the call site source line. Given the control-flow graph of a function, we look for subgraphs with basic blocks that have a high percentage (80%) of instructions that have that property. If we consider again interleaved functions, this heuristic works almost every time (with Intel compilers) and helps figuring out the name of the inlined function. Actually the name is given by the destination function of the call site. This algorithm is implemented internally and requires no input from the users. If desired, users can disable inlined functions detection in the configuration part of their script.

Even if it can be considered as a weak approach, we propose this debug information based heuristic because it is by far more accurate and faster than our original pattern matching algorithm (which does not require debug symbols and can be used as a fallback). It works with GNU and Intel compilers, considering that most users use one of these two compilers (in HPC) for a variety of languages (Fortran, C or C++).

4) Probe Insertion Issues: We introduce the concept of conditional probes in order to control the conditions under which a probe should be executed or not. After inserting a probe, a set of conditions can be applied on it. Thanks to this approach, it is possible to solve challenging issues.

One recurrent concern when dealing with instrumentation is the ability to insert probes wherever the users asks for. The usual solution involves using function relocation which actually does not work with function pointers since they

continue to point to the original function. Inserting probes anywhere is not always possible at limited cost. For instance, on x86_64 architecture, the *dc* benchmark contains 11 functions with insufficient space for probe insertion happens. The common workaround is the concept of trampolines. It is used to find the required space close to the instrumentation site. This works most of the time. However, when considering one byte instructions (i.e. return instruction), trampolines are useless. Indeed, the smaller branch instruction that can be used for trampolines need at least two bytes. Therefore, when trampolines cannot be found or instructions are too short, the only existing technique is to resort to a trap instruction which has a size of one byte. Basically, a trap instruction invokes a system trap (signal) handler that can then execute the probe. The induced overhead is however consequent, larger by a factor of 15 compared to a regular probe.

We propose a new algorithm, which is part of MIL internals, that solves most of these cases, including the issue observed in the *dc* benchmark. Algorithm 1 details our approach. We perform a control flow analysis to figure out the predecessors of the current block where instrumentation should have taken place and instrument them. We go through these predecessors and verify that there is enough space to insert probes. If not, we have no choice but to insert a trap instruction. If there is enough space in all the predecessor blocks, then we have to determine for each of them if their target is the current insertion block or not (where the probe must be inserted). One case is quite complex, when considering a conditional branch. Since we only can insert the probe before it, we must add a condition so that the probe is only executed if we are sure that the flow is going to the current instrumented block. When considering the previous example, that means when branching to the exit block of the function.

In a nutshell, our method minimizes, and even removes, the number of trap instructions needed to correctly instrument a function.

5) Debug symbols: MIL works even without debug information. It is only used in order to relate assembly (dissassembled binary) instructions to source line codes. As a consequence, only heuristics and analyses based on source lines will fail. Thus, all the major features and improvements we described, compared to the other instrumentation frameworks, still work. Having said that, we reasonably think that debug information will be most often present.

Algorithm 1: InsertProbe

```
input  : probe to insert
output : SUCCESS or FAILURE

inst ← GetInsertInst(prob);
block ← GetBlockFromInst(inst);
if IsSmallBlock(block) then
    predBlocks ← GetPredBlock(block);
    foreach pb in predBlocks do
        if IsSmallBlock(pb) then
            Insert a trap instruction (INT3)
        LIB ← GetLastInstOfBlock(pb);
        if InstIsBranch(LIB) then
            if InstIsUncondBranch(LIB) then
                ProbInsert(inst,prob,BEFORE)
            else // InstIsCondBranch(LIB)
                brTargB ← GetBranchTarg(LIB);
                if brTargB == block then
                    BC ←
                    GetOppositeBranchCond(LIB);
                    CV ← ExtractCompareVal(BC);
                    ProbCondInsert(inst,prob,CV,BEFORE);
                else
                    ProbInsert(inst,prob,AFTER);
        else
            ProbInsert(inst,prob,AFTER);
```

IV. Building Performance Tools

A systematic performance analysis approach must adopt a measurement methodology where critical performance bottlenecks can be identified at a coarse level and then instrumentation at a finer level can pinpoint performance issues. The challenge is to create a performance analysis system that supports both flexible instrumentation that preserves code properties relevant to the user and lightweight performance measurement that keeps overheads to a minimum.

The TAU Performance System [9] from the University of Oregon is a performance evaluation toolkit that supports several instrumentation, measurement, and analysis alternatives. TAU presents a good target to prototype a MIL-based instrumentation tool because it has challenging requirements. The goal in integrating TAU with MIL was to simplify the usage of TAU and create an efficient binary rewriter for multi-threaded applications. A new script, named *tau_rewrite*, has been added to the TAU distribution in order to add instrumentation to binary files and dynamic shared objects using MIL (included in the Program Database Toolkit). The tool enables users to inject a specified TAU measurement library while rewriting the executable.

Figure 6 shows the instrumentation file in MIL for the TAU performance tool. Probes calling the TAU runtime are placed at the entries and exits of the functions of the program. For each processed function, a specific registration call is added to the stub of functions executed when the binary is loaded. It is a more flexible approach to generate a stub of calls while discovering functions. At the end of the program, results are dumped by the TAU runtime. This is all what is necessary to enable binary instrumentation for the TAU performance measurement demonstrated below. The code achieving the same functionality, for Dyninst, requires around 200 lines of code, according to the authors.

Figure 7 depicts a simple standalone profiler completely written in MIL using embedded probes. The aim here is to show that we can easily and quickly implement a performance tool without having to actually manipulate complex data structures. The example contains two main sections, namely, runtime code and the events. The events part first specifies

```
fct_iter = Iterator:new(-1);

this:setRunDir("output_path/");
mb = this:addBinaryMain("./bt.S");
mb:setOutputSuffix("_i");
mb:setProperty("instru_trace_log",true);
--Program entry probe
e_exit = mb:newEvent("at_exit");
p_exit = e_exit:newProbeExt("tau_cleanup","libTau.so");
--Instrumentation at function level
fct = mb:addFunction();
--Probe at function entries
e_entries = fct:newEvent("entries");
p_entries = e_entries:newProbeExt("traceEntry","libTau.so");
p_entries:addParamIterCurr(fct_iter);
--Special event to fill Binary:at_entry from function level
e_ape = p_entries:newEvent("at_program_entry");
p_ape = e_ape:newProbeExt("trace_register_func","libTau.so");
p_ape:addParamIterNext(fct_iter);
--Probe at function exits
e_exits = fct:newEvent("exits");
p_exits = e_exits:newProbeExt("traceExit","libTau.so");
p_exits:addParamIterCurr(fct_iter);
```

Fig. 6. TAU instrumentation file using MIL.

that we will target function instrumentation, and that we will be using runtime code written in Lua (embedded probes). Then instrumentation probes are specified targeting the entries and exits of functions. The code of these probes is defined in the first section of the script. We create a global table (each thread will have its own) that will contain timing information (incremented by each call). At the end of the execution, the table is dumped.

V. Experiments

In addition to evaluating MIL from a functional standpoint, it is important to compare the quality of the instrumentation framework on real applications and against existing binary rewriting tools. The OpenMP NAS parallel benchmarks [14] are used for testing instrumentation speed and execution overhead. For all experiments TAU is used for performance measurements.

Parallel applications such as OpenMP codes are optimized and transformed by the compilers in a way that hampers static binary instrumentation. Besides usual compiler optimizations, parallel regions are transformed into new functions and called through function pointers.

In the following experiments, we want to compare both the overhead of instrumented parallel NAS benchmark codes and overall the precision of the resulting data. To achieve this, the TAU profiling tool [1]is configured to use MIL, Dyninst [1], PEBIL [3] tools. While using the latest versions of PEBIL and MIL, we must mention that we used Dyninst 7.0.1 because it covered more benchmarks (more crashes with Dyninst 8). To be completely fair, we verified that there was no overhead difference between versions 7 and 8.

[1]We use the following TAU package: http://tau.uoregon.edu/tau_releases/tau-2.21.2p2.tgz and the associated PDT toolkit http://www.cs.uoregon.edu/research/tau/pdt_releases/pdt-3.19p1.tar.gz

```
-## Runtime code section ##
dyn meta_info={}; dyn results={}; dyn myfreq = 2799489000;
-- Get current clock cycles
dyn function timer() return timer() end
-- Gathers meta information initialize structures
dyn function register_function(fct_name,fid)
  meta_info[fid] = fct_name;
  results[fid] = {start = 0,inc_time = 0,calls = 0};
end
-- Start counting time at function entry
dyn function fct_start(fid)
  results[fid].start = timer();
  results[fid].calls = results[fid].calls + 1;
end
-- Stores/Accumulates time at function exit
dyn function fct_stop(fid)
  results[fid].inc_time = results[fid].inc_time +
  os.difftime(timer(),results[fid].start);
end
-- Show profiling results
dyn function fct_dump()
  print("Simple profiler results");
  print("Function name\t| Calls \t| Inclusive time");
  for id,result in pairs(results) do
    fct_name = meta_info[id];
    print(fct_name.."\t"..result.calls.."\t"..
         (result.inc_time/myfreq).." seconds");
  end
end
--## Events sections ##
--Use MAQAO builtin info_summary (function class)
function fct_info_summary(func)
  return func:info_summary();
end
--functions to be present in runtime environment
this:importC("timer","get_rdtsc","libmilrt.so");
--Events definition
fct_iter = Iterator:new(-1);

this:setRunDir("output_path/");
mb = this:addBinaryMain("./bt.S");
mb:setOutputSuffix("_iMDyn");
mb:setProperty("instru_trace_log",true);
--Program entry probe
e_entry = mb:newEvent("at_exit");
p_entry = e_entry:newProbeDyn(fct_dump);
--Instrumentation at function level
fct = mb:addFunction();
--Probe at function entries
e_entries = fct:newEvent("entries");
p_entries = e_entries:newProbeDyn(fct_start);
p_entries:addParamIterCurr(fct_iter);
--Special event to fill Binary:at_entry from function level
e_entries_ape = p_entries:newEvent("at_program_entry");
p_entries_ape = e_entries_ape:newProbeDyn(fct_register);
p_entries_ape:addParamUser(fct_info_summary);
p_entries_ape:addParamIterNext(fct_iter);
--Probe at function exits
e_exits = fct:newEvent("exits");
p_exits = e_exits:newProbeDyn(fct_stop);
p_exits:addParamIterCurr(fct_iter);
```

Fig. 7. Simple profiler all in MIL. This file contains the definition of the probes in LUA (under the runtime code section) with the definition of where to insert this instrumentation.

The experiments are run on a dual socket six-core 2.27Ghz Xeon Westmere-EP X5650 (total of twelve cores) machine. The Intel Fortran compiler (12.1.4) was used to compile the benchmarks and execute them with the OpenMP runtime. All benchmarks are compiled with the -O3 optimization level and with debug symbols (-g). The TAU profiling measurements were the same in each case. Only the overhead of invoking

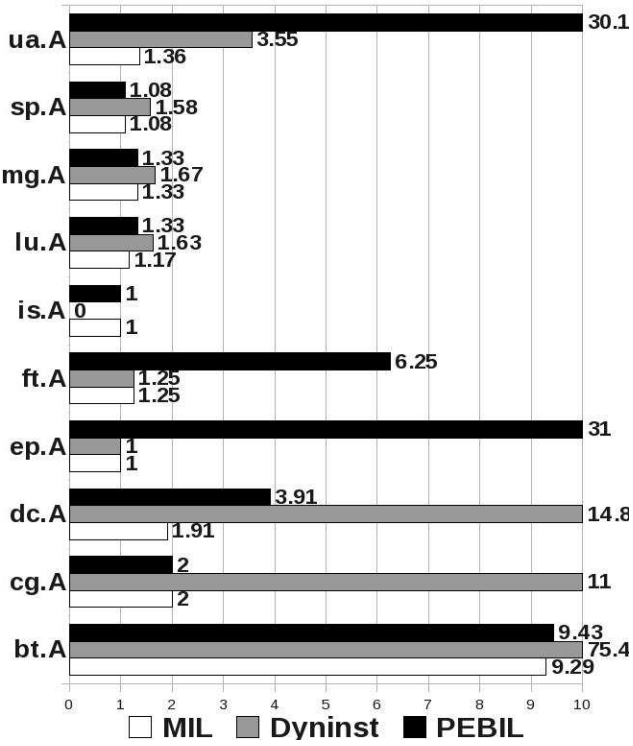

Fig. 8. Comparing overhead time on NAS OMP benchmarks for MIL, Dyninst and PEBIL using TAU. X axis reports the overhead ratio compared to the original run. Lower is better. Overhead ratios greater than 10 are cut. A zero ratio means a crash at runtime

the measurement code is different, and this is a result of how the instrumentation was done by each tool. MIL, Dyninst and PEBIL similarly use static binary rewriting. All measurements could be viewed using the TAU *pprof* profile analysis tool.

Figure 8 summarizes the obtained results of the TAU profiling tool when using MIL, Dyninst and PEBIL on the Class A OpenMP NPB suite. MIL has a lower or equivalent overhead compared to Dyninst in all cases. Interestingly, *bt*.A and *dc*.A reveal an important overhead factor for all tools. For *dc*.A, the difference between MIL and Dyninst (a factor 7) comes from the different ways to handle one-byte basic blocks (see Algorithm 1).

MIL also has a lower or equivalent overhead compared to PEBIL in all cases. PEBIL behaves like MIL in half of the benchmarks but have huge overheads for *ep*.A and *ua*.A. To a lesser extent, there is a non negligible overhead for *ft*.A and *dc*.A.

After having studied the overheads, the next aspect we must check is the quality of the results. Having a low overhead execution time means nothing if the quality of results is not in the cards. Figure 9 exhibits the output obtained with MIL, Dyninst and PEBIL profiling results for thread 1 (out of twelve) when considering the NAS OMP *bt*.A benchmark. Dyninst reports but fails to display the function names for the functions executed by the OpenMP runtime. As mentioned before, since we are able to statically identify the new OpenMP functions, we can provide a more accurate information to TAU. Since both tools find the same hotspots within the same proportions (roughly 32% for each dominant hotspot), we expect a higher number of instructions inserted at binary level by Dyninst. That is exactly what is happening since Dyninst uses a *trampoline* mechanism inducing multiple branches for one insertion. Our binary rewriting layer only inserts one level of indirection, directly adding instrumentation instructions to the displaced basic block without additional branches. Furthermore, since whole basic blocks are moved when adding instrumentation, our approach reduces the overhead when multiple instrumentations are to be performed in the same basic block. Observed results on the *bt* benchmark highlight this kind of case.

PEBIL generates less overhead when compared to Dyninst but it actually fails to detect functions executed by the OpenMP runtime as shown in Figure 9.c. According to the results, thread 1 does not consume more than 2% of the wall execution time.

VI. Related Works

The approach we presented in this paper is based on static binary instrumentation techniques and in particular focuses on instrumentation for OpenMP codes. We describe an expressive language for instrumentation that can be used by existing performance analysis tools and provides some unique features.

Concerning instrumentation techniques, we propose an approach very similar to what is presented in Dyninst [1], [16]. Also from the Dyninst Project, the DynC language [17] is a subset of C with a domain specification for selecting the location of a resource. Compared to MIL, user-defined filters in DynC can only be of limited types of statements (assignments, if-then-else, calls to ”mutatee” functions). In MIL, these filters are written in Lua language with MAQAO API using all features of the Lua language itself (loops, function declaration, control structures) along with call to external functions, inline assembly and all the other features which are specific to instrumentation. More recently Dyninst Project introduced PatchAPI which is an intent to perform new types of instrumentation through CFG annotation.

PEBIL [3] or Saxena *et al.* [18]. More precisely, Dyninst uses two trampolines (two branches) before reaching the instrumentation code. This is due to the capacity of Dyninst to add/remove at runtime instrumentation code. We choose instead to have a static approach, able to insert code with low overhead. This difference shows on the previous section.

Work of Saxena *et al.* [18] uses an offline approach for binary disassembling and a backend based on nasm assembler for generating machine code for new instructions introduced during rewriting. With MIL, all assembly instructions added or modified are directly modified in the binary form (no textual representation). This enables assembly binary instruction modifications and injection.

Comparing PEBIL [3] and our work, PEBIL resorts to whole function relocation in order to apply overhead reduction techniques. However, function relocation does not work with OpenMP codes, since function pointers are passed to the OpenMP runtime. MIL is able to handle OpenMP codes because it uses block relocation and advanced static analyses

```
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0          256     1:04.767           1        1012   64767483 .TAU application
 32.4       20,744       21,005         201      207576     104503 x_solve#omp#loop#1
 32.0       20,525       20,720         201      155905     103089 y_solve#omp#loop#1
 31.5       20,260       20,427         201      136524     101628 z_solve#omp#loop#1
  2.2        1,402        1,402         202           0       6942 compute_rhs#omp#region#1
  1.0          515          666           2      100001     333030 initialize#omp#region#1
  0.2          150          150      100001           0          2 exact_solution
  0.2          138          138      100001           0          1 binvcrhs [THROTTLED]
```

(a) TAU profiler using MIL

```
---------------------------------------------------------------------------------------
%Time    Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0           63     8:48.699           1        1012  528699937 .TAU application
 33.1     2:53.907     2:54.752         201      136524     869416 void targ4189a1()
 33.0     2:52.955     2:54.255         201      207576     866943 void targ413191()
 32.8     2:52.373     2:53.392         201      155905     862647 void targ414bf1()
  0.7        2,656        3,541           2      100001    1770777 void targ402bae()
  0.3        1,378        1,378         202           0       6826 void targ40c003()
  0.2          896          896           1           0     896002 void targ4042de()
  0.2          885          885      100001           0          9 void exact_solution()
```

(b) TAU profiler using Dyninst

```
Exclusive    Inclusive       #Call      #Subrs  Inclusive Name
              msec   total msec                          usec/call
---------------------------------------------------------------------------------------
100.0     1:05.057     1:05.900           1      607742   65900127 .TAU application
  0.2          153          153      100001           0          2 exact_solution_
  0.2          137          137      100001           0          1 binvcrhs_ [THROTTLED]
  0.2          132          132      100001           0          1 matmul_sub_ [THROTTLED]
  0.2          130          130      100001           0          1 matvec_sub_ [THROTTLED]
  0.2          112          112      100001           0          1 lhsinit_ [THROTTLED]
  0.2          108          108      100001           0          1 binvrhs_ [THROTTLED]
  0.1           59           59         815           0         74 __kmpc_barrier
```

(c) TAU profiler using PEBIL

Fig. 9. Comparison of `pprof` tool output for thread 1 of NAS OMP *bt*.A benchmark when MIL, Dyninst and PEBIL tools being used by TAU profiler

to detect interleaved functions. Andrew Bernat *et al.* [19] also suggest that the conservative approach of PEBIL is dangerous because it assumes that particular instructions (e.g., calls) are safe to move without transformation. PEBIL focuses onproviding a way to insert code snippets (avoiding a call in the trampoline) and minimizing context saving. MIL offers similar mechanisms with the `nowrap` option, reducing context saving and being able to directly specify inline assembly code. Moreover, PEBIL is an API-oriented approach which is less flexible than an instrumentation language.

For OpenMP performance analysis, POMP [20] proposes a performance monitoring interface for OpenMP. Tools implementing this interface, such as OmpTrace[21] based on Dynamic Probe Class Library[22] or OPARI[13] are based on source to source modifications, with the inherent known limitations: instrumentation may prevent compiler optimization, the code analyzed is not the code the user want to analyze. In INTONE [23], OpenMP directives are directly instrumented by the compiler. The approach taken with MIL is to provide instrumentation able to capture per thread information. OpenMP parallel loops, OpenMP API functions can therefore be instrumented and results of the instrumentation can expose OpenMP parallel execution to the performance system. Besides most compilers implement directives by inserting calls to the runtime. This is dependent of the OpenMP runtime but can provide a larger implementation of the POMP interface. Going further and capturing runtime decisions (size of chunks for instance) is not handled by MIL so far.

Finally, several instrumentation languages have been proposed in the past [6], [7], [8]. Atune-IL instrumentation language corresponds to pragma inserted in the source code. These pragmas are then handled to the instrumentation tool. This approach enforces recompilation of the application. Mussler *et al.* propose an instrumentation tool driven by configuration files. The filters used in their instrumentation language do not handle blocks or instructions and value profiling does

not seem possible, as it is in our tool. However, they propose predefined filters using some more elaborate metrics on the code. Predefined filters may apply to limited class of applications, but selecting *a priori* which parts of the code are of interest is in general intractable. This is not the approach taken here in MIL. Besides, their tool is based on Dyninst for binary rewriting.

VII. Conclusion

In this paper, we present MIL, a rich instrumentation language that reduces the complexity of writing performance analysis tools for high performance computing. Using static binary instrumentation that does not require additional compilation pass, MIL proposes a rich interface to instrument applications for a large range of uses, from profiling of functions, hardware counter analysis to value profiling. MIL provides a filtering mechanism to instrument only some specific code fragments, from functions, loops, to individual assembly instructions, and different types of instrumentations can be performed simultaneously on different code fragments. In this paper, we illustrated the capabilities of MIL on parallel OpenMP applications. Besides, a scripting mechanism relying on an API for code analysis offers the possibility to extend instrumentations and performance analyses.

We believe MIL offers a unique way to implement and investigate new performance analysis techniques, able to tackle the challenging complexity of performance tuning for multi-threaded applications. We have demonstrated the flexibility of MIL through the study of different scenarii, exploring different granularities and through its integration in TAU performance analysis framework. The MIL interpreter offers a rich abstraction layer based on static analysis and a robust binary rewriting tool. Execution overheads have been evaluated on NAS Benchmarks and compared to Dyninst and PEBIL, similar instrumentation frameworks. They show that the instrumentation provided by MIL has a low overhead.

For future work, we plan to extend the language to support OpenMP events. We also plan to introduce an iterative instrumentation approach in order to automatically take into account previous profiles. It may be a key feature in designing systematic performance evaluation tools, refining at each step the identification of performance issues.

References

[1] B. Buck and J. K. Hollingsworth, "An api for runtime code patching," *Intl. Journal on High Performance Computing Applications*, vol. 14, pp. 317–329, November 2000.

[2] D. Barthou, A. Charif Rubial, W. Jalby, S. Koliai, and C. Valensi, "Performance tuning of x86 openmp codes with maqao," in *Tools for High Performance Computing 2009*, M. S. Mller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds. Springer Berlin Heidelberg, 2010, pp. 95–113.

[3] M. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "Pebil: Efficient static binary instrumentation for linux," in *IEEE Intl. Symp. on Performance Analysis of Systems and Software*, 2010, pp. 175–183.

[4] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Redd, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation*. ACM Press, 2005, pp. 190–200.

[5] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, vol. 42. ACM Press, 2007, pp. 89–100.

[6] J. K. Hollingsworth, O. Niam, B. P. Miller, Z. Xu, M. J. R. Goncalves, and L. Zheng, "Mdl: A language and compiler for dynamic program instrumentation," *ACM/IEEE Intl. Conf. on Parallel Architectures and Compilation Techniques*, vol. 1525, pp. 201–212, 1997. [Online]. Available: http://portal.acm.org/citation.cfm?id=522659.825654

[7] C. A. Schaefer, V. Pankratius, and W. F. Tichy, "Atune-IL: An instrumentation language for auto-tuning parallel applications." in *Euro-Par Conference*, ser. Lect. Notes in Computer Science. Springer-Verlag, 2009, pp. 9–20.

[8] J. Mußler, D. Lorenz, and F. Wolf, "Reducing the overhead of direct application instrumentation using prior static analysis," in *Euro-Par Conference*, ser. Lect. Notes in Computer Science, vol. 6852. Springer-Verlag, Sep. 2011, pp. 65–76.

[9] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Intl. Journal on High Performance Computing Applications*, vol. 20, pp. 287–331, 2006.

[10] MAQAO, "Modular assembly quality analyser and optimizer," http://www.maqao.org.

[11] P. C. U. of Rio de Janeiro in Brazil, "Lua is a powerful, fast, lightweight, embeddable scripting language." [Online]. Available: http://www.lua.org

[12] M. Pall, "Luajit." [Online]. Available: http://luajit.org

[13] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, "Towards a performance tool interface for openmp: An approach based on directive rewriting," in *Workshop on OpenMP*, 2001.

[14] "Nas parallel benchmarks," http://www.nas.nasa.gov/publications/npb.html.

[15] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady, *SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance*. Springer-Verlag, 2001, vol. 2104, pp. 1–10. [Online]. Available: http://www.springerlink.com/index/80FUKBECRGXM8GKW.pdf

[16] A. R. Bernat and B. P. Miller, "Anywhere, any-time binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, ser. PASTE '11. New York, NY, USA: ACM, 2011, pp. 9–16.

[17] Dyninst, "Dync language description." [Online]. Available: ftp://ftp.cs.wisc.edu/paradyn/releases/release7.0/doc/dynC_API.pdf

[18] P. Saxena, R. Sekar, and V. Puranik, "Efficient Fine-Grained Binary Instrumentation with Applications to Taint-Tracking," in *ACM/IEEE Intl. Symp. on Code Optimization and Generation*, Apr. 2008.

[19] A. R. Bernat, K. Roundy, and B. P. Miller, "Efficient, sensitivity resistant binary instrumentation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 89–99.

[20] L. D. Rose, B. Mohr, and S. R. Seelam, "Profiling and tracing openmp applications with pomp based monitoring libraries." in *Euro-Par Conference*, ser. Lect. Notes in Computer Science. Springer-Verlag, 2004, pp. 39–46.

[21] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J. Vetter, "A dynamic tracing mechanism for performance analysis of OpenMP applications," in *Workshop on OpenMP applications and tools*, Jul. 2001.

[22] L. DeRose, T. Hoover, and J. Hollingsworth, "The dynamic probe class library - an infrastructure for developing instrumentation for performance tools," in *IEEE Intl. Parallel and Distributed Processing Symposium*, 2001.

[23] E. Ayguadé, M. Brorsson, H. Brunst, H. C. Hoppe, S. Karlsson, X. Martorell, W. E. Nagel, F. Schlimbach, G. Utrera, and M. Winkler, "OpenMP Performance Analysis Approach in the INTONE Project," in *Workshop on OpenMP*, 2001.