Handling Multi-Versioning in LLVM: Code Tracking and Cloning

Alexandra Jimborean, Vincent Loechner, Philippe Clauss

Handling Multi-Versioning in LLVM: Code Tracking and Cloning

Alexandra Jimborean Vincent Loechner Philippe Clauss
CAMUS group, INRIA Nancy Grand Est and LSIIT, Université de Strasbourg, France
{alexandra.jimborean,vincent.loechner,philippe.clauss}@inria.fr

Abstract

Instrumentation by sampling, adaptive computing and dynamic optimization can be efficiently implemented using multiple versions of a code region. Ideally, compilers should automatically handle the generation of such multiple versions. In this work we discuss the problem of multi-versioning in the situation where each version requires a different intermediate representation. We expose the limits of nowadays compilers regarding these aspects and provide our solutions to overcome them, using the LLVM compiler as our research platform. The paper is focused on three main aspects: tracking code in LLVM IR, cloning, and communication between low-level and high-level representations.

Aiming at performance and minimal impact on the behavior of the original code, we describe our strategies to guide the interaction between the newly inserted code and the optimization passes, from annotating code using metadata to inlining assembly code in LLVM IR. Our target is performing code instrumentation and optimization, with an interest in loops. We build a version in x86_64 assembly code to acquire low-level information, and multiple versions in LLVM IR for performing high-level code transformations. The selection mechanism consists in callbacks to a generic runtime system. Preliminary results on the SPEC CPU 2006 and the Pointer Intensive Benchmark suite show that our framework has a negligible overhead in most cases, when instrumenting the most time consuming loop nests.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization, Run-time environments

Keywords Multi-versioning, code tracking, cloning, LLVM IR, x86_64 assembly

1. Introduction

With the increasing complexity of available hardware, including multicores and co-processors, multi-versioning has become a classical technique for using efficiently all available resources [4, 9, 11, 16, 23, 27]. It consists in compiling and embedding in the binary different versions of a hot region of code. It is particularly adequate for periodic instrumentation, adaptive version selection and dynamic optimization in general.

The motivation of this work is to build a framework including a compiler and a runtime system able to switch from a version of a code sample to another dynamically, without interrupting the execution. Our main goal is to provide support for advances in the following fields:

- instrumentation and sampling: instrumenting code usually induces a huge overhead (typically 10x at least, and up to 1000x for complex instrumentations [12]). Sampling is a classical technique to minimize overhead, that consists in running the instrumented instructions only during some parts of the execution time. An efficient technique to implement this strategy [1, 7] is to generate two (or more) versions of a code at compile time, one instrumented and the other one non-instrumented (or some partially instrumented). The runtime system can then periodically switch from a version to another to active/deactivate instrumentation.
- adaptive version selection: in some cases, the performance ratio between several versions of a code heavily depends on dynamic factors, such as the input data size, the processor load and number of available cores, or even the architecture itself when distributing a multi-platform executable. Thus, the best performing version to be executed at a given time on a given computer cannot be determined statically by the compiler, but should rely on a dynamic selection process. The runtime system may switch from a version of code to another, depending on the dynamic factors [4, 25].
- dynamic optimization: further, some parts of the code can be compiled at execution time (JIT compilation) to be optimized according to the current context.

Multi-versioning is an efficient technique for implementing such dynamic processes. However, in order to be transparent to the user, it requires the compiler to generate different versions of a piece of code automatically, and to customize it to be handled by a runtime system. It relies on two main features: tracking the interesting regions of code from the source to the assembly; and cloning those regions in the IR or in assembly.

Code tracking is a classical problem in debuggers, and it is well known that compiler optimizations makes source-level tracking and debugging a difficult problem [2, 26]. As we are interested in performance, this is an important issue: we want to run full optimizations (typically -O3) and still be able to track some regions of the source code till the assembly generation. Some work has been done on dynamic optimization [18–20] with a limited scope of optimizations, and mostly in virtual machines. We address a similar problem, but in the case of a classical compilation chain, where an executable file is generated from source code by the compiler.

There have been some experiments in introducing multi-versioning in compilers, in particular the *Interactive Compilation Interface (ICI)* [17], being included in *gcc 4.5* [16]. This is closely related to our work, sharing the same objectives. However, this framework implements multi-versioning and cloning at function level only, while we want to perform fine grain adaptiveness. For example, it does not allow to efficiently activate/deactivate instrumentation at loop-level, some iterations of a loop nest being instrumented while the other ones are not: the proposed approach would require a call to the runtime system in the body of the loop to be evaluated at each iteration, thus inducing a high overhead.

We implemented our proposal in the LLVM compiler suite [22] and developped an associated runtime system. We use a modified *Clang* frontend [6], to handle a specific *#pragma* marking the regions of code of interest. We attach metadata to those regions in the IR to track them till code generation. For duplication, we extract them in functions and clone them, as a function is the minimal unit that can be compiled independently by LLVM. Finally, we insert long jumps in x86_64 assembly and add a specific extra segment in the binary file for the runtime system to be able to find back those jumps and patch them when needed.

The paper is organized as follows. We give an overview on the requirements for a compiler to handle multi-versioning in section 2.

We present some related work in section 3. LLVM is briefly introduced in section 4. In section 5, we present how we use metadata to track regions of code from the source to the cloned versions in LLVM. Low-level problems, implying interaction between IR and x86_64 assembly code are addressed in section 6. Finally, we conclude in section 7.

2. General issues

Whether the multiple versions are prepared statically or generated dynamically, the main steps in achieving multi-versioning would be the following:

- identify the code regions marked for processing
- clone the regions
- customize each clone to create different versions
- build a mechanism to enable switching between versions.

In what follows, we address the open questions and the limits of modern compilers, concerning multi-versioning.

Extending the intermediate representation vs using annotations. Compilers use an internal intermediate representation for manipulating code at compile time. The question that arises is how to delimit interesting code regions translated into this form.

The naive approach is to use *dummy* instructions as barriers. Namely, instructions that already exist, do not modify the semantics of the code, and are recognized by the compiler as marking the beginning and the end of the region. The great disadvantage is the implication that one has to find a particular instruction that takes this role in all cases, hence, it is not used anywhere else in the code, which is rather a strong assumption.

Another option is to extend the internal representation with additional instructions, having the role of barriers. The drawback is that the compiler has to be rewritten to accept new instructions. Moreover, these instructions might influence the code generator or even prevent some optimizations. Also, in both approaches, using barriers is not a viable strategy with higher optimization levels due to instruction reordering. On the other hand, since they are included in the set of instructions, they are not eliminated during optimization phases, which is a very important aspect, as explained in the following.

At last, some compilers allow including annotations in the intermediate representation, such as metadata carried by instructions. This seems to be the most inoffensive solution, as the code generator and the optimization phases are not disturbed. Thus, one can attach metadata to all instructions residing in the interesting regions, without influencing the transformation phases or paying the price of instruction reordering. However, not all optimizations preserve the attached information and it might be difficult, after optimization, to recover the code originally marked for multi-versioning.

Our solution to this problem is discussed in section 5.

Higher level vs lower level IR. Choosing a high- or a low-level intermediate representation is an open discussion, both forms presenting advantages and disadvantages. For code manipulation purposes, such as loop transformations or various high-level optimizations, preserving high-level information available in the source code significantly facilitates the process. On the contrary, for retrieving low-level information, for instance register or memory location accesses, one requires an intermediate representation that makes this information available or easy to track. Nevertheless, a low-level internal representation is not general enough to cover all architectures and is not suitable for most of the compilation stages.

In multi-versioning, choosing one representation or another might be a challenge, especially if each version is designed with a different purpose. For instance, one version may be tailored to perform low-level instrumentation, and another to apply the results of the instrumentation stage and perform high-level code optimizations. Therefore, each version should be manipulated into the most convenient representation.

We present our reasoning in these matters in the ending of section 5.

Communication between high- and low-level representations. Not only each version may be represented in a different IR, but, in addition, multi-versioning implies the presence of a runtime system, able to decide dynamically the version to be executed. Consequently, there are several situations where control flow between high- and low-level representations must be supported:

- communication between versions (in distinct representations),
- communication between versions and embedding code (in distinct representations),
- communication with the runtime system.

Nowadays compilers do not permit these types of communication, as they do not handle control flow entering or exiting lower level representations, such as inline assembly. Inline assembly is expected to 'fall through' to the following code. The gcc 4.5 compiler offers some support in these matters. Namely, it handles jumps from inline assembly to labels defined in C, but not the other way around. Also, jumps from an assembly code to another are not supported. To overcome this problem, one has to overwrite by hand part of the control flow graph – expressed in the higher intermediate representation – with branches generated from inline assembly code containing labels and jumps. Special attention must be given, as compilers do not parse the inline assemblies for their semantics.

In depth details concerning this approach are given in section 6.

Inserted code should not disturb the behavior of the original code. Inserting additional code, such as instructions for tracking or the mechanism to switch between versions, might have a negative impact on performance. Compilers must be tailored to generate multiple versions in a manner which minimally influences the behavior of the original code and does not degrade the result of the optimization phases. On the other hand, aggressive optimizations could lead to alterations of the inserted code.

Compilers must efficiently manage the interplay between the inserted code and the optimization passes.

We address this aspect throughout all stages of code manipulation. Details can be found in sections 5 and 6.

3. Related work

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.

Solutions have been proposed [2, 26] addressing the well-known *code location problem*, locating an original statement in the optimized code, and the *data-value problem*, retrieving the value of a variable which is not available due to code modifications. 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.* [2] as a method for acquiring extended debugging information, communicated from one optimization phase to another. It is thus demonstrated that by annotating the symbol table and creating maps between the source code and the optimized code, it is possible to highlight optimizations as loop interchange, loop invariant optimizations or to track the value of a variable, regardless if it resides in memory or in a register. Van Deursen *et al.* [26] give an efficient implementation method for origin tracking, which is a method for incrementally computing the relation between pieces of the program such as identifiers, expressions, or statements. However, this is presented only as a prototype for the construction of bi-directional mappings between source programs and optimized code.

More recent and daring work tackling debugging of dynamically optimized code has been reported [18, 20]. Kumar *et al.* [20]

describe a set of techniques to monitor code transformations performed by a dynamic optimizer and to communicate this information to a native debugger. The steps are first to create a transformation descriptor of an instruction or data variable, then generate debug information, and to transmit it to the native debugger. 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. A strategy to avoid this problem is offered by the developers of Java HotSpot compiler [19] who interpret the unoptimized code during debug sessions. In [18] the focus is on reporting the expected values of source variables computed in the optimized code.

In the gcc compiler [13], 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 [6] 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. Much work has been oriented towards this research direction [1, 5, 7, 10, 11, 14–16, 23, 24].

Cloning, multi-versioning, instrumentation by sampling. Multi-versioning is a widely adopted strategy to reduce the cost of code instrumentation by sampling [1, 5, 7, 14, 15, 24]. A selection mechanism periodically switches execution between a number of versions embedding instrumentation code and the original version. Efforts to reduce the runtime overhead and an efficient framework for instrumentation by sampling are presented in [1, 7]. Chilimbi and Hirzel [5, 15] 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 [8] for capturing sequences of data references (dynamic executions of loads or stores).

An interesting use of sampling is presented by Chilimbi and Hauswirth [14] 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 [1] 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.

Multi-versioning in optimizations. Fursin *et al.* [11] present a framework for continuous compilation within gcc for cloning code sections, applying various optimizations on the clones, and randomly selecting one version for execution. Evaluation is done using *gprof* profiler. Luo *et al.* [23] use the Open64 4.0 compiler and the Interactive Compilation Interface [17] to select a limited number of optimized versions across all datasets, avoiding performance loss or code-explosion. Heuristic methods are employed to find a representative set of optimizations, and machine learning techniques correlate characteristics of the datasets with the optimized versions. Interactive Compilation Interface (ICI) [17] has been developed with the aim of providing access to the internal functionalities of compilers. Extensions to ICI [16] 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.

ADAPT [27] is a high-level adaptive optimization system. It proposes a domain specific language allowing the user to specify the heuristics for applying optimizations dynamically. ADAPT reads the descriptions and generates the executable code for a target application to apply the user-defined techniques. The optimization targets of ADAPT are the loop nests containing no I/O operations and no function calls. Considering the runtime information, a decision is taken whether optimization would be profitable. Applying the user defined heuristics, the set of optimizations is chosen and new code versions are generated. Before executing a code section, the framework verifies if experimental versions are available, otherwise the best known version is executed. Compared to our proposal, ADAPT requires a lot of source code modifications by the user, and to our knowledge no automatic multi-versioning using ADAPT has been reported.

4. LLVM Intermediate Representation

The LLVM Intermediate Representation (LLVM IR) is built upon the Static Single Assignment (SSA) form and it confers type safety, low-level operations and flexibility. An increasing number of high-level languages may be translated into the LLVM representation, which is the internal language used throughout all phases of the LLVM compilation strategy.

The LLVM code representation is designed to be light-weight and low-level, for efficient compiler transformations and analyses. On the other hand, it provides type information and supports mapping higher level information from the source code, which facilitates the development of various optimization passes [21].

Although LLVM IR already offers support for embedding high-level information, new methods have been developed to include annotations, debugging information or to attach metadata to the IR.

Until LLVM 2.6, debug information represented a channel from the front-end to the DWARF emitter, without being included in the executable code. Moreover, it was encoded using global variables with tags, which prevented a number of optimization passes and was very expensive from the time and memory footprint viewpoint.

An important enhancement in these regards was brought in LLVM 2.7, with the development of the metadata. The main goals were to provide the means to attach information in the IR, without influencing the optimizers (unless metadata was explicitly specified for this). Also, the cost, in terms of time and memory use, has significantly been reduced. Metadata is attached to instructions and improves the implementation of the debug information.

The optimizers do not have to be aware of the metadata, but the other side of the coin is that they do not preserve it. Hence, metadata information might be lost in code transformations. Special care might be taken to update metadata information during optimization phases, nevertheless, this is not suitable to all passes. Especially when aggressive code transformations are performed, such as with -O3 optimization level, tracking code and metadata is particularly difficult.

LLVM provides a wide range of functionalities to facilitate code transformations. Regarding multi-versioning, LLVM offers support for cloning, however limited. There exists a suite of *clone* utilities, able to create copies of instructions, basic blocks or functions, but no correlation is made between values in the source and in the clones. Therefore, LLVM cloning can only be applied in some very specific situations.

5. Tracking code in LLVM IR using attached metadata

From C/C++ to LLVM IR with metadata. Our goal is to enable the compiler to create a series of code versions, among which a runtime system can dynamically select one or another for execu-

tion. In this respect, a new pragma is defined and inserted in the source code for delimiting the code regions of interest.

One has to track the code annotations from the source level and identify the equivalent code regions in the intermediate representation. There are several solutions to achieve this task in LLVM.

Firstly, adding a new intrinsic or a new instruction in LLVM, designed with no other purpose, but to mark the beginning and the end of the region. However, there is a number of disadvantages resulting from this approach:

1. Adding new instructions is discouraged in LLVM, as all passes have to be updated and maintained to work with the new functionalities. Since LLVM already includes a considerable number of analysis and transformation passes, this will surely lead to a significant amount of work [3].
2. In case a new functionality can be expressed as a function call, then adding an intrinsic is a more elegant and simple solution. Intrinsics do not require to update the optimizers, but the LLVM IR and the code generator must be extended to support it. However, if the intrinsic does not have any side-effect, the optimizers will remove it.
3. Barriers are a reliable solution when no optimizations are applied, but become unsafe with a higher optimization level. The strongest argument against using barriers is that instructions belonging to the region might be hoisted above or sank below the barriers (or, vice-versa, instructions that did not originally belong to the region, can be included).

Secondly, attaching metadata to all instructions in the code region. This strategy gives an answer to all the problems displayed above, since LLVM already offers support for metadata, it does not influence the optimizers and it is not disturbed by instruction reordering.

In our work we use the metadata based method. The difficulties of tracking code throughout optimization phases is that metadata information is not preserved, and that code suffers significant transformations. For instance, if one marks the instructions building up a loop, after running the 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 the 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 is considered. However, in this manner, we ensure that all instructions of the targeted code region are safely enclosed.

Cloning Once the region is identified, several clones are created. In LLVM IR, a set of restrictions is strictly imposed as SSA form must be preserved:

1. Instructions and their return values are equivalent. Hence, in the example:

   ```
   %tmp = load i32* %i, align 4
   %inc = add i32 %tmp, 1
   ```

   instruction `%inc` uses the value stored in `%tmp`. When cloning using the LLVM *clone* functions, the result is:

   ```
   %tmp_clone = load i32* %i, align 4
   %inc_clone = add i32 %tmp, 1
   ```

   whereas the target is:

   ```
   %tmp_clone = load i32* %i, align 4
   %inc_clone = add i32 %tmp_clone, 1
   ```

   Note that instructions outside the region are not cloned (`%i`). Nevertheless, they may be used both by instructions belonging to the region and by their clones. For instance instructions `%tmp` and `%tmp_clone` use `%i`, but `%i` is not cloned.
2. Each value (instruction, basic block) must have a unique parent. It cannot be duplicated in the same function, nor copied in a new one, unless it is removed from its parent function. For this reason, we cannot simply insert each value twice, but we need to create and maintain individual clones.
3. Each value must dominate all its uses.

Our proposal for cloning is to create a map between all instructions and their clones; similarly, for all cloned basic blocks. In figure 1A, blocks BB1 to BB4 belong to the region marked for multi-versioning and BB1_clone to BB4_clone are their clones. As clones are created, a map containing the pairs of original and cloned values is maintained, e.g. BB1 $\rightarrow$ BB1_clone. Using the *clone* function available in LLVM, each instruction or basic blocks will use the same values as its original version. Hence blocks BB1_clone to BB4_clone will point to blocks BB1 to BB5, instead of using the cloned versions of the blocks from the region. Namely, they should point to BB1_clone to BB4_clone, since these are the corresponding clones, and to BB5. As BB5 does not belong to the region, it is used both by BB3 and BB3_clone.

The objective is to rebuild the control flow graph between the clones, as illustrated in figure 1B. Similarly for instructions, each clone that uses a value (either instruction or basic block) from the region is updated to use its corresponding cloned version. For instance, block BB1_clone branches to block BB2. As BB1_clone is identified as a clone version, and BB2 is an original version belonging to the region, the edge BB1_clone to BB2 is suppressed and replaced by the edge BB1_clone to BB2_clone. In other words, the clone BB1_clone is updated to use the value BB2_clone, instead of the original version BB2. On the other hand, a clone version is allowed to use an original value which does not belong to the region: BB3_clone branches to block BB5.

When clones are created, they are not automatically assigned a parent. We do this manually, by inserting each cloned basic block in the same parent function as the original version. In the case of instructions, they are inserted in the corresponding clone of the basic block, and not in the original block.

With this, we achieve to create a copy of the code marked for multi-versioning, while fulfilling the above mentioned constraints.

In order to customize the copies, we extract each version of code in a separate function (figure 1C). Original blocks, BB1 to BB4 are extracted in function *Version_1* and replaced in the original code with a call to this function. In the same manner, the clones BB1_clone - BB4_clone are extracted in function *Version_2* and a call is inserted. We use the function *LLVM::ExtractCodeRegion*, which automatically identifies the values that must be sent as parameters and updates the values used outside of the function. For preserving the SSA form, *LLVM::ExtractCodeRegion* will create multiple copies of the values used outside the function, updated with the results computed in the body of the function. Consequently, it is highly important to replace the uses of the original values in the clones, before extracting each code version in a separate function. Not doing so leads to violation of constraint number three, because copies of the original values would be created.

Multi-versioning Having created the clones and extracted them in separate functions, we need a mechanism to allow the runtime system to switch between them dynamically. In this respect, for each set of clones and original version, we build a basic block consisting in a condition, and branching to function calls depending on the result of its evaluation. The condition is evaluated through a callback to the runtime system, which will decide the function to be invoked. Recall that each function represents a different version (figure 1C).

Extracting versions in separate functions allows us to decide about the most convenient representation. Clones designed for

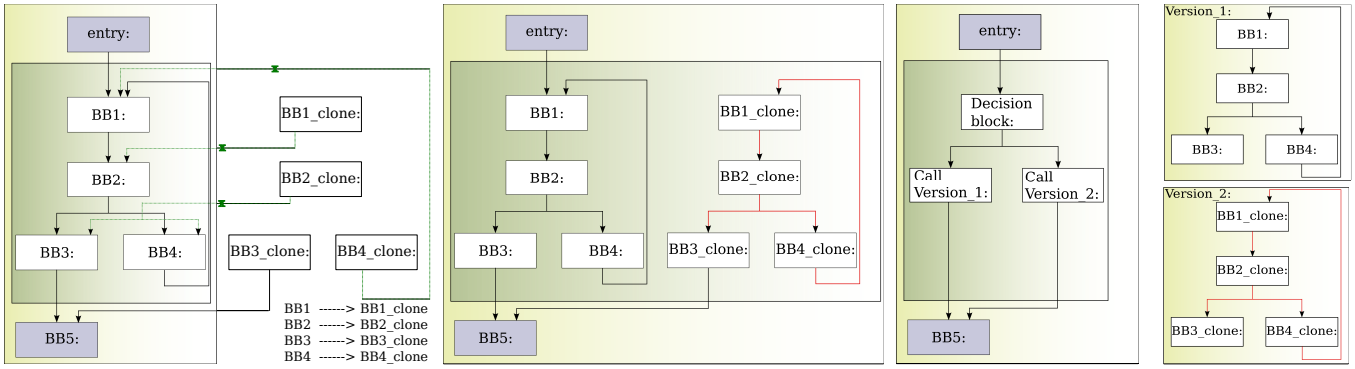

Figure 1. Multi-versioning.

high-level code transformations are represented in LLVM IR, whereas clones targeting low-level information are translated into x86_64. Each clone is compiled, customized and further 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 to x86_64 form. The clones represented in the LLVM IR are inlined back in the original code after 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 minimal, even when the most time consuming regions of code are marked for multi-versioning.

6. Handling jumps between LLVM IR and inline assembly

In our proposal, the multiple versions are generated statically and separated in new functions. Nevertheless, we designed a generic framework to manage dynamically generated code and switch between all available versions. In this respect, the switching mechanism requires to insert a call to the function containing the version selected for execution. Furthermore, the runtime system in charge with version selection is enabled to manage the various operations for which different code versions are generated.

As a challenging goal, we focus on loop instrumentation and, more precisely, on interpolating memory addresses accessed inside loop nests. For performance reasons, we tackle loop instrumentation by sampling. Consequently, the runtime system switches between the original and instrumented versions. Furthermore, the runtime system manages various operations required for this type of instrumentation, namely, processing the acquired information and computing the interpolation functions.

Callbacks. The original code, enclosing the multiple versions and the switching mechanism, is sprinkled with callbacks to the runtime system, positioned at some key-points of the program. To preserve genericness, we developed a modular runtime system, each module consisting in the set of functions required for each operation. Moreover, a callback is inserted as in figure 2.

Saving and restoring the stack red zone and the scratch registers is common to any callback, however, to stay on generic realms, we insert a call to the function located at address 0x0, belonging to the module located at address 0x0. The runtime system will patch the code with the correct addresses of the function and its corresponding module. For this reason, we insert the callback code in assembly x86_64 representation, inlined in the enclosing code in LLVM IR form. Moreover, the two instructions

```
mov    $0x0,%rdi //address of the module
mov    $0x0,%rsi //address of the function
```

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

Figure 2. callback in x86_64 assembly code.

which have to be patched, are inserted in their hexadecimal equivalent form, such that enough space is available to accommodate 64 bits addresses.

Labels and jumps. Constant communication must be ensured between the enclosing code, the multiple versions and the runtime system. In these regards, the runtime system must be able to identify the beginning and the end of each version of code, as well as the address of the code to resume execution when returning from a code version.

We mark the key-points of interest by inserting labels as inline assembly code in the above mentioned positions. In figure 3 labels *ORIG* and *END_O* are inserted as x86_64 code, inlined in the LLVM IR code to mark beginning and end of the original version. Similarly, *INSTRU* and *END_I* mark the borders of the second version.

Additionally, the mechanism that allows switching between versions is written entirely in inline x86_64 assembly code. By default, the framework is designed to execute the original version of code. In contrast, when the runtime system is available, instrumentation is enabled. For this, the runtime system patches the branch that points to the original version to point to the switching mechanism. Toggling between versions is achieved by means of a decision block that contains callbacks to the runtime system and jumps to each version of code. Since all callbacks are patched, the code is in x86_64 representation and hexadecimal form, to ensure preciseness. The code structure is depicted in figure 3.

Each callback to the runtime system performed from the decision block requires patches. As a consequence, the size of the inserted code must be fixed. Therefore, each `jmp` is replaced with a jump on a fixed number of bits, either 8 or 32 bits, which will prevent the compiler to generate variable sized `jmp` instructions, as illustrated in table 1.

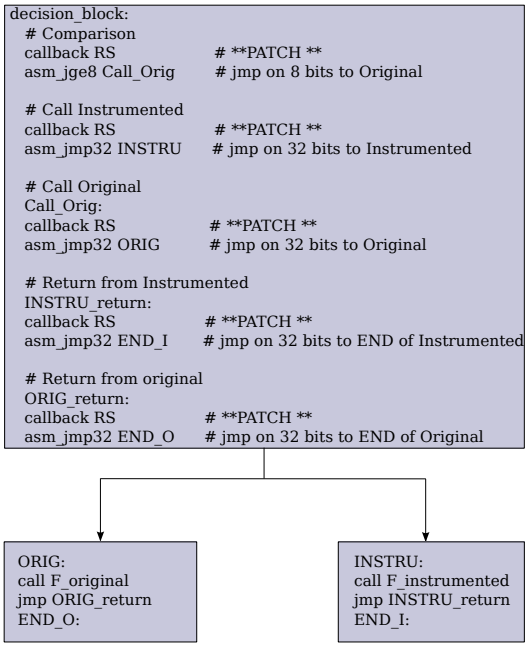

Figure 3. Code structure.

Macro	Hexadecimal form
asm_jge8 TARGET	.byte 0X7D .byte \TARGET \()-.-1
asm_jge32 TARGET	.byte 0X0F, 0X8D .long \TARGET \()-.-4

Table 1. Inline assembly code in hexadecimal representation.

However, in the LLVM IR the names of the labels created by the code generator are not yet available, also the code suffers significant transformations when converting from LLVM IR to x86_64 assembly code. In this respect, each jump inserted as inline assembly must be accompanied by a label, inserted in the convenient position in the code. What we obtain is a partial control flow graph managed as x86_64 assembly code, inlined in the LLVM IR.

Handling CFG as inline assembly code. Nowadays compilers do not allow control flow entering or exiting inline assembly code. Furthermore, inline asm is regarded as a constrained string, emptied of semantics, simply printed to the .s file when machine code is generated. Having no support from traditional compilers, the partial control flow graph in inline assembly code must be handled while preserving the original control flow graph, maintained by the compiler. The labels and jumps inside inline assembly code, partially rewrite the control flow graph, as shown in figure 4.

Blocks BB1 to BB5 and edges E1 to E3 represent the original control flow graph, maintained by the LLVM compiler. By inserting inline assembly code, we shadow part of this graph, by adding a new block, NBB, and replacing a series of edges. A new edge NE0, branching to the new block NBB, is added. Also, edges E1 and E3, originally from BB1 to BB2 and BB3, are replaced with NE1 and NE2, from NBB to BB2 and BB3. Similarly, E3 is replaced by NE3, connecting now BB5 and NBB.

Nevertheless, the inline code is not accessible to the LLVM compiler in this compilation phase, hence, partial rewriting of the control flow graph must be totally transparent. Our approach follows the guidelines below:

1. **Keep original CFG represented in LLVM IR:**
   LLVM does not allow blocks terminating with a non-terminating instruction (terminating instructions are *branch*, *switch*, *return* instructions for example). In consequence, the edges E1, E2 and E3 must be preserved, they cannot be replaced with inline assembly code.

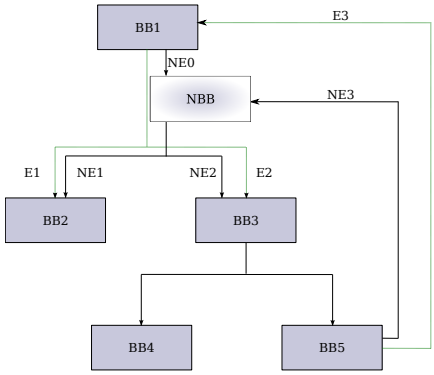

Figure 4. Control flow graph rewritten by inline code.

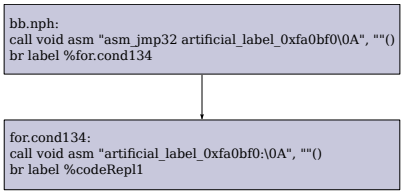

Figure 5. Control flow graph rewritten in inline code.

2. **Overwrite branches with jumps in inline code:**
   The solution at hand would be to replace edges E1-E3 with new ones, represented in the LLVM IR: NE0-NE3 represented as LLVM IR terminating instruction, rather than inline assembly code. However, as presented in subsection *Labels and jumps*, the runtime system requires a fixed sized code, in order to patch the callbacks. Using LLVM IR terminating instructions, the size of the generated `jmp` instructions may vary.

   Our strategy is to precede the branches with fixed size `jmp` instructions in inline x86_64 assembly code, such as `asm_jge32 TARGET`, previously presented. The targets of the jumps are uniquely generated labels, inserted as first instructions in the target blocks. An example is given in figure 5.

3. **Ensure that the overwritten branches are not reachable in the generated code:**
   Not only we duplicate parts of the original control flow graph with instructions represented in x86_64 assembly code, but we also change a series of branches, as illustrated in figure 4. Therefore, special care must be taken to ensure that the newly inserted edges and the ones intended to be rewritten do not interfere. Shortly, we must handle the inline assembly code, such that, in the generated machine code, the edges chosen for elimination are not reachable.

   This may prove to be a challenge, due to optimization passes that perform block fusion, instruction reordering or similar low-level code transformations. A number of optimization passes are executed by default in LLVM; they cannot and should not be disabled. Thus, it is impossible to prevent various optimizations from making multiple copies of the inline assembly code. On one hand, we aim to minimally disturb the optimization process, but, on the other hand, we require the inline code to stay unchanged.

4. **Ensure that inline assembly code is not:**

   - **duplicated**: copying inline code leads to errors due to name conflicts generated by multiple declarations of labels. To avoid it, one must update the original control flow graph such that the optimizers will not attempt to copy the x86_64 code in multiple blocks. The solution we propose is to create

a new block for each snippet of inline code. Although in the LLVM IR form, a new branch is added, when converting to machine code, the block is inlined, thus, no additional `jmp` instructions are required.

- **eliminated**: new basic blocks have to be evaluated as reachable by the LLVM compiler, otherwise they are eliminated as dead code. Namely, there must be at least one branch represented in the LLVM IR pointing to the new blocks. Jumps inserted in inline assembly code targeting labels from new blocks are not accessible, yet not recognized by the compiler. In consequence, one must alter the original control flow graph to include the new block, and, simultaneously, manage the control flow graph expressed in inline assembly code to bypass this branch.
- **relocated:** instructions reordering has an undesirable effect on our framework, unless influenced from the LLVM IR. It is of high importance to place correctly the artificial labels we insert for marking the beginning and end of each code version. Nevertheless, the code generator reserves its rights regarding the order of the instructions. For instance the LLVM IR code:

```
artificial_label:
function_call
```

is converted into x86_64 assembly code as:

```
save_machine_state_for_function_call
artificial_label:
function_call
restore_machine_state_for_function_call
```

which causes problems when jumping from the inline code to the `artificial_label`. As in the case of code duplication, the solution is to create a new block containing only the inline code. Consequently, the code generator and the optimizers place the `artificial_label` in the correct position.

5. **Minimally influence code behaviour and performance:** Since our goal is code instrumentation and profiling, we aim to grasp accurate information concerning the behaviour of the code, without degrading the performance. Nevertheless, introducing instrumentation code, as well as the mechanism that allows switching between versions, has an impact on the code generator and optimizers. In this respect, we use metadata information, the least invasive form of tracking code, and we perform multi-versioning in the LLVM IR only after the optimization phases complete. Still, there is a number of optimization passes which are run by default by LLVM before the code generation step. The interaction between these passes and the newly introduced code has an influence on both: the behaviour of the optimizers is affected by the presence of the inserted code, whereas the new code suffers transformations in the optimization process. The compromise we accept is to minimally influence the optimizers such that the code remains unchanged, which is a strict constraint for ensuring the functioning of our framework.

   From the performance standpoint, we tackle instrumentation by sampling, which motivates the need of multi-versioning, and we enable our framework to support higher optimization levels (O2, O3 - the highest available in LLVM).

Experiments. We have implemented the strategies described above to perform loop instrumentation by sampling. Namely, we aim to linearly interpolate the memory locations accessed inside the loops, when possible. To reduce the overhead, the instrumented version of the loop executes for a fixed number of iterations to acquire low-level information and then execution continues with the original version of the loop.

Our experiments, conducted on the SPEC CPU 2006 and on the Pointer Intensive benchmark suite, reveal almost negligible overhead in most cases, of less than 4%, with -O0 optimization level, when instrumenting the most time consuming functions [28]. 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 induced by using VMAD. For each instrumented loop nest, the dynamic profiling is activated each time its enclosing function is invoked.

For most programs, VMAD induces a very low runtime overhead, which is even negligible for `perlbench`, `bzip2`, `milc`, `hmmer`, `h264ref` and `lbm`. For the programs `sjeng` and `sphinx3`, the significant overheads are mainly due to the fact that the instrumented loops execute only a few iterations, but they are enclosed by functions that are called many times. Thus all iterations are run while being fully instrumented since each call represents a very low execution time. However, the profiling strategy could be improved in order to manage such cases by disabling the instrumentation after a few calls. Program `milc` shows an opposite behavior since the loops execute a very high number of iterations and each iteration executes only a few memory instructions. In such a case the runtime overhead is quite 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.

Furthermore, there is ongoing work related to supporting higher optimization levels (-O2, -O3). Currently, our framework supports -O3 optimization level, at the price of an increased overhead, 3x for a number of benchmarks, in contrast to less than 0.5% for others. We have investigated the large disproportion and additional tests prove that instructions inserted in the optimized code cause a disturbance of the optimization phase. On the other hand, executing the optimizations after inserting the instrumentation code has an undesirable effect, especially due to the *Global Value Numbering (gvn)* pass, which – among others – eliminates redundant code and leads to a new reordering of instructions. We managed to overcome this problem in some situations and we are confident that we can generalize our strategies to handle any particularities of the code.

7. Conclusions

In this article we present a technique to perform multi-versioning in LLVM IR and a selection mechanism which interacts with a generic runtime system. Code tracking is an important aspect in multi-versioning. We mark hot regions - with a focus on loops - in the source code and track them in the LLVM IR. We discuss the advantages and drawbacks of various methods for tracking regions of code, and we emphasize the use of metadata, the least invasive method of code annotation. Our goal is to minimally influence the behaviour of the original code and of the optimizers. The challenge is, however, to recover the regions marked for multi-versioning, after the code has been highly optimized.

Next, we propose a method for cloning regions of code in LLVM, such that the SSA form is preserved and the LLVM IR constraints are fulfilled. We implement cloning at a coarser granularity (code regions, loop nests), and we base our strategy on the underlying *clone* functions available in LLVM at instruction level. Each version of code is extracted in a function processed independently, converted into a suitable representation (LLVM IR, x86_64 assembly).

The selection mechanism consists in callbacks to a generic runtime system, designed to handle various types of code instrumentation and optimization, by patching the code. In the LLVM IR we inline callbacks represented either in hexadecimal format or in x64_64 assembly. Mixing LLVM IR and a lower-level representation poses many difficulties, as the compiler does not allow jumping to and from inline code. Our approach consists in managing the

Table 2. 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	code size increase	instrum. loops	instrum. instruct.	instrum. mem. accesses	linear mem. accesses
perlbench	0.073%	50%	53	3,873	404,388	8,420
bzip2	0.24%	218%	25	502	1,053	608
mcf	20.76%	213%	6	138	4,054,863	2,848,589
milc	0.081%	44%	16	195	1,988,256,195	1,988,256,000
hmmer	0.062%	63%	22	742	845	0
sjeng	182%	80%	7	662	1,155,459,440	1,032,148,267
libquantum	3.88%	21%	5	42	203,581	203,078
h264ref	0.49%	0.44%	8	349	32,452,013	30,707,102
lbm	0%	170%	7	136	358	0
sphinx3	172%	20%	5	194	78,437,958	51,566,707
anagram	-5.37%*	73%	3	53	159	134
bc	183%*	11%	4	142	302,034	243,785
ft	-8.46%*	86%	4	36	36	22
ks	29.7%*	268%	5	102	42,298	29,524

* irrelevant (short time measures)

control flow from inline assembly, by inserting labels and jumps in the optimized code.

We complement previous works by outlining a method for cloning regions of code, with a focus on highly optimized loops. We describe a mechanism that allows controlling the version of code to be executed, at the level of a loop iteration. And we propose new means of communicating with a generic runtime system, by blending LLVM IR, x86_64 assembly code. Our perspectives regard increasing the accuracy in tracking optimized code, enhancing the framework to automatically support any type of instrumentation or code optimization, and improving the performance of our framework with higher optimization levels. Our long term goal is to develop an API that allows the user to describe the instrumentation / optimization type, and, based on the provided input, to automatically generate multiple versions customized for the new functionality. Accordingly, we plan to extend the runtime system to include support for different instrumentations and for dynamic code generation.

References

[1] M. Arnold and B. G. Ryder. A framework for reducing the cost of instrumented code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI, 2001.

[2] G. Brooks, G. J. Hansen, and S. Simmons. A new approach to debugging optimized code. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI, 1992.

[3] M. Brukman, B. Jones, N. Begeman, and C. Lattner. Extending LLVM: Adding instructions, intrinsics, types, etc. `http://llvm.org/docs/ExtendingLLVM.html`.

[4] X. Chen and S. Long. Adaptive multi-versioning for OpenMP parallelization via machine learning. In *15th Int. Conf. on Parallel and Distributed Systems (ICPADS)*, 2009.

[5] T. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI, 2002.

[6] Clang. A C language family frontend for LLVM. `http://clang.llvm.org`.

[7] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *Int. Symposium on Software Testing and Analysis, ISSTA'07*. ACM, 2007.

[8] A. Edwards, H. Vo, and A. Srivastava. Vulcan binary transformation in a distributed environment. Technical report, 2001.

[9] G. Fursin and O. Temam. Collective optimization: A practical collaborative approach. *ACM Trans. Archit. Code Optim.*, 7, Dec. 2010.

[10] G. Fursin, A. Cohen, M. O'Boyle, and O. Temam. A practical method for quickly evaluating program optimizations. In *High Performance Embedded Architectures and Compilers*, LNCS. 2005.

[11] G. Fursin, C. Miranda, S. Pop, A. Cohen, and O. Temam. Practical Run-time Adaptation with Procedure Cloning to Enable Continuous Collective Compilation. In *GCC Developers' Summit*, 2007.

[12] X. Gao, M. Laurenzano, B. Simon, and A. Snavely. Reducing overheads for acquiring dynamic memory traces. In *Workload Characterization Symposium*, 2005.

[13] GCC. The GNU Compiler Collection. `http://gcc.gnu.org`.

[14] M. Hauswirth and T. M. Chilimbi. 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.

[15] M. Hirzel and T. Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM Workshop on FeedbackDirected and Dynamic Optimization FDDO4*, 2001.

[16] Y. Huang, L. Peng, C. Wu, Y. Kashnikov, J. Rennecke, and G. Fursin. 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'10)*, Pisa Italy, 2010. Google Summer of Code'09.

[17] ICI. Interactive Compilation Interface. `http://ctuning.org/ici`.

[18] C. Jaramillo, R. Gupta, and M. L. Soffa. Fulldoc: A full reporting debugger for optimized code. In *7th International Symposium on Static Analysis*, SAS '00. Springer-Verlag, 2000.

[19] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5, May 2008.

[20] N. Kumar, B. Childers, and M. L. Soffa. Transparent debugging of dynamically optimized code. In *Int. Symp. on Code Generation and Optimization*, CGO '09. IEEE Computer Society, 2009.

[21] C. Lattner and V. Adve. LLVM language reference manual. `http://llvm.org/docs/LangRef.html`.

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

[23] L. Luo, Y. Chen, C. Wu, S. Long, and G. Fursin. Finding representative sets of optimizations for adaptive multiversioning applications. In *International Workshop on Statistical and Machine learning approaches to ARchitectures and compilaTion*, Paphos Chypre, 2009.

[24] D. Marino, M. Musuvathi, and S. Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI, 2009.

[25] B. Pradelle, P. Clauss, and V. Loechner. Adaptive runtime selection of parallel schedules. Research report, Sep. 2010. `http://hal.inria.fr/inria-00534723/en/`.

[26] A. van Deursen, P. Klint, and F. Tip. Origin tracking. *J. Symb. Comput.*, 15, May 1993.

[27] M. J. Voss and R. Eigemann. High-level adaptive program optimization with ADAPT. In *PPoPP '01*, pages 93–102. ACM, 2001.

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