Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy

Salwa Kobeissi, Alain Ketterlin, Philippe Clauss

Rec2Poly: Converting Recursions to Polyhedral Optimized Loops Using an Inspector-Executor Strategy

Salwa Kobeissi, Alain Ketterlin and Philippe Clauss

Inria Camus, ICube lab., CNRS,
University of Strasbourg
Strasbourg, France

Abstract. In this paper, we propose Rec2Poly, a framework which detects automatically if recursive programs may be transformed into affine loops that are compliant with the polyhedral model. If successful, the replacing loops can then take advantage of advanced loop optimizing and parallelizing transformations as tiling or skewing.
Rec2Poly is made of two main phases: an offline profiling phase and an inspector-executor phase. In the profiling phase, the original recursive program, which has been instrumented, is run. Whenever possible, the trace of collected information is used to build equivalent affine loops from the runtime behavior. Then, an inspector-executor program is automatically generated, where the inspector is made of a light version of the original recursive program, whose aim is reduced to the generation and verification of the information which is essential to ensure the correctness of the equivalent affine loop program. The collected information is mainly related to the touched memory addresses and the control flow of the so-called "impacting" basic blocks of instructions. Moreover, in order to exhibit the lowest possible time-overhead, the inspector is implemented as a parallel process where several memory buffers of information are verified simultaneously. Finally, the executor is made of the equivalent affine loops that have been optimized and parallelized.

Keywords: Polyhedral model · Recursive functions · Automatic program parallelization and optimization.

1 Introduction

From code development to final execution on a hardware platform, a software goes through many transformation phases, making the final executable code significantly different from the initial source code. The final goal is obviously to generate an executable which is semantically equivalent to the input source code, but whose runtime behavior is satisfactory regarding execution time, code size, energy consumption, or security. Compilers apply many optimization passes on the input source code that often modify or even remove instructions, control structures, or data structures. Such code transformations may apply from the instruction level until the global code structure. A clear example of the latter kind of transformations, performed by mainstream compilers, is the transformation of tail-recursive calls into loops. Indeed, recursive programs generally suffer from not being as easily handled as other code structures like loops, for automatic optimization and parallelization. In the literature dealing with recursion optimization, either recursive functions are handled directly «as they are» [5, 10], or are firstly transformed into loops [1]. Although such approaches may provide significant performance improvements, they do not capture cases where recursive codes can be rewritten as *affine* loops, which are the most convenient loops for efficient data locality and parallelization optimizations. Affine loops are compliant with the polyhedral model [3, 4], which is a well-known mathematical framework unifying all the most important loop optimizing transformations as loop interchange, skewing or tiling.

Being a static (source-level) framework, the polyhedral model places stringent conditions on the programs that it can handle. Many programs do not directly fit the model, either because of superficial languages idiosyncrasies, or because of radical language differences (as in the case of recursive functions). It often turns out that source programs that seem not to fit the model actually have a matching behavior at runtime, at least for significant portions of their execution. The APOLLO speculative optimizer [13, 9] aims at capturing these transient polyhedral behaviors and leverage polyhedral tools to optimize them at runtime.

We present Rec2Poly, a framework devoted to the transformation of recursive codes into affine loops. Rec2Poly detects a polyhedral-compliant behavior of a target recursive code at runtime, using an offline profiling phase. When successful, it builds a semantically equivalent code where all the execution flow related to recursive functions is replaced with affine loops. Moreover, the so-generated loops are parallelized and optimized thanks to polyhedral transformations. Since this transformation is based on a runtime profiling, its validity is not ensured whatever the input data. Thus, it is speculative and a fast runtime verification mechanism is also generated, which follows an inspector-executor scheme [12, 11]. At runtime, the inspector verifies that the affine loops are valid regarding the current execution context. If the inspector does not detect an unpredicted behavior, the executor launches the optimized parallel loops. Note that the original recursive code is launched in parallel with the inspector, in order to save the time-overhead of Rec2Poly if the affine loops are not valid in the current context. We firstly presented our proof of concept of Rec2Poly's analysis, profiling and recursion to optimized loops transformation phases in a previous work [8]. In this paper, we extend these phases in addition to introducing the new embedded verification feature based on an inspector-executor strategy.

The originality of our approach is twofold: (1) Seeking a polyhedral-compliant runtime behavior in recursions; and (2) using an inspector-executor scheme not only verifying memory access patterns, as it is usually done when using this scheme, but also verifying the control flow as being compatible with affine loops. Note that our approach can also be seen as dynamic code rewriting.

The paper is organized as follows. Next Section presents an overview of the Rec2Poly framework, while the following sections provide more details on its phases. Section 3 explains how important functions involved in the recursions are automatically identified, as well as the important instructions of these functions. It also presents essential aspects regarding memory instructions. In Section 4, it is explained how variables which are local to functions are globalized for code instrumentation, in order to promote memory accesses following affine functions of loop indices. Rec2Poly mostly relies on a software tool called NLR (Nested Loop Recognition) whose main features are recalled in Section 5. Generation of the inspector program is presented in Section 6, while the generation of the optimized affine loops is presented in Section 7. Experiments are presented in Section 8 and related work in Section 9. Conclusions are given in Section 10.

2 Overview of the Rec2Poly Framework

The main phases of Rec2Poly are depicted in Figure 1. Every analysis, transformation and code generation phase has been implemented as passes of the mainstream compiler Clang-LLVM[1].

A target recursive code is first deeply analyzed in order to identify the recursive functions as well as the functions that may be invoked by, or that may invoke, the recursive ones. We call these identified functions as *impacting functions*. Then, the Backward Static Slice (BSS) of every memory store instruction in these functions is built, by collecting the identifiers of every Basic Block (BB) that contains at least one instruction involved in the computation of the target memory address or in the computation of the stored value. We characterize such basic blocks as *impacting basic blocks*.

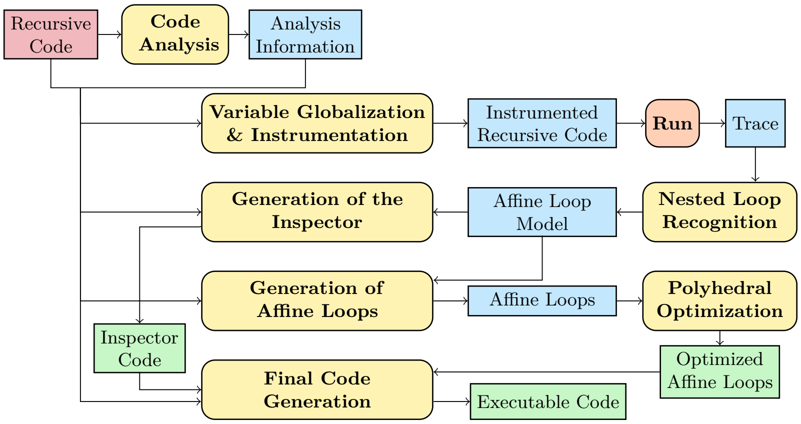


Fig. 1: Rec2Poly

[1] http://www.llvm.org

An instrumented version of the target recursive code is then generated by using the so-collected information. In this code, an invocation counter is added to each impacting function, and each data structure which is local to an impacting function is transformed into a global data structure indexed using this counter (this is called Variable Globalization). Moreover, every impacting function is augmented with instructions for generating the output trace, which is composed of impacting basic block identifiers, invocation counter values and memory addresses referenced in the impacting basic blocks, through load and store instructions.

After having run the instrumented code, the generated code is given as input to the Nested Loop Recognition software tool NLR [7]. NLR generates a representation of the whole trace made of affine loops computing affine expressions. Then, the so-generated affine loop model is used to build a fast parallel inspector code and a code made of optimized and parallelized loops, which is dedicated to replace the impacting functions in the original recursive code. Since the replacing loops are based on the modeling of one execution of the target recursive code, their validity for further executions must be ensured at runtime. This is achieved by the inspector code whose role is to verify that the original recursive code still behaves in compliance with the replacing loops.

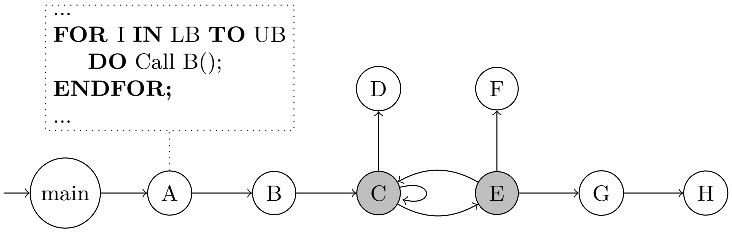
Since the replacing loops are made of affine loops, they can benefit from polyhedral optimization and parallelization transformations. For this purpose, we use the state-of-the-art polyhedral compiler Pluto [3]. Finally, the executable code is made of the inspector code and the transformed code made of optimized loops. More details on the main phases of Rec2Poly are given in the following subsections.

3 Code Analysis

First of all, Rec2Poly, our LLVM-Clang based tool, takes as an input a target recursive source code and transforms it into its intermediate representation (IR) that will be analyzed and transformed in the following steps. We do not activate the tail call elimination LLVM pass which transforms tail recursive calls into loops for two reasons: (1) the way the target recursive function is transformed may not result in an affine loop and (2) if there are several nested recursive calls in the target code, only one tail call may be eliminated.

Rec2Poly checks if the program involves any recursions, and, if so, it identifies these recursions and the functions participating in them. In order to detect recursions, it uses the call graph extracted from the LLVM IR of the program. Figure 2 shows an example of a call graph of a program made up of nine functions: *main*, A, B, C, D, E, F, G and H where: function *main* calls A which calls B which calls C; C invokes itself, E and D; E calls back C, and calls F and G; G calls H. In this example, note that function C exhibits a direct recursion with itself and an indirect recursion through function E.

From the call graph, Rec2Poly seeks strongly connected components (SCC), which are sub-graphs where every node is reachable from every other node. In



```
...
FOR I IN LB TO UB
    DO Call B();
ENDFOR;
...
```

Fig. 2: Example of a Call Graph of an Arbitrary Recursive Program

this context, a cycle in a SCC means that a recursion occurs among the functions associated to the nodes involved in this cycle. If the cycle is made up of only one node, *i.e.* a loop, then it is a direct recursion. Otherwise, it is indirect. For the example in Figure 2, there are two SCC's: one loop over C showing a direct recursion, and a cycle from C to E and E back to C, showing an indirect recursion between C and E.

Recursion Reachability Recognition: We are interested in tracking impacting basic blocks, whether they are executed directly or indirectly by the recursive functions. For this reason, in addition to the recursive functions themselves, our framework also determines their reachability in the program. Reachability means all the functions that can be reached by a sequence of calls initiated by the recursive functions themselves. In Figure 2, the reachability of the recursive functions C and E includes: D (directly called by C), F and G (directly called by E) and H (indirectly called by E through G).

Recursion Source Recognition: Not only do we track functions constituting a recursion and their reachabilities in a program, but also the source of recursions. The initial source function may be a function invoking, from within a loop, a recursive function, either directly or indirectly, through a chain of invoked functions. In addition, in case of indirect invocation of the recursion from a loop, all other functions participating in the sequence of calls from the initial source function to the recursion are considered as taking part of this source.

Analyzing source functions is necessary since otherwise, the profiling phase would be incomplete. A looping behavior detected afterwards and associated to the recursion itself would be incorrect when the recursion is invoked from within a loop. Furthermore, it helps in understanding how a recursion behaves relatively to its iterative invocations, and obviously to build affine loops which are equivalent to this whole part of the original program.

For instance, the set of source functions of the recursion in Figure 2 includes A and B because A calls B from the body of a for-loop, and B, in turn, calls the recursive function C. The loop in A cannot benefit from efficient loop polyhedral optimizations, due to the existence of the embedded recursive call. If it is possible to replace it with equivalent affine loops, the loop may eventually be able to take advantage of sophisticated optimizations.

Impacting Basic Blocks: Such basic blocks are identified in the following way. In the LLVM intermediate representation of the program, our framework marks the stores to the *main data structure*. The *main data structure* is defined as being the final output data structure of the recursion and its corresponding reachable functions. Then, for every store instruction of this kind, it also marks every instruction leading and contributing to it, i.e., its *backward static slice* (BSS). A backward static slice is the set of instructions existing in the code of a program that may affect a certain value, *i.e.*, in our case, a value stored in the data structure which is the final output of the recursion.

Intra-Function and Inter-Function Memory Behavior Analysis: Additionally to the identification of a looping behavior of the targeted recursion, the sequence of memory addresses touched by each memory instruction, inside the impacting basic blocks, must be successfully modeled by affine functions of the surrounding loop indices.

However, among different execution instances of the same target recursive code, with exactly the same input data and the same hardware platform, the touched memory addresses are obviously not the same, since data structures are not always allocated at the same memory addresses. Nevertheless, the memory behavior relatively to the base addresses of the data structures may still be identical among the execution instances. Moreover, we are interested in cases where the relative memory behavior can be modeled by affine functions of surrounding loop indices. Thus, the memory offsets that are relative to the base addresses are collected from instrumentation.

When handling data structures which are local to functions, the analysis phase requires two steps:

- Intra-Function Analysis: Each memory access is associated to its corresponding base address visible in the scope of the current function i.e., the parameters of the function are the farthest analysis point.
- Inter-Function Analysis: If the accessed data structures are function parameters, intra-function analysis is not enough. Memory analysis propagates further outside the function to trace arguments fed to the function. Inter-function analysis associates each access to its actual base address in the program.

On the other hand, when handling global data structures, the accessed memory addresses can be directly associated to their base addresses.

4 Local Variable Globalization and Code Instrumentation

The first goal of Rec2Poly is to detect an affine behavior of the impacting functions, regarding their control flow, and also their memory accesses: for each memory instruction, the sequence of touched memory addresses must potentially be represented as affine expressions of surrounding loop indices. However, at each invocation of an impacting function , its local data structures are obviously allocated on the call stack. Thus, accesses to these local structures can never exhibit any affine memory accesses across all invocations of the function. Moreover, in the affine loops that are expected to replace the impacting functions, these data structures must obviously still be referenced. This is why in the instrumented code, all data structures that are local to an impacting function are transformed into global arrays, which are indexed by the function invocation counter. In this way, references to these globalized data may exhibit affine behaviors whether the related functions are invoked following an affine control flow.

5 Nested Loop Recognition

Rec2Poly instruments the target recursive program in order to generate an execution trace. This trace is made of tuples composed by:

- the impacting basic block ID;
- for each memory instruction in the current basic block: the relative offset of the touched memory address.

After having been generated by running the instrumented recursive program, the trace is analyzed by NLR. The NLR software tool, originally presented in [7], takes as input a trace of a program execution and constructs a sequence of loop nests that produce the same original trace when run. The applications of this algorithm include: (1) program behavior modeling for any measured quantity such as memory accesses, (2) execution trace compressing and (3) value prediction, i.e, extrapolating loops under construction (while reading input) to predict incoming values.

In our tool, not only do we use NLR to model memory accesses, which is one of its original goals, but also to model sequences of basic blocks IDs, which is more singular. Given our trace of a target recursive program, if NLR builds affine loop nests including the interesting basic blocks IDs and memory addresses interpolated by the constructed loop indices, then the generation of equivalent affine loops may be performed.

Two examples of NLR outputs are shown in Figures 3a and 3b. The generated loops exhibit the way basic blocks (`BB1`, `BB2`, `BB3`, `BB4`) inside functions (`F1`, `F2`) are invoked by following an affine looping behavior, and how memory is referenced through relative addresses that can be modeled as affine functions of the loop indices.

Note that in Figure 3b, NLR uses one of its more advanced features which detects that the affine modeling of a trace may be subject to some basic unknown values. NLR discovers these values and exhibits a memory behavior which is actually not fully affine: some coefficients in the affine functions may be lists of values. In the showed example, each list contains `10` integer values which are successively used to compute the referenced memory address. For example,

```
for i0 = 0 to 99
 val F1::BB1
 for i1 = 0 to 49
   val F2::BB1
   , 0
   for i2 = 0 to 24
     val F2::BB2
     , 1*i0
     , 4*i0 + 2*i1 + 1*i2
     , 4*i0 + 1*i1 + 1*i2
   val F2:BB3
 val F2::BB4
```

(a) NLR Model for Linear Control and Memory Behavior

```
for i0 = 0 to 99
 val F1::BB1
 for i1 = 0 to 49
   val F2::BB1
   , 0
   for i2 = 0 to 24
     val F2::BB2
     , 1*i0
     , [10:3,5,...,1][i0] + 2*i1 + 1*i2
     , [10:7,1,...,6][i0] + 1*i1 + 1*i2
   val F2:BB3
 val F2::BB4
```

(b) NLR Model for Linear Control and Non-Linear Memory Behavior

Fig. 3: NLR models

`[10:3,5,...,1][i0]` means that:

$$[10:3,5,...,1][i_0] = \begin{cases} 3 \ \text{ if } i_0 \ \textit{modulo} \ 10 = 0 \\ 5 \ \text{ if } i_0 \ \textit{modulo} \ 10 = 1 \\ ... \\ 1 \ \text{ if } i_0 \ \textit{modulo} \ 10 = 9 \end{cases}$$

6 Generation of a Fast Parallel Inspector

The affine loop model generated by NLR is then used by Rec2Poly to generate the inspector. Its role will be to verify that the optimized and parallelized affine loops, which replace the recursive program, are still correct in the current execution context. It is made of three main kinds of components:

1. *Trace generators*, which are minimal versions of the original recursive program, devoted to producing the same kind of execution traces as the one which was generated at the profiling phase, *i.e.*, tuples made of functions and basic blocks IDs, and referenced memory addresses;
2. *Verifiers*, whose role is to check if the generated traces are still compliant with the NLR affine loop model;
3. *A parameter saver*, whose role is to collect function input values which are used by instructions of the impacting basic blocks.

We illustrate their functionality and how Rec2Poly modifies the IR of a given recursive code to build its suitable inspector.

Trace Generators: A trace generator is made up of light minimal clones of the impacting functions, *i.e.*, source, recursive and reachable functions. Its role is to generate a trace representing the actual control flow or the sequence of touched memory addresses.

After cloning impacting functions and their basic blocks, Rec2Poly removes instructions that involve access to memory such as stores and loads. Instructions that are fundamental to preserve a correct control behavior of these functions must be preserved such as branches, conditions, loop related instructions and calls. We assume for this study that conditional branches do not depend on any memory access. Moreover, in the clones, the referenced impacting functions in call instructions must be replaced by their proper clones.

A trace generator is expected to output a trace so the latter can be verified against the NLR affine loop model. For this sake, global memory buffers or arrays are added to the IR.

The Inspector must be significantly slower than the original recursive program, such that the final couple Inspector-Executor provides significant speedups. Generating one complete trace of full tuples of values, similar to the trace generated at the profiling phase, would be too costly. Thus, in order to guarantee a fast trace generation process, an inspector, created by Rec2Poly, is composed of multiple trace generators, *i.e.*, multiple clones of impacting functions, that are executed in parallel each by a distinct parallel thread. Each of these generators is responsible for generating one sub-part of the trace, *e.g.*, one generates the whole control flow IDs, and the others generate sequences of touched memory addresses. Accordingly, Rec2poly is expected to tackle a load balancing issue among threads by deciding how many and which memory accesses a single trace generator must handle.

Verifiers: For every trace generator, Rec2Poly creates a corresponding trace verifier based on the NLR affine loop model. Each verifier is generated as a new function that implements the NLR loops and minimal versions of their enclosed basic blocks. At runtime, the trace verifiers are also launched in parallel threads.

Parameter Saver: Some input arguments of impacting functions may be values transmitted by the calling function and used directly by instructions. Such parameters have to be collected specifically in order to instantiate the replacing loops. Like trace generators, a parameter saver is made up of a minimal light version of the code part involving impacting functions. It saves function input values in a global buffer array at the entry of every impacting function. Parameter savers are also executed simultaneously with trace generators and verifiers.

Further Inspector Optimizations: In some cases, the inspector does not need to handle every memory access inside an impacting basic block. For instance, in cases of array accesses, if the same array is accessed several times through indices computed using the same induction variable, then only one of these accesses is worth being handled and verified by the inspector.

Figure 4 shows the call graph of an example of inspector based on the arbitrary example of recursive program of Figure 2. It shows that the main function launches $M + 2$ parallel threads. There are one parameter saver and $M/2$ trace generators, which require $M/2+1$ minimal light clones of the impacting functions which are the source functions A and B; the recursive functions C and E; and the reachable functions D, F, G and H. For every thread that initiates a trace generator, there is a thread initiating a verifier function. Each trace generator and its verifier have their own set of buffers to process on, and they synchronize using two semaphores. For instance, Thread 1 and Thread 2 synchronize using Semaphores sem 0 and sem 1. Each trace generator saves its traces in its dedicated $N + 1$ buffers to be verified by its corresponding verifier.

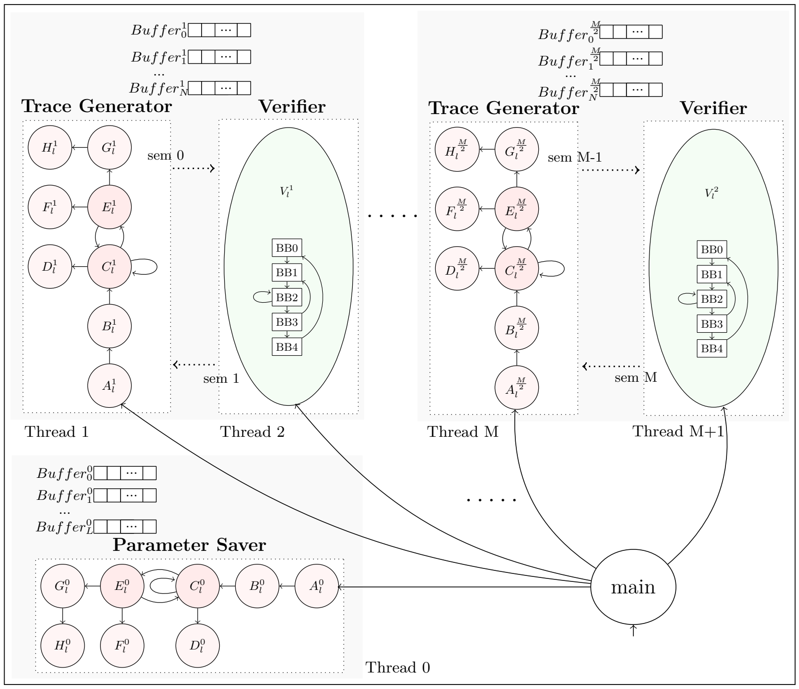


Fig. 4: Detailed Inspector Call Graph Example

7 Generation of Optimized Affine Loops as Executor

This last phase of code generation takes as input the recursive program after variable globalization, and the corresponding NLR affine loop model obtained at the profiling phase. Rec2Poly builds the replacing affine loop program by: (1) cloning the impacting basic blocks; (2) replacing the referenced memory addresses by the corresponding NLR affine functions added to the related base addresses collected at runtime; and (3) replacing the use of function input values by the values collected at runtime by the Inspector. Finally, the function called in the original program to initiate the recursion is replaced by the newly created function made up of the iterative code constructed from NLR affine loop model.

As mentioned in Section 5, NLR may produce two types of loop models: (1) affine control and memory behavior and (2) affine control and non-linear memory behavior. We use different approaches for optimizing each of these types of affine loops, as it is explained below.

Loops with Affine Control and Memory Behavior: This is the most favorable case with pure affine loops which are ready to be optimized using an automatic polyhedral optimizer as Pluto. However, since we generate and transform code in LLVM Intermediate Representation, we need to feed Pluto with an OpenScop representation [2] of the affine loops.

Loops with Affine Control and Non-Linear Memory Behavior: In this case, polyhedral automatic optimizers cannot be used. However, efficient loop parallelization can be achieved that requires a dedicated dependency analysis process. It consists of computing the ranges of touched memory addresses by store and load instructions at each iteration, in order to build independent sets of iterations. Finally, the outer loop is broken into two nested loops: the outer one iterating over lists of loop indices values and the inner one over the indices values inside each list. The outer one is parallelized into parallel threads.

8 Experiments

The following programs have been compiled with Clang version 6.0 and flags `-O3 -march=native`, and run on two Intel Xeon CPU E5-2650 v3 @ 2.30GHz of 10 cores each. Parallel programs have been run using 20 threads on the 20 cores of the hardware platform.

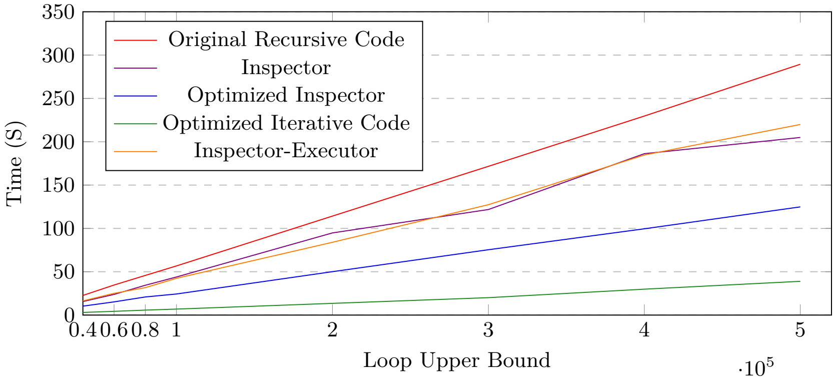


Fig. 5: Program Heat Experimental Results

Our first experiment has been conducted on program Heat which is a recursive C implementation of a stencil computation. It involves a direct recursion invoked from within a loop. Its reachable functions also include for-loops accessing memory numerous times. In such an example, the recursion distorts the existence of loops, and it is interesting to test how much time performance can be gained by removing the recursion and applying polyhedral optimizations to the recursion-free loop nest. Both control and memory behavior are linear. Hence, an affine loop code can be reconstructed automatically from the NLR affine loop model. The affine iterative code equivalent to this recursive program is optimized using Pluto. Two versions of the inspector have been experimented here: (1) the original inspector and (2) the optimized inspector. In the original inspector, we verify all memory addresses referenced by impacting functions besides the control. Thus, 10 trace generators and 10 verifiers are launched. On the other hand, in the optimized version, half of the time overhead is eliminated by not verifying redundant memory accesses, thus only 5 trace generators and 5 verifiers are generated. Figure 5 shows the obtained execution times (vertical axis), relatively to an increasing value of the upper bound of the loop invoking the recursion (horizontal axis). The inspector version of the code, given that the loop bound equals 500000, has about 29% better performance than the original recursive code that can be optimized more deeply to perform 56.8% faster. The equivalent optimized iterative code executes about 86.5% faster than the original code. Overall, by executing both of the optimized inspector and executor together instead of the recursive code, the gain is about 24%.

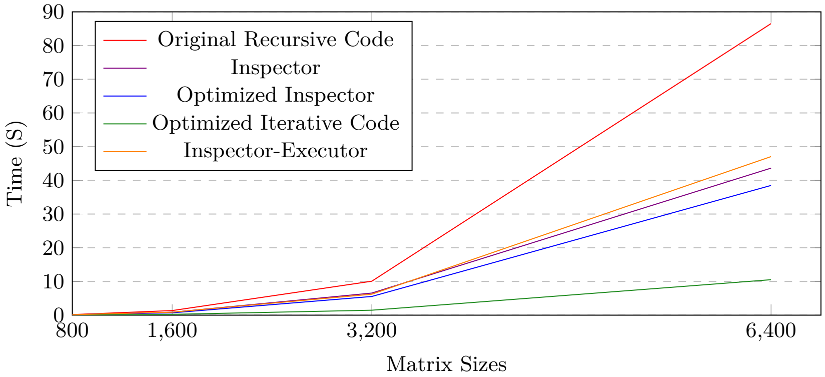


Fig. 6: Program GEMM Experimental Results

Our second experiment was conducted on a C implementation of the recursive matrix-matrix multiplication (GEMM) handling sub-matrices by successive dichotomy until a given threshold. It involves an indirect recursion among four functions. There is one reachable function from this recursion that includes a sequence of affine loops accessing memory. The recursion in this program has a linear control, but a non-linear memory behavior. Thus, the iterative code handles independent lists of iterations that are executed in parallel. Note that the original inspector is composed of 7 trace generators and 7 verifiers where the optimized version includes only 4 trace generators in addition to 4 verifiers. Figure 6 shows the obtained execution times. For matrix size 6400 × 6400, the inspector version executes about 49.5% faster than the original recursive code while the optimized inspector executes 55.5% faster. The equivalent optimized iterative code executes about 87.8% faster than the original code. Overall, by executing both the optimized inspector and executor together instead of the recursive code, the gain is about 45.6%.

9 Related Work

Our study has been inspired by the automatic speculative polyhedral loop optimizer APOLLO [13, 9]. APOLLO applies automatic, speculative and dynamic loop optimizing and parallelizing transformations. It addresses loop nests that do not have an affine structure, yet adopt an affine memory behavior at run-time discovered by dynamic profiling. Also, APOLLO's verification of speculative loop transformations is partially based on the inspector-executor paradigm. In comparison, Rec2Poly goes further by handling recursive codes and making use of a profiling technique not only to discover the memory behavior, but also the control behavior. Finally, Rec2Poly applies the Inspector-Executor paradigm to verify recursions against a loop model, which requires the verification of both control and memory behaviors, while APOLLO uses this paradigm only to verify the memory behavior of loop nests against the one of a predictive loop model.

Multiple works have been published to transform recursive codes and optimize them. Nevertheless, the proposed approaches are mainly static and involve task parallelization where several recursive calls are run simultaneously.

PolyRec [14] optimizes nested recursive programs by polyhedral scheduling transformations. To allow such optimizations, PolyRec represents recursive function instances and their dependences as polyhedra, and applies scheduling transformations. Nevertheless, their approach is exclusively committed to particular forms of recursions such that recursive invocations are nested and data is organized in two trees, the inner and outer trees.

Gupta *et al.* [6] propose an approach to optimize recursive task parallel programs through lessening task creation and termination overhead. Adriadne [10] is a compiler that retrieves, from recursive functions, directive-based parallelism. It applies either: (1) recursion elimination, (2) parallel-reduction removing recursion such that workload is distributed to independent tasks, or (3) thread-safe recursive functions parallelization involving independent recursive calls. Adriadne is solely devoted to recursive functions whose parameters remain unchanged among recursive calls except for one integer parameter.

10 Conclusion

To our knowledge, Rec2Poly is the first attempt of speculative program optimization involving the rewriting of the target code. We have shown that using such an approach, some recursive programs may take advantage of efficient affine loops optimizations, and even take advantage of advanced transformations of the polyhedral model.

However, while the inspector-executor mechanism is adapted to such speculative optimizations, the final performance is mostly relying on the performance of the inspector. We have shown that the inspector must also be deeply optimized and parallelized to lower its time overhead. In the near future, we will still investigate strategies to lower even further the inspector time-overhead.

References

1. Arsac, J., Kodratoff, Y.: Some techniques for recursion removal from recursive functions. ACM Trans. Program. Lang. Syst. **4**(2), 295–322 (Apr 1982)
2. Bastoul, C.: Openscop: A specification and a library for data exchange in polyhedral compilation tools. Tech. rep., University of Paris-Sud, France (Sept 2011)
3. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI '08. pp. 101–113. ACM (2008)
4. Feautrier, P., Lengauer, C.: Polyhedron model. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 1581–1592. Springer US (2011)
5. Gupta, M., Mukhopadhyay, S., Sinha, N.: Automatic parallelization of recursive procedures. International Journal of Parallel Programming **28**(6), 537–562 (Dec 2000)
6. Gupta, S., Shrivastava, R., Nandivada, V.K.: Optimizing recursive task parallel programs. In: Proceedings of the International Conference on Supercomputing. pp. 11:1–11:11. ICS '17, ACM, New York, NY, USA (2017)
7. Ketterlin, A., Clauss, P.: Prediction and trace compression of data access addresses through nested loop recognition. In: Proceedings of the 6th IEEE/ACM International Symposium on Code Generation and Optimization. pp. 94–103. CGO'08, ACM, New York, NY, USA (2008)
8. Kobeissi, S., Clauss, P.: The Polyhedral Model Beyond Loops Recursion Optimization and Parallelization Through Polyhedral Modeling. In: IMPACT 2019 - 9th International Workshop on Polyhedral Compilation Techniques, In conjunction with HiPEAC 2019. Valencia, Spain (Jan 2019)
9. Martinez Caamano, J.M., Selva, M., Clauss, P., Baloian, A., Wolff, W.: Full runtime polyhedral optimizing loop transformations with the generation, instantiation, and scheduling of code-bones. Concurrency and Computation: Practice and Experience **29**(15) (Jun 2017)
10. Mastoras, A., Manis, G.: Ariadne - directive-based parallelism extraction from recursive functions. J. Parallel Distrib. Comput. **86**(C), 16–28 (Dec 2015)
11. Rauchwerger, L., Padua, D.A.: The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. IEEE Transactions on Parallel and Distributed Systems **10**(2), 160–180 (1999)
12. Saltz, J.H., Mirchandaney, R., Crowley, K.: Run-time parallelization and scheduling of loops. IEEE Transactions on Computers **40**(5), 603–612 (1991)
13. Sukumaran-Rajam, A., Clauss, P.: The polyhedral model of nonlinear loops. ACM Trans. Archit. Code Optim. **12**(4), 48:1–48:27 (Dec 2015)
14. Sundararajah, K., Kulkarni, M.: Scheduling transformation and dependence tests for recursive programs (2018)