Is Source-code Isolation Viable for Performance Characterization?

Chadi Akel, Yuriy Kashnikov, Pablo de Oliveira Castro, William Jalby
Université de Versailles Saint-Quentin-en-Yvelines, France
Exascale Computing Research, France

***Abstract*—Source-code isolation finds and extracts the hotspots of an application as independent isolated fragments of code, called *codelets*. Codelets can be modified, compiled, run, and measured independently from the original application. Source-code isolation reduces benchmarking cost and allows piece-wise optimization of an application. Source-code isolation is faster than whole-program benchmarking and optimization since the user can concentrate only on the bottlenecks.**

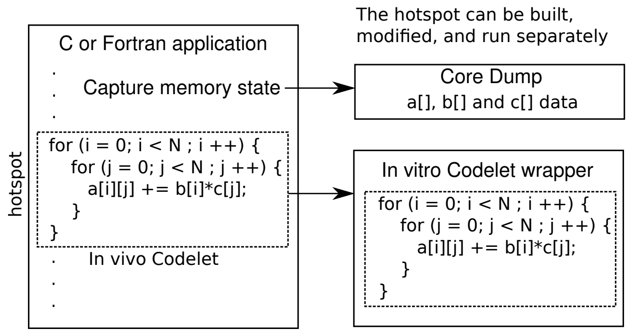
This paper examines the viability of using isolated codelets in place of the original application for performance characterization and optimization. On the NAS benchmarks, we show that codelets capture 92.3% of the original execution time. We present a set of techniques for keeping codelets as faithful as possible to the original hotspots: 63.6% of the codelets have the same assembly as the original hotspots and 81.6% of the codelets have the same run time performance as the original hotspots.

I. Introduction

Benchmarking scientific applications is a costly, but necessary, process to validate software and hardware optimizations. Usually, in scientific applications the performance critical sections, or *hotspots*, represent a small fraction of the total source lines [1]. Lee and Hall [2], Petit and Bodin [3], and Liao et al. [4] propose to outline and isolate the hotspots from the rest of the application to simplify benchmarking and performance tuning. The source code is analyzed and hotspots outlined; then isolated versions of the hotspots, called *codelets*, are produced. Codelets can be compiled and run independently from the original application. Breaking an application into independent codelets provides multiple benefits:

- Benchmarking isolated hotspots instead of whole applications is attractive because the user can concentrate on each hotspot separately, with a reduced build and run cost.
- Codelets can be individually modified to evaluate the payoff of new optimizations.
- Different codelets may expose different performance bottlenecks, and react differently to optimizations. Isolating codelets can be used to tune performance at a fine-grain level.

Nevertheless, this benchmarking process is only viable if the codelets faithfully capture the original application performance behavior. Yet, Lee and Hall observed a "*fairly significant variation in the performance monitoring results returned by PAPI [between the original and isolated code].*" In this paper, we study the viability of source-code isolation for performance characterization through three important questions that have not been addressed by previous work:



```
for (i = 0; i < N ; i ++) {
   for (j = 0; j < N ; j ++) {
      a[i][j] += b[i]*c[j];
   }
}
```

Fig. 1: Codelet extraction process.

- *Coverage*: What percentage of an application running time can be captured with code isolation?
- *Assembly fidelity*: How close is the assembly code between the isolated and original versions?
- *Runtime fidelity*: How close is the runtime performance between the isolated and original version?

Our study uses the Codelet Finder tool developed by CAPS enterprise [5]. Figure 1 schematizes the codelet extraction process of Codelet Finder. Codelet Finder isolates hotspots in `C` and `Fortran` applications, and extracts them as separate source files. It also captures the memory state in the initial application to allow replaying the codelet with the same dataset. In the rest of the paper we will use the following terminology: a hotspot inside of the original application, is called **in vivo** codelet. An isolated hotspot, compiled and run independently from the original application, is called **in vitro** codelet.

We evaluate codelets coverage and fidelity for the NAS 3.0 serial benchmarks [6] using dataset sizes in class B. All our experiments were done on an **Intel(R) Xeon(R) CPU L5609 @ 1.87GHz with 8GB of RAM and 12MB L3 cache**.

II. Codelet Finder Overview

Codelet Finder [5] deconstructs an application, by identifying the hotspots and extracting them as standalone *codelets*. Codelet Finder is an extension of the ASTEX hot path extractor [3] and is similar to Code Isolator [2]. Figure 2 presents the workflow of codelet finder.

First, Codelet Finder profiles the target application using the GNU profiler to detect the hotspots. The user does not need to tweak the build process: Codelet Finder automatically instruments and profiles the application. Second, Codelet

Finder analyzes each hotspot and marks all loops it contains. Codelet Finder extracts each loop as a separate codelet. The tool generates for each codelet a `Makefile` and a wrapper to build and execute the hotspot as a standalone program. Finally, Codelet Finder runs the original application and captures the memory locations accessed by each extracted loop. The memory state is saved into a memory dump file. The codelet wrapper loads this file before a codelet is run to restore the original execution environment.

Code Isolator [2] and Codelet Finder differ in the method for capturing the execution environment. Codelet Isolator analyzes the static data flow of the original application to determine exactly which data structures need to be captured. This method produces small dumps because only the needed data is captured, but cannot deal with pointer aliasing. In contrast, Codelet Finder extracts a full memory dump of the memory of the original application. A full memory dump is much larger but handles the pointer aliasing problem since the full memory is recorded. It also preserves the relative alignment and offsets between data structures.

Codelet Finder saves the original memory state to a file and restores it before running a codelet in vitro, to preserve the in vivo execution environment. Loading the memory state from a file has the advantage of allowing the user to modify the file to try the effect of different datasets. Yet, the dataset loading mechanism of Codelet Finder has a huge overhead, since the dataset has to be parsed and copied to memory.

Since we propose to use codelets to accelerate benchmarking and optimization, it is important that the overhead of running a hotspot *in vitro* is negligible. Therefore, we modified the default memory restoration mechanism of Codelet Finder. Instead of loading a dataset from a file at run time, we directly patch the dataset into the BSS section of the codelet ELF binary during the link phase. Therefore, the dataset is directly mapped into memory by the Operating System when the binary is loaded. By statically including the dataset in the executable, the memory restoration overhead becomes almost negligible.

III. Motivating example: faster benchmarking with Codelet Finder

In this section we show on a simple example, the benefits of using code isolation for accelerating benchmarks. We consider a hotspot from NAS SP (Scalar Pentadiagonal solver) extracted from `y_solve.f` file between lines 27 and 287. This region of code is executed 401 times in the original application and represents more than 14.5% of the application execution time.

Codelet Finder isolates this hotspot and captures the original dataset memory dump. The codelet wrapper executes a first run of the hotspot to warm up the cache (details are further described in section V-B). Then, we benchmark the hotspot: we measure Cycles Per Instructions (CPI) on ten consecutive calls of the hotspot to improve accuracy.

In the original application, we measured 0.58 CPI, whereas in the isolated codelet we measured 0.60 CPI. The measurement error is therefore 4.4%. Benchmarking the whole original

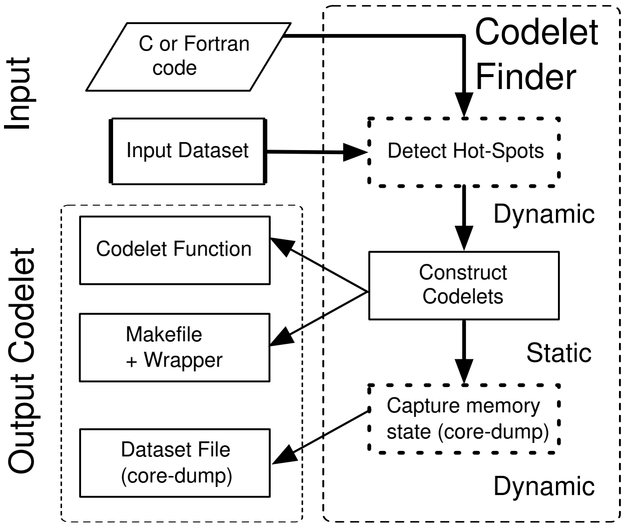


Fig. 2: Codelet Finder Workflow.

application took 215.32 seconds including the 401 calls to the hotspot that took 31.17 seconds. In contrast, benchmarking the in vitro codelet took 0.98 seconds because (1) only the `y_solve` codelet was executed and (2) the hotspot was invoked 10 times instead of 401 times. This represents a $\times 219$ speedup in benchmarking time.

The 0.98 seconds can be decomposed in three subparts:

- 0.03 seconds were spend to launch the codelet wrapper, they include the OS loading the binary to memory.
- 0.12 seconds were spend on the initial warm-up codelet execution
- 0.83 seconds were spend on the benchmarked ten codelet invocations

The warm-up execution is more costly because the accessed data is not yet in cache.

This example demonstrates the huge speedup that can be achieved when benchmarking isolated code. This method is particularly interesting when the user is only interested in measuring or optimizing the performance of a small fraction of the total source code.

For this particular hotspot, the CPI measured in vitro was accurate (4.4%) because the hotspot has the same behavior in vivo and in vitro: assembly and runtime are close. Clearly to use in vitro benchmarking we need to guarantee that:

- the codelet can be extracted
- the codelet has the same behavior in vivo and in vitro

In the following sections, we investigate if and when these two conditions are satisfied for the NAS benchmarks.

IV. Coverage

To efficiently use codelets for benchmarking we must ensure that the set of codelets extracted cover most of the application's original execution time.

The out-of-the-box version of Codelet Finder cannot extract codelets that call functions defined in another source file. This limitation was severely decreasing the coverage of Codelet

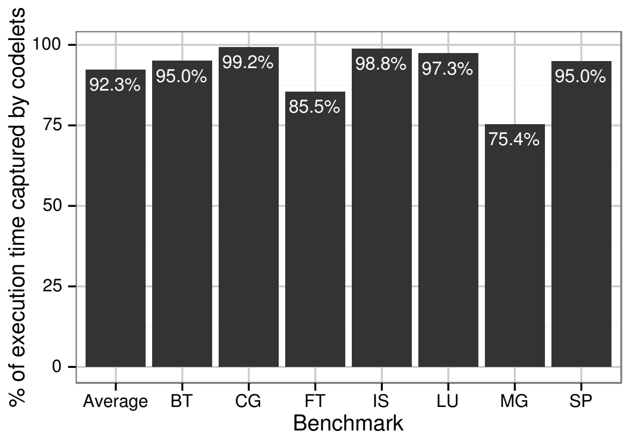


Fig. 3: Execution time coverage of the extracted codelets in NAS benchmarks.

Finder, therefore we extended Codelet Finder to support extraction of codelets calling functions in other files. The results that follow include our extension.

Figure 3 shows for each NAS benchmark the percentage of execution time captured by codelets. On average, the extracted codelets cover 92.3% of the execution time. MG has the lowest coverage, at 75.4%. The missing 24.6% of the time is spent outside of the hotspots extracted by Codelet Finder. There are two reasons for this: (1) Codelet Finder only extracts hotspots delimited by a loop, execution time spent outside of loops cannot be captured. (2) The current version of Codelet Finder does not extract hotspots that call external libraries or manipulate file descriptors, since it cannot guarantee that the extracted codelet preserves the runtime state of the original application.

Despite these limitations, the coverage achieved by Codelet Finder is over 75% for all the NAS benchmarks. This coverage ratio seems sufficient to use codelets in place of the original application for most performance studies.

V. Discrepancies between the in vivo and in vitro codelets

Is the behavior of a codelet identical between the in vivo and in vitro versions? If the answer is positive, then a codelet can be used in place of the original hotspot for benchmarking and code optimization. We approach this fundamental question by analyzing the causes of discrepancies during code isolation, providing mitigation techniques when possible, and quantifying the similarity between in vivo and in vitro versions on the NAS benchmarks. We separate discrepancies in two categories assembly discrepancies and runtime discrepancies.

A. Assembly discrepancies

Code isolation tools extract codelets at the source level. This allows to test multiple compilers or source code transformations but has a potential drawback: the assembly code compiled for the codelet (in vitro) may differ from the assembly code compiled for the original hotspot in the application (in vivo). Below we study three cases of assembly discrepancies and how to mitigate them.

1) Dereferencing: In their Code Isolator paper [2] Lee and Hall give an example of a codelet extracted from the nonlinear finite element application LS-DYNA. In figure 4 we reproduce Lee and Hall's example.

We observe that the variables in the in vitro code of figure 4 have been dereferenced. By passing the variables by reference, it is easy to preserve the values modifications during the codelet execution. This is a classic technique in code outliers [4] which has the unfortunate side-effect of disabling many compiler optimizations. The Intel Compiler 12.1.0 `-O3` optimization level produces shorter and more efficient assembly for the in vivo version than for the in vitro version. The isolated code is also slower. With the original dataset sizes used in [2], the isolated code is 1.26 times slower (median of 15 executions).

To mitigate the problem with variable dereferencing the variable cloning algorithm proposed by Liao et al. [4] is effective. Variable cloning copies the pointer variables to fresh local variables in the scope of the codelet.

Codelet Finder does not use the dereferencing technique during inlining. Variables are passed directly by value as parameters of the codelet wrapper. Therefore, Codelet Finder produces matching in vivo and in vitro versions for the example in figure 4.

2) Interference with Loop Variables: Codelet Finder is subject to other problems, such as interference with loop variables. The codelet in figure 5 is extracted with Codelet Finder from the NAS SP benchmark. Because the codelet does not use the `rhs` array after the computation, the compiler is free to fully remove the codelet. Codelet Finder inserts the `cf_anti_dead_code` calls to prevent the compiler from applying dead code elimination to the whole loop; `cf_anti_dead_code` is defined as an empty function.

Nevertheless, this causes further problems. Because of the `cf_anti_dead_code_(k)` call, the compiler infers that the loop variable `k` is live after the loop. This prevents the compiler heuristics from fully unrolling the innermost loop in the in vitro codelet, which it does in the in vivo codelet. By introducing anti dead code calls on loop variables, Codelet Finder hampers some compiler transformations, such as unroll. Also loop variables and loop bound variables (`nz2`) are declared as function parameters of the codelet wrapper. We have observed that, in some cases, the compiler also does not apply some optimizations for variables declared as function parameters.

To fix these problem:

- we remove `cf_anti_dead_code` calls for loop variables which are not live-in after the loop
- we apply variable cloning [4] to the loop variables and loop bound variables, so they are declared as local variables instead of function parameters

We'll refer to these optimizations as the **loop-variable-fix**. Figure 6 shows the same SP codelet after applying these modifications.

```fortran
do 20 j=1, size
  do 10 i=1, l_len
    l(lp+1) = matrix(pl+i)
10 continue
  lp = lp + 64
  pl = pl + sl
  sl = sl - 1
20 continue
```

(a) In vivo (original) codelet

```c
void codelet(int *size, int *l_len,
       float (*matrix)[], float (*l)[],
       int *lp, int *pl, int *sl) {
    int i, j;
    for (j = 1; j < *size; j++) {
        for (i = 1; i < *l_len; i++)
            (*l)[*lp+i] = (*matrix)[*pl+i];
        *lp = *lp + 64;
        *pl = *pl + *sl;
        *sl = *sl - 1;
}}
```

(b) In vitro (isolated) codelet

Fig. 4: In vivo and in vitro codelets extracted with Code Isolator [2] from LS-DYNA application.

```fortran
subroutine codelet(rhs,i,j,k,m,
                   nz2,ny2,nz2,dt)
 INTEGER :: i, j, k, m, nz2, ny2, nz2, dt
 do k=1, nz2
   do j=1, ny2
     do i=1, nx2
        do m=1, 5
          rhs(m, i, j, k) =
             rhs(m, i, j, k) * dt
 end do; end do; end do; end do
 call cf_anti_dead_code_(rhs)
 call cf_anti_dead_code_(k)
!   ...
```

Fig. 5: Codelet Finder codelet extracted from NAS SP (Scalar Pentadiagonal solver) in the file `rhs.f` between lines 387 and 395.

```fortran
subroutine codelet(rhs,i,j,k,m,
                   nz2,ny2,nz2,dt)
 INTEGER :: i, j, k, m, nz2, ny2, nz2, dt

! variable cloning for variables involved
! in loop indexes or bounds
 INTEGER :: i_cl, j_cl, k_cl, m_cl
 INTEGER :: nz2_cl, ny2_cl, nx2_cl
 i_cl = i ; j_cl = j ; k_cl = k; m_cl = m
 nz2_cl = nz2 ; ny2_cl = ny2; nx2_cl = nx2

 do k_cl=1, nz2_cl
   do j_cl=1, ny2_cl
     do i_cl=1, nx2_cl
       do m_cl=1, 5
         rhs(m_cl, i_cl, j_cl, k_cl) =
            rhs(m_cl, i_cl, j_cl, k_cl) * dt
 end do; end do; end do; end do
 call cf_anti_dead_code_(rhs)
!   ...
```

Fig. 6: NAS SP (Scalar Pentadiagonal solver) codelet with the **loop-variable-fix**. All variables involved in loop indexes or bounds are cloned.

3) Compiler Heuristics: Modern compilers use complex checks to measure the profitability and legality of an optimization before it is applied. When a codelets is extracted from the application, the code before and after the hotspot is not preserved. This may affect the optimizations that the compiler applies to the code, and therefore change the produced assembly.

Also, the compiler high level optimization passes often fuses or combine (unroll and jam) adjacent loops. If the two neighbor loops are not extracted together in the same codelet, the fusing transformation is no longer possible.

To mitigate these effects, we extract each hotspot with its maximal context. That is to say, we extract codelets from the outermost loop enclosing each hotspot. By preserving each hotspot source context, we reduce the interference with compiler heuristics and preserve complex transformations such as fusion or unroll-and-jam.

B. Runtime Discrepancies

Most of the time codelets that have the same assembly in vivo and in vitro have the same runtime behavior because both versions are executed with the same dataset. Nevertheless, the runtime behavior may be different if the state of the caches is different or due to memory alignment problems.

Codelet Finder captures the memory state of the original application just before entering a hotspot. The state of the caches is not captured. When executing codelets in vitro, we execute a first run to warm the caches and load the dataset in memory.

In cases where in the original application the data accessed was not in cache, warming up the cache in vitro may change the runtime behavior because the number of misses will change between the original application and the codelets.

Codelet Isolator [2] includes a method to restore the state of the L1 cache by recording the memory accesses preceding a codelet in the original application. Before running the isolated codelet, the same memory accesses are prefetched. This method preserves the L1 state between the in vivo and in vitro versions but has not yet been implemented in Codelet Finder.

MAQAO metric	Description
Unroll factor	Number of times the loop was unrolled by the compiler
Nb instr.	Number of instructions
Nb FLOP add-sub	Number of floating point additions or subtractions instructions
Nb FLOP mul	Number of floating point multiplication instructions
Nb FLOP div	Number of floating point division instructions
Bytes stored	Number of stored bytes per loop iteration
Bytes loaded	Number of loaded bytes per loop iteration
Arith. Intensity	Arithmetic intensity $FLOP/(bytesloaded + bytesstored)$
Nb cycles P[0..5]	Pressure on each dispatch port
Vec. ratio (%) INT	Percentage of integer vector instructions
Vec. ratio (%) FP	Percentage of floating vector instructions
Static IPC	Static prediction of the IPC assuming all memory access hit L1

TABLE I: MAQAO metrics used to quantify assembly discrepancy.

Codelets are also subject to alignment issues. Depending on the architecture, the performance of a code may be affected by false Store Forward stalls (4K aliasing), bank conflicts or misaligned access [7]. Because Codelet Finder captures the full memory state of the original application, the relative alignment between arrays and data structures is preserved preventing most of the alignment problems.

This section presented different types of discrepancies between the in vivo and in vitro versions, and introduced mitigation techniques such as the loop-variable-fix.

VI. Quantifying Discrepancies

We quantify the discrepancies between the in vivo and in vitro versions for codelets extracted from NAS benchmarks. For each codelet extracted we have tested for:

- **assembly discrepancies**, differences between the in vivo and in vitro assembly code
- **runtime discrepancies**, differences between the in vivo and in vitro runtime behavior

A. Methodology

All benchmarks and examples were compiled using Intel compiler `12.1.0` with the `-O3` flag. At this optimization level, the Intel compiler applies aggressive optimizations such as unroll-and-jam, inter procedural optimization or vectorization.

The analysis was performed only on the significant NAS hotspots. Therefore, we removed all codelets that capture less than 1% of the application runtime. After removal, 44 codelets remained over 226 codelets extracted.

The assembly analysis was conducted using the MAQAO static loop analyzer [8], [9] which provides detailed assembly characteristics. Directly comparing the assembly code of the two versions would report many false positives in discrepancies, such as identical assemblies that differ only in offsets relative to the program counter, or identical assemblies that differ only in the names of the registers they use. Therefore, instead of directly comparing the assembly code, we compare the set of assembly characteristics reported by the MAQAO static analyzer detailed in Table I. We consider that the in vivo and in vitro codelets match if the difference between those characteristics is under 15% for all of them.

The runtime discrepancy analysis uses two metrics: instructions retired and CPI. Comparing the instruction retired metric ensures that the in vivo and in vitro version execute the same number of assembly instructions. CPI is the mean number of cycles per instruction, this metric is used to check that the performance of the two version is similar. To measure these metrics we instrumented in vivo and in vitro codelets with Likwid 3.0 [10]. Likwid introduces a small overhead necessary to read performance counters. When instrumenting the codelets we wrapped the codelet invocation in a repetition loop. By measuring one hundred codelet executions we mitigate the instrumentation overhead. Again we consider that the in vivo and in vitro versions match if the difference in the above metrics is less than 15%.

B. Results

Figure 7 summarizes the assembly discrepancy between the in vivo and in vitro NAS codelets. Results are reported per benchmark, bars show the percentage of codelets whose assembly code matched between the in vivo and in vitro version. Dark bars represent the matching with the out of the box Codelet Finder version, Light bars represent the matching after applying our loop-variable-fix.

For the Conjugate Gradient (CG) benchmark, no single codelet matches between the in vivo and in vitro versions. Codelets inside CG are large outer loops containing up to five small inner loops. The compiler's heuristics are fusing the inner loops more aggressively in the in vivo version than in the in vitro version. Therefore, the assemblies between in vivo and in vitro do not match.

Without the loop-variable-fix the overall matching is 49%. With the loop-variable-fix, the overall matching increases to 65%.

Figure 8 summarizes the runtime discrepancy between the in vivo and in vitro codelets. Without the loop-variable-fix the overall runtime matching is 58%. With the loop-variable-fix the runtime matching increases to 81.6%.

Table II tallies the codelets in four categories depending if their assembly match, their runtime match, both match, neither match:

a) Assembly and Runtime matches: the most frequent scenario with 52.1% of the codelets. Most of the time if assembly is the same for the codelet and the application, the in vivo and in vitro runtime will be similar.

b) Nothing matches: 6.9% of the codelets do not match at assembly nor at runtime.

c) Only Assembly matches: This category considers codelets which have the same assembly but have different in vivo and in vitro runtimes. It's an uncommon case that happens for seven codelets in MG, FT and SP. In the original benchmarks these seven codelets are invoked multiple times

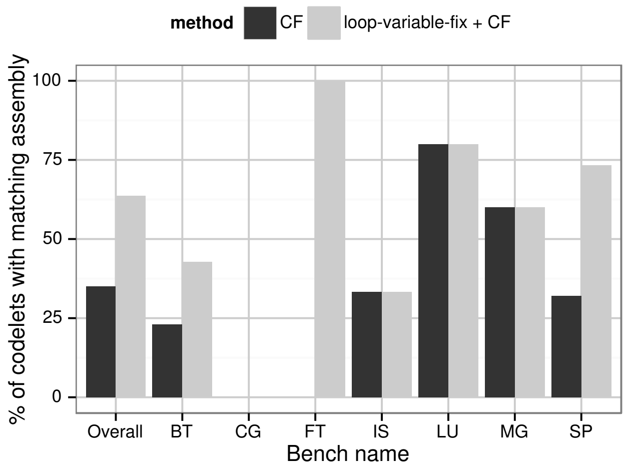


Fig. 7: Percentage of matching assembly codelets between the in vivo and in vitro codelets.

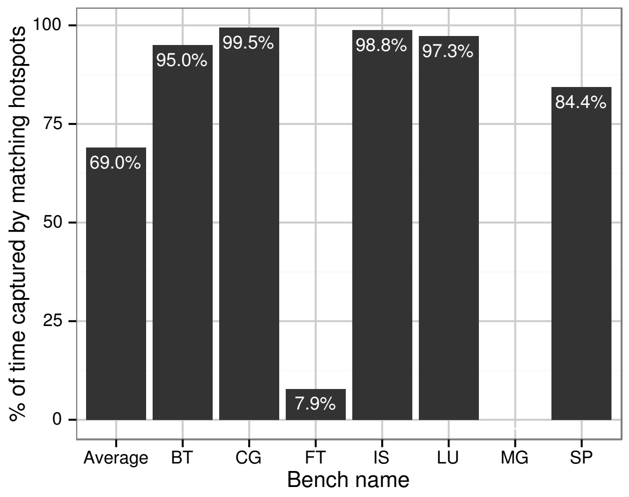


Fig. 9: Percentage of time captured by codelets preserving the original runtime.

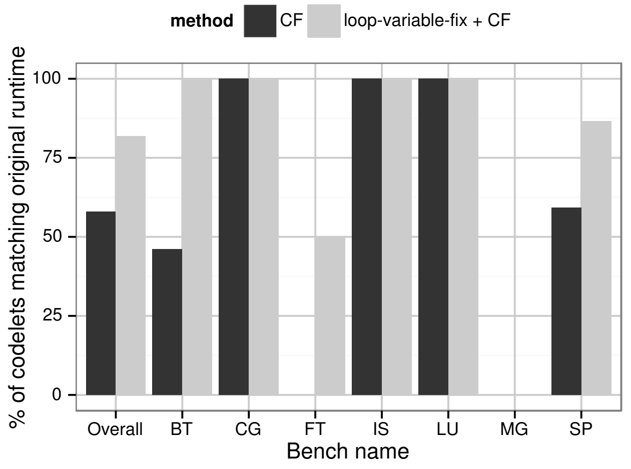


Fig. 8: Percentage of codelets preserving the original runtime.

We partition codelets in four classes:

a) Run **and** Asm : in vivo and in vitro assembly and runtime match
b) Run: in vivo and in vitro runtime match
c) Asm: in vivo and in vitro assembly match
d) Nothing: in vivo and in vitro match neither at runtime nor at assembly

Bench	Run **and** Asm (%)	Run (%)	Asm (%)	Nothing(%)
BT	42.9	57.1	0.0	0.0
CG	0.0	100.0	0.0	0.0
FT	50.0	0.0	50.0	0.0
IS	33.3	66.7	0.0	0.0
LU	80.0	20.0	0.0	0.0
MG	0.0	0.0	60.0	40.0
SP	66.7	20.0	6.7	6.7
Overall	52.1	29.5	11.5	6.9

TABLE II: Runtime and Assembly matching between codelets. Most of the codelets with matching assembly have matching runtimes.

with different dataset sizes. Because capturing memory dumps is costly, Codelet Finder captures only the dataset for the first call of a hotspot. Therefore the codelet execution matches only the first invocation inside of the application. In our runtime matching statistics we have counted this as a discrepancy because the codelet runtime only matches the first invocation but none of the others. If the user wants to study a different dataset, Codelet Finder can be configured to capture the n^{th} dataset instead of the first.

d) Only Runtime matches: This category considers codelets with matching runtime but different assembly between the in vivo and in vitro version. 29.5% of the codelets are in this scenario. The assembly is different because of compiler optimizations. The fact that the runtime is similar means that the optimizations did not impact performance of the hotspot. This is for example the case in CG, which has 0% assembly matching, but achieves 100% runtime matching. CG does not match at the assembly level because of aggressive loop fusion in the in vivo codelet. Nevertheless, the fusion does not affect the performance, therefore the in vivo and in vitro runtime match.

Overall for the NAS benchmarks, 63.6% of the codelets match the original hotspot assembly, and on our test architecture 81.6% of the codelets match the original runtime.

Codelets can therefore be used to optimize or benchmark an application most of the time. But discrepancies should be expected. Depending on the performance study goals, these discrepancies can or cannot be tolerated.

Instead of counting how many codelets have matching runtime, Figure 9 shows what percentage of the original application runtime is captured by matching codelets. On average matching codelets capture 69.0% of the original application runtime.

We say that a codelet matches the original runtime if the

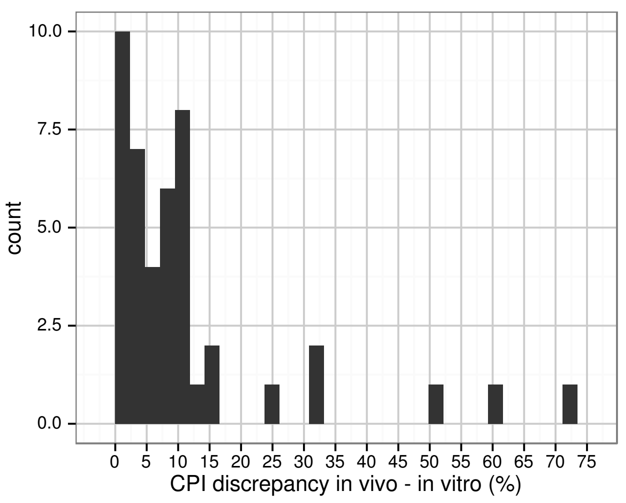


Fig. 10: Distribution of the percentage discrepancy in CPI between the in vivo and in vitro codelets.

difference in CPI and retired instructions between in vivo and in vitro is less than 15%. But what is the median discrepancy one can expect when using codelets? Figure 10 gives the distribution of the runtime discrepancies between the in vivo and in vitro codelets of NAS. The median CPI discrepancy over all the codelets is 7.6%.

Discrepancies mainly occur due to aggressive compiler optimizations. By lowering the optimization level, we can dramatically increase the number of assembly matching codelets. For instance, when compiling with `icc/ifort -O0`, 88.6% of NAS codelets have the same assembly in vivo and in vitro. The 11.4% remaining discrepancies are due to small differences in the memory addressing. For example, function `compute_rhs` in SP uses a global array named `rhs` which in the codelet becomes a locally allocated array. Because the former is accessed using an absolute memory address and the later is addressed through the stack, the assembly is slightly different.

When using code isolation in a performance study, we recommend to check which set of codelets match and discard the others. If the runtime matching analysis is too costly, the assembly matching analysis may be used in place. Indeed, we show that codelets matching at the assembly level, usually match at runtime. The only exception which happens for seven codelets is when hotspots are called with many different datasets in the same application.

VII. Conclusion

This paper examines the viability of code isolation for optimization and benchmarking. Code isolation allows to extract the hotspots from a given application as independent micro-benchmarks that can be modified, build and run independently from the original application. Working with codelets instead of whole applications lowers considerably the cost of optimization or benchmarking. Yet, codelets can only be used for this purpose if they faithfully represent the original application runtime and assembly characteristics.

For the NAS benchmarks, we conclude that code isolation captures 92.3% of the total running time of the original applications. We show that 63.6% of the isolated codelets are faithful to the original assembly and 81.6% of the codelets match the original runtime. We have implemented a source-to-source transformation tool that automatically performs the loop-variable-fix described in section V-A.

As future works, we would like to extend this study to include more architectures and benchmarks, including industrial HPC programs. In this paper we have shown that codelets can be used for piece-wise characterization of programs; it would be interesting to evaluate in what measure codelets can be used for piece-wise optimization of programs by focusing a performance auto-tuner on each separate codelet.

When using codelets in a performance study we recommend to keep only the codelets that match. The cheap static assembly check should be enough in most of the cases. Codelets have already been used in performance studies. Kashnikov et al. [11] used codelets to study the effects of compiler flags on the performance of 144 benchmarks. Noudohouenou et al. [12] propose a fast performance simulator based on codelets.

Acknowledgements

The authors would like to thank G. Carl Evans, Eric Petit, Sandrine Piriou, and Mihail Popov for their precious help.

This work has been conducted by the Exascale Computing Research laboratory, thanks to the support of CEA, GENCI, Intel, UVSQ. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.

References

[1] D. E. Knuth, "An empirical study of Fortran programs," *Software: Practice and Experience*, vol. 1, no. 2, pp. 105–133, 1971.

[2] Y.-J. Lee and M. Hall, "A code isolator: Isolating code fragments from large programs," in *Languages and Compilers for High Performance Computing*. Springer, 2005, pp. 164–178.

[3] E. Petit, G. Papaure, F. Bodin *et al.*, "Astex: a hot path based thread extractor for distributed memory system on a chip," in *Proceedings of Compilers for Parallel Computers workshop (CPC2006)*, 2006.

[4] C. Liao, D. J. Quinlan, R. Vuduc, and T. Panas, "Effective source-to-source outlining to support whole program empirical optimization," in *Languages and Compilers for Parallel Computing*. Springer, 2010, pp. 308–322.

[5] CAPS entreprises. Codelet finder. [Online]. Available: http://www.caps-entreprise.com/

[6] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, "The nas parallel benchmarks summary and preliminary results," in *Supercomputing, 1991. Supercomputing'91. Proceedings of the 1991 ACM/IEEE Conference on*. IEEE, 1991, pp. 158–165.

[7] A. Fog, "The microarchitecture of intel and amd cpu's: An optimization guide for assembly programmers and compiler makers," *Copenhagen University College of Engineering*, 2013.

[8] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, W. Jalby *et al.*, "Maqao: Modular assembler quality analyzer and optimizer for itanium 2," in *The 4th Workshop on EPIC architectures and compiler technology, San Jose*, 2005.

[9] Y. Kashnikov, P. de Oliveira Castro, E. Oseret, and W. Jalby, “Evaluating Architecture and Compiler Design through Static Loop Analysis,” in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. IEEE Computer Society, (to appear) 2013.

[10] J. Treibig, G. Hager, and G. Wellein, “Likwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*. IEEE, 2010, pp. 207–216.

[11] Y. Kashnikov, J. C. Beyler, and W. Jalby, “Compiler optimizations: Machine learning versus o3,” in *Proceedings of the 25th international conference on Languages and Compilers for Parallel Computing*, ser. LCPC’12, Tokyo, Japan, 2012.

[12] J. Noudohouenou, V. Palomares, W. Jalby, D. C. Wong, D. J. Kuck, and J. C. Beyler, “Simsys: a performance simulation framework,” in *Proceedings of the 2013 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*. ACM, 2013, p. 1.