# Piecewise Holistic Autotuning of Compiler and Runtime Parameters

Mihail Popov[1], Chadi Akel[2], William Jalby[1], and Pablo de Oliveira Castro[1]

[1] Université de Versailles Saint-Quentin-en-Yvelines, Université Paris-Saclay
{mihail.popov,william.jalby,pablo.oliveira}@uvsq.fr
[2] Exascale Computing Research chadi.akel@exascale-computing.eu

**Abstract.** Current architecture complexity requires fine tuning of compiler and runtime parameters to achieve full potential performance. Autotuning substantially improves default parameters in many scenarios but it is a costly process requiring a long iterative evaluation.
We propose an automatic piecewise autotuner based on CERE (Codelet Extractor and REplayer). CERE decomposes applications into small pieces called codelets: each codelet maps to a loop or to an OpenMP parallel region and can be replayed as a standalone program.
Codelet autotuning achieves better speedups at a lower tuning cost. By grouping codelet invocations with the same performance behavior, CERE reduces the number of loops or OpenMP regions to be evaluated. Moreover unlike whole-program tuning, CERE customizes the set of best parameters for each specific OpenMP region or loop.
We demonstrate CERE tuning of compiler optimizations, number of threads and thread affinity on a NUMA architecture. On average over the NAS 3.0 benchmarks, we achieve a speedup of `1.08`× after tuning. Tuning a single codelet is `13`× cheaper than whole-program evaluation and estimates the tuning impact on the original region with a 94.7% accuracy. On a Reverse Time Migration (RTM) proto-application we achieve a `1.11`× speedup with a `200`× cheaper exploration.

## 1 Introduction

The current increase of architecture complexity, multiple cores, out-of-order execution, complex memory hierarchies, and non-uniform memory access (NUMA) complicates the performance characterization. Achieving full efficiency requires fine tuning parameters such as the degree of parallelism, thread placement or compiler optimization. Runtime and compiler standard parameter levels (such as `-O3` compiler flag or `scatter` thread placement) achieve good-enough performance across most of the codes and architectures. But they cannot take advantage of target-specific optimizations since they must correctly work on a large panel of architectures.

Finding the optimal parameters may lead to substantial improvement but is a costly and time consuming process. For example, compilers such as LLVM [1] `3.4` provide more than sixty passes. Passes have different impact depending on their order of execution and can be executed many times. This leads to a huge exploration space: considering only sequences of 30 passes requires to explore a space over $60^{30}$ points.

Even worse, some applications may have different optimal parameters for different code regions. For example, compute bound loops and memory bound loops within the same function will not be sensitive to the same compiler optimizations.

There are different approaches to tune parameters. Iterative compilation [2] is a well known automated search method for solving the compiler optimization phase ordering problem. The idea is to apply successive compiler transformations to a program and to evaluate them by executing the resulting code. Similar execution driven studies [3, 4] explore the efficiency of different thread placement strategies or frequencies. Smart search algorithms [5, 6] through the parameter space reduce the evaluation cost. Genetic algorithms [7, 8] or adaptive learning [9, 10] accelerate the search by avoiding unnecessary parameters.

A common point of these execution driven studies is that they require a full program evaluation and execution to quantify the impact of a single parameter value. The problem is that executing application is costly and time consuming, especially if we have thousands of points to evaluate. Also, as regions of code do not benefit from the same parameters, an overall program-evaluation (or *monolithic* evaluation) is not able to achieve the optimal per region optimization. In other words, these studies are expensive to perform and do not necessary lead to the optimal parameters.

In this paper we propose a piecewise exploration framework based on CERE [11] (Codelet Extractor and REplayer) which enhance both the search cost and the search benefits. We partition applications into small pieces called *codelets*. Each independent loop or OpenMP parallel region is extracted as a codelet that can be replayed as a standalone program. Instead of evaluating parameters on the whole application, we separately evaluate them on each codelet (section 3.2). The piecewise evaluation leads to find the best parameters for each region. Combining these regions within a single binary is called *hybridization* and outperforms traditional *monolithic* tuning (section 3.3).

Using codelets as proxies for autotuning requires that codelets faithfully reproduce the application behavior with the exploring parameters. This requires a warmup of the memory state. CERE already implements various warmup strategies. To enable thread placement exploration, we extend these warmups with a new NUMA ownership strategy (section 3.1).

The contributions of this paper are:

- A novel automatic autotuner based on codelets and integrated in CERE.
- A holistic piecewise tuning approach that addresses degree of parallelism, thread placement, NUMA effects, and compiler optimization passes.
- The validation of the codelet tuning over the NAS benchmarks and an industrial proto-application with compiler and runtime parameters.
- A NUMA aware memory page capture and replay.

## 2 Motivating Example

| thread | affinity | xsolve | ysolve | zsolve | rhs | total |
|---|---|---|---|---|---|---|
| s2 | 0;8 | 32.3 | 23 | 28.5 | 23 | 106.8 |
| c2 | 0;1 | 21.4 | 17.6 | 18.1 | 23.7 | 80.8 |
| h2 | 0;16 | 40 | 32.6 | 23 | 46.1 | 141.7 |
| s4 | 0;8;1;9 | 25.9 | 20.9 | 26 | 12.1 | 84.9 |
| c4 | 0;1;2;3 | 15.5 | 12.7 | 13.8 | 13.2 | 55.2 |
| h4 | 0;16;1;17 | 23.8 | 17.5 | 16 | 24.3 | 81.5 |
| s8 | 0;8;1;9;...;11 | 24.4 | 21.9 | 28.6 | 6.9 | 81.8 |
| c8 | 0;1;2;3;...;7 | 14.4 | 13.4 | 14.3 | 9.1 | 51.2 |
| h8 | 0;16;1;17;...;19 | 17.7 | 14.2 | 13.9 | 13.5 | 59.3 |
| s16 | 16 scatter | 25.1 | 21.4 | 35.5 | 5.3 | 87.4 |
| c16 | 16 compact | 17 | 15 | 15.5 | 9.7 | 57.2 |
| h32 | 32 scatter | 36 | 31.2 | 38.9 | 6.4 | 112.4 |

**Table 1.** Execution time in megacycles of SP parallel regions across different thread affinities with `-O3` optimization. For $n$ threads, we consider three affinities: scatter $s_n$, compact $c_n$, and hyperthread $h_n$. Executing SP with the c8 affinity provides an overall speedup of `1.71`× over the standard (s16).

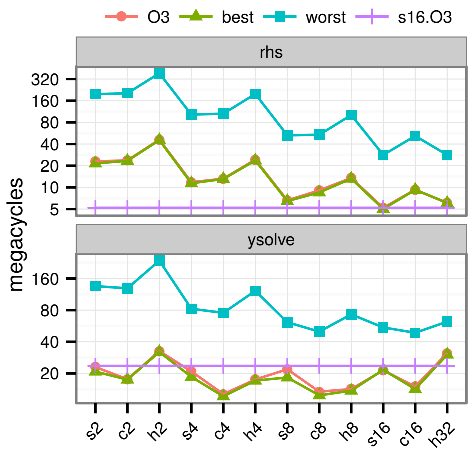

**Fig. 1.** Tuning exploration for two SP regions. For each affinity, we plot the best, worst, and `-O3` optimization sequences. Custom optimization beats `-O3` for `s2`,`s4`, and `s8` on `ysolve`.

We will demonstrate how CERE operates on SP, a Scalar Penta-diagonal solver, from the C version of the NPB 3.0 OpenMP benchmarks [12]. CERE autotuning achieved a `1.82`× performance speedup over the standard parameters levels. Thanks to the CERE codelet approach, the exploration time was approximately five times cheaper compared to the whole-program iterative compilation.

CERE starts by profiling SP and automatically selecting representative OpenMP regions to tune. `Xsolve`, `ysolve`, `zsolve`, and `rhs` are selected and cover 93% of SP execution time. CERE extracts these regions as codelets and tunes them with a holistic exploration across three dimensions: thread number, thread placement, and LLVM compiler passes. Once satisfying parameters are found, CERE produces an hybrid application where each region uses the best found parameters.

In this study, we explored the interactions between `12` thread configurations combining different number of threads and affinity mappings and `150` LLVM optimization sequences generated using the random sub-sampling presented in section 4. Combining them produces an exploration space of `1800` points, which gives an insight of how costly it is to simultaneously tune multiple parameters.

Figure 1 shows the performance of two SP parallel regions across this exploration space. We notice that there is a strong interaction between the compiler and the thread parameters as they both significantly impact the performances. Moreover, the best parameters are different for the two regions: scatter placement is best for `rhs` while compact benefits `ysolve`.

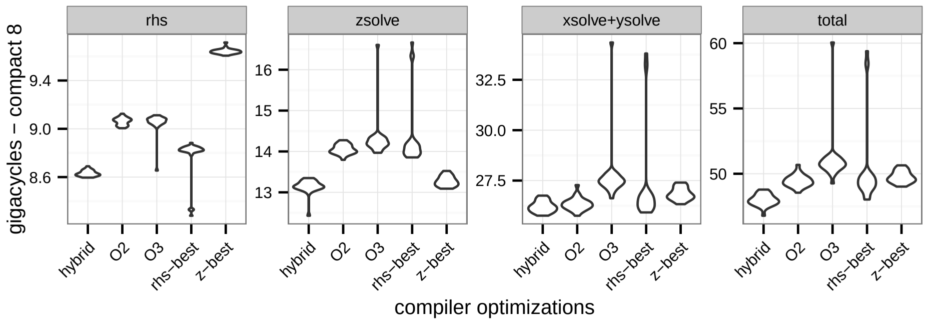

**Fig. 2.** Violin plot execution time of SP regions using best NUMA affinity. Measures were performed 31 times to ensure reproducibility. When measuring total execution time, Hybrid outperforms all other optimization levels, since each region uses the best optimization sequence available.

CERE makes it possible, through codelet replay, to independently explore each region. Moreover, thanks to CERE replay prediction model presented in section 3.2, it is possible to quickly evaluate the impact of each configuration on only a few datasets. CERE evaluates thread affinities and compiler optimizations on SP, respectively `5.84`× and `4.52`× times faster than a full application evaluation while keeping a low average error of `2.33%`.

Custom parameters outperform the standard `16 threads scatter s16 -O3` on SP. Table 1 shows the performance of different thread affinities compiled with `-O3`. The best custom thread affinity `0;1;2;3;4;5;6;7` (single NUMA socket) achieves a speedup of `1.71`× over the standard `16 threads scatter` (two NUMA sockets).

We explored with CERE `350` compiler optimization sequences on the best single NUMA configuration found above. `Xsolve` and `ysolve` work best at the default `-O2 level`, but a custom best sequence is found for `zsolve` and `rhs`. Figure 2 shows the performance of each region compiled with the default optimization and the best custom sequences. No single sequence is the best for all regions. CERE hybrid compilation produces a binary where each region is compiled using its best sequence, achieving a speedup that cannot be reproduced using traditional monolithic compilation.

## 3 CERE AutoTuner

CERE [11, 12] is an open source framework for code isolation. CERE finds and extracts loops or OpenMP parallel regions from an application as isolated fragments of code, called codelets. Codelets can be modified, compiled, run, and measured independently from the original application.

Figure 3 presents how a region is captured as a codelet and replayed. Using codelets as a proxy for application characterization requires two steps: *capture* and *replay*. During the capture, the execution state is saved for each region.

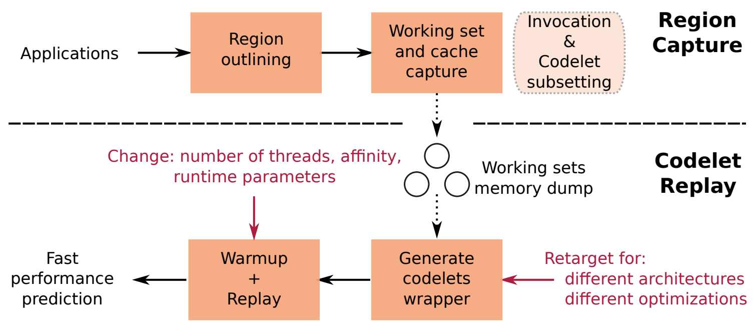
Applications
Region outlining
Working set and cache capture
Invocation & Codelet subsetting
Region Capture
Change: number of threads, affinity, runtime parameters
Working sets memory dump
Codelet Replay
Fast performance prediction
Warmup + Replay
Generate codelets wrapper
Retarget for: different architectures different optimizations

**Fig. 3.** Codelet capture and replay workflow

During the replay, CERE restores the codelet memory and cache state before executing the region. At replay, a cache and NUMA page ownership warmup is necessary to ensure that the replay execution context is close to the original.

CERE extracts regions at the compiler Intermediate Representation (IR) level after clang front-end translation but before LLVM middle-end optimizations. This allows us to re-target the codelet compilation and execution.

## 3.1 NUMA Aware Warmup

A replay has to faithfully reproduce the original invocation context. CERE already handles two issues: it restores the memory working set of the region and warms up the cache to avoid cold-start bias [13].

It uses a snapshot of the memory at page level granularity. With a memory protection mechanism, the memory pages containing the working set are captured. During replay, pages are remaped to their original addresses. CERE includes different cache warmup approaches [11] that operate by replaying the memory access history at a page granularity before running the codelet.

We outline a new aspect: the placement of the pages across the NUMA nodes. Due to the node local first touch policy, a page is mapped to the core which first attempts to use it. We must ensure that pages are mapped to the same NUMA nodes as they have been in the original run. The problem is that pages are not necessarily bound to the same NUMA nodes across the different thread affinities. `Scatter` maximizes the number of NUMA nodes while `compact` minimizes it.

Figure 4 outlines this problem on a 2-NUMA nodes architecture. CERE default warmup uses a single thread to remap the pages to their original addresses: all the pages are bound to a single NUMA node. Replays accurately predict the execution time as long as the affinity binds the threads to the same NUMA node. Otherwise, the replay pays NUMA latencies that do not appear in the original run and which cause prediction discrepancies.

To solve this issue, we enhance the page capture by saving, for each page, the first thread that touches it. During replay, before replaying the codelet code, each thread touches the pages that it has saved at the capture. Hence, pages are mapped to the NUMA node of the thread which is the first to touch them.

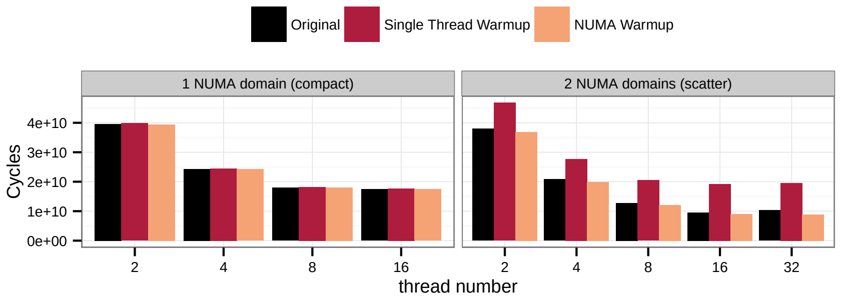

**Fig. 4.** Prediction accuracy of a single threaded warmup versus a NUMA aware warmup on BT `xsolve`. Only a NUMA aware warmup is able to predict this region execution time on a multi NUMA node configuration.

To ensure a correct NUMA mapping at replay when we change the number of threads, we must both not exceed the number of threads at capture and spread the pages across the replaying threads.

### 3.2 Piecewise Optimization with Codelets

Regions within an application may not be sensitive to the same optimizations: SP `rhs` and `zsolve` regions in section 2 have different best compiler optimizations. Unlike monolithic approaches, CERE enables tuning each codelet independently.

The piecewise search not only improves the benefits over a monolithic tuning, but also accelerates the exploration by avoiding the execution of useless compiler sequences (see in experiments Fig. 7) or regions. IS benefits from this as it only times a sorting algorithm included in a region which represents `22%` of the application execution time. Through a codelet, CERE extracts the sorting region and tunes it without executing the rest of the application.

Codelets also accelerate the evaluation of each region. Regions may have performance variations across their different invocations. Using a clustering method, CERE classes these invocations and selects a representative subset of invocations to be replayed. We only execute the subset to predict the region execution time.

We assume that the tuning parameters have a similar impact on the invocations within the same cluster. Figure 5 illustrates this assumption across two parameters on MG `resid`. `Resid` has 42 invocations grouped in 3 performance classes. The invocations remain in the same classes across the parameters. So, by replaying 3 instead of 42 invocations, CERE predicts the region execution for each parameter to explore.

### 3.3 Hybrid Compilation

The piecewise tuning finds the best compiler optimizations for each loop and OpenMP region. Unfortunately, LLVM does not provide a mechanism to select compiler optimizations at the function or loop granularity. To compile each region with a different set of optimizations we must extract each region in its own

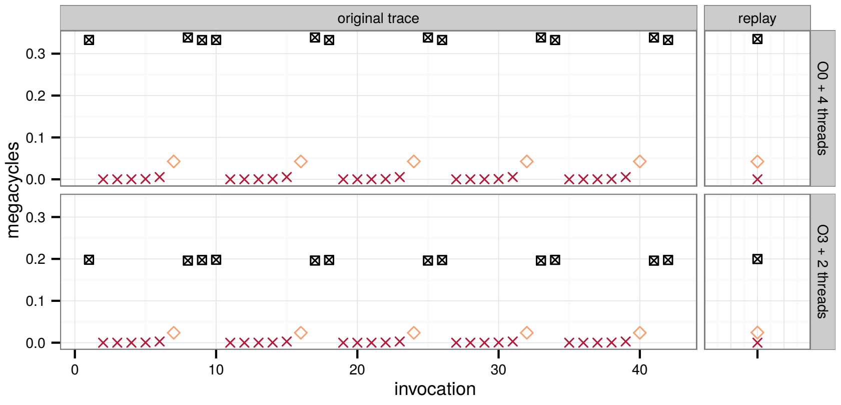

**Fig. 5.** MG `resid` invocations execution time on Sandy Bridge over `-O3` and `-O0` with respectively 2 and 4 threads. Each representative invocation predicts its performance class execution time.

compilation unit. We leverage the `extract` tool included in LLVM which allows to extract an IR function to a separate IR file.

The first step is outlining each region of interest in its own IR function. Before any middle-end optimization is applied, each region is moved to a separate compilation unit using `LLVM extract`. A special pass changes the visibility of symbols used by the extracted region from internal to global so that they are not removed by the compiler. Then, the best compiler sequence found is applied to each separate IR file and an object file is produced. Finally, all the objects files are linked together producing an hybrid binary.

## 4 Experiments and Validation

This section validates both usage of codelets as proxies to tune parameters and production of hybrid binaries. Codelets capture most of the application hotspots [11]. Nevertheless, we must demonstrate that codelet tuning helps finding optimal parameters and reducing the search cost. To accurately predict best parameters, codelet replays must capture the original application reaction to the different compiler and thread configurations.

We used two different Intel CPU micro-architectures: a Sandy Bridge E5 with 64 GB of RAM and an Ivy Bridge i7-3770 with 16 GB of RAM. We chose Sandy Bridge to explore thread affinities because it has 2 NUMA sockets and each socket has 8 physical (16 hyper-threaded) cores.

Thread configurations were selected to explore different degrees of parallelism, NUMA and hyper-threading effects. Sandy Bridge has 16 physical cores, so we did not explore configurations beyond 32 threads. We used the Intel kmp affinity [14] notation to characterize the thread placement. Cores ranked between 0 and 7 reference the physical cores of the first NUMA node while cores between 8 and 15 reference the physical cores of the second NUMA node. Similarly, cores from 16 to 23 and from 24 to 31 reference the hyper-threaded cores of respectively the first and the second NUMA node.

The compilation search was performed on `LLVM 3.4` using a random pass selection. We use `LLVM opt` and `llc` to change respectively middle-end and back-end optimizations. Middle-end passes have different impact depending on their order of execution, and can be executed multiple times. `-O3` is a manually tuned sequence composed of 65 ordered passes aiming to provide good performances. In this paper, random compilation sequences were generated by down-sampling the `-O3` default sequence. Each pass was removed with a 0.7 probability, and the process was repeated four times to explore the impact of pass repetitions. We empirically found that this generation method produces good and diverse candidates. Back-end passes were selected among `-O0`,`-O1`,`-O2` and `-O3`.

We performed the experiments on the NAS 3.0 sequential [15] and C OpenMP parallel [16] benchmarks (respectively NAS SER and NPB) with CLASS A datasets and on a Reverse Time Migration [17] (RTM) proto-application.

### 4.1 Thread Number and Affinity Tuning

This section presents the thread affinity tuning results. CERE page memory capture was performed on a `16 threads scatter` run. Table 2 evaluates CERE thread affinities replay accuracy and reduction factor over NAS OpenMP. We focused on regions representing more than `5%` of the application execution time. On average, a region exploration is `6.55`× faster with codelets than with whole-program evaluations. Tuning all the SP regions from the motivating example with codelets is five times faster as SP has four regions with an average acceleration of twenty per region. CERE uses an optimistic warmup: it replays four times the codelet over itself. These replays are not amortized on EP and MG: the first executes the main parallel region once in the original execution while the second requires many invocation replays to support the multiple performance classes. As we increase the data sets, the warmup cost overhead becomes smaller compared to the replay execution time. We tested `xsolve` BT with CLASS B data sets and a single warmup invocation to achieve an acceleration of `9.48`×, twice the one achieved in class A, with an accuracy of `98.36%`.

The average CERE prediction accuracy is `93.66%`. It allows the autotuner to outperform the standard scatter `s16` over `EP`, `FT`, `LU`, and `SP` and to perform an average speedup of `1.40`× (see Fig. 6). We note that there is no thread affinity to privilege over the others: `h32`, `s16`, and `c8` are all optimal on at least two applications.

### 4.2 Compiler Passes Tuning and Hybridization

Table 2 also presents CERE predictions through compiler optimizations with `3000` compiler sequences for `BT`, `500` for `MG` and `1000` for the others NAS SER. The average CERE prediction accuracy and acceleration for a region is `95.8%` and `20.61`×. Figure 7 presents the number of explored compiler sequences required

| | Compiler passes | | | Thread affinity | | |
|---|---|---|---|---|---|---|
| **Benchmarks** | #Regions | Accuracy | Reduction factor | #Regions | Accuracy | Reduction factor |
| BT | 3 | 98.73 | 79.63 | 4 | 95.24 | 5.28 |
| CG | 2 | 98.65 | 3.39 | 2 | 79.48 | 1.23 |
| FT | 5 | 98.3 | 2.6 | 5 | 90.71 | 2.17 |
| IS | 3 | 96.64 | 1.26 | 2 | 94.85 | 1.04 |
| SP | 6 | 98.78 | 68.9 | 4 | 97.66 | 20.07 |
| LU | 7 | 95.04 | 8.49 | 2 | 99.00 | 12.64 |
| EP | 1 | 83.08 | 0.36 | 1 | 99.31 | 0.25 |
| MG | 4 | 97.22 | 0.28 | 4 | 93.04 | 0.45 |
| **Average** | | 95.8 | 20.61 | | 93.66 | 5.39 |

**Table 2.** The **accuracy** of the codelet prediction is the relative difference between the original and the replay execution time. The benchmark **reduction factor** or acceleration is the exploration time saved when studying a codelet instead of the whole application. CERE fails to accelerate EP and MG evaluation: EP has a single region with one invocation while MG displays many performance variations.

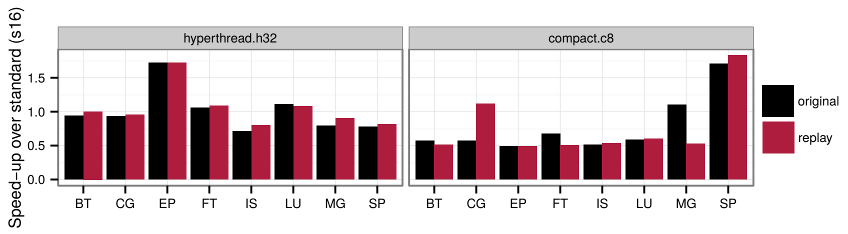

**Fig. 6.** Original and CERE predicted speedup for two thread configurations. Replay speedup is the ratio between the replayed target and the replayed standard configuration. CERE accurately predicts the best thread affinities in six out of eight benchmarks. For CG and MG, we miss-predict configurations that use all the physical cores.

to achieve a speedup over `1.04`× per region. We empirically determined this speedup value. Unlike monolithic approaches which must continue exploration until all regions are optimized, codelets can stop the search over a region once a satisfying speedup is found and focus the exploration on other regions. Here, CERE evaluates `BT ysolve 461` times instead of `3000` times. Each evaluation is on average `99` times cheaper than a full application run due to the codelet invocations clustering.

The focus of this paper is not on the compiler flag selection, that is why a naive random compiler pass search was used. Nevertheless, CERE results could be improved with more sophisticated techniques for passes selection such as genetic algorithms [6] which would also benefit from the piecewise approach.

CERE outperforms the standard `-O3` over `BT`, `SP`, and `IS` with an average speedup of `1.06`× (see Fig. 8). `IS` random generator and sorting algorithm do not benefit from the same optimizations which explains the significant difference between the hybrid and the monolithic approach. Hybrid binaries based on original or replay explorations have the same performances which ensure that we do not miss any optimizations through the codelets.

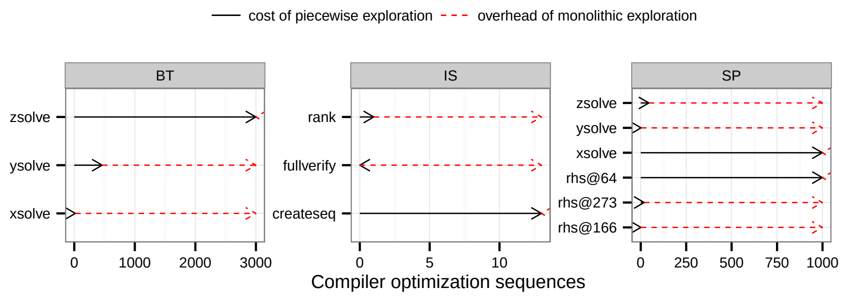

**Fig. 7.** Compiler sequences required to get a speedup over `1.04`× per region. CERE evaluates the sequences in the same order for all the regions. Exploring regions separately is cheaper because we stop tuning a region as soon as the speedup is reached.

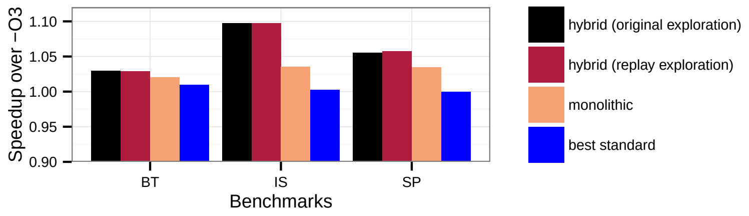

**Fig. 8.** Speedups over `-O3`. We only observe speedups from the iterative search over `BT`, `SP`, and `IS`. Best standard is the more efficient default optimization (either `-O1`, `-O2`, or `-O3`). Monolithic is best whole program sequence optimization. Hybrids are build upon optimizations found either with codelets or with original application runs.

We make the simplifying assumption that optimizing a region does not affect other regions. This is not always true: due to memory effects, it is possible to have performance interactions between neighbors. We find a compilation sequence which gives a speedup of `x1.08`× over LU `jacu`. Unfortunately, optimizing `jacu` has the side effect of slowing down by `0.92`× the neighboring region `jacld`.

To stress the CERE prediction accuracy model, we performed a simultaneous search of 1000 compiler sequences across the thread affinities on LU `ssor`. CERE predicted region execution time with a mean accuracy of 99% across parameters.

Finally, we used CERE to tune the RTM proto-application used in a imaging system for geophysical depth, and provided by Asma Farjallah and Total [9]. RTM is dominated by one Jacobi stencil computation called 3000000 times and which represents `91.1%` of the total execution time. CERE extracts this loop and performs a compiler search of 300 passes. This codelet is `200`× faster to evaluate and finds a compiler optimization `1.11`× faster than `-O3`.

## 5 Related Work

While most of the research try to accelerate the iterative compilation by pruning the exploration space [5–8, 10], this paper proposes a transverse approach which do not focus on the search space but rather accelerates the evaluation of each exploration point through a benchmark reduction technique.

Usual benchmark reduction techniques take advantage of phases to reduce the simulation cost [18]. They cannot be directly used for compiler tuning as they operate on the assembly. Fursin and al. [19] managed to take advantage of the application phases: they evaluate multiple optimizations for a region with a single run by versioning the different iterations of the region. However, they do not use any code isolation techniques so they cannot focus the search which is problematic when a region of interest has a few invocations compared to the others. Oliveira et al. [20] cluster together codelets that have the same performance behavior, and keep only one representative copy for each group. This benchmark reduction is complementary to the invocations clustering presented in this paper and should accelerate the overall search. We must find clustering metrics that are relevant for compiler optimizations and thread affinities.

Like us, Kulkarni and al. [21] propose a piecewise search at the function level granularity. They propose a per-function compilation using the VPO compiler framework. Yet, they do not use any extraction mechanism during the search: exploring two functions within the same file requires to execute the program many times. Purini and al. [22] find, through LLVM iterative compilation runs, good general sets of compilation sequences that should work well on any given program. They can quickly tune new applications by directly searching passes within the good set instead of exploring the whole optimization space. Codelets could serve proxies to quickly find and test these optimal sequences.

## 6 Conclusion

In this paper we present an autotuner based on CERE codelets. Codelets serve as proxies for tuning applications holistically, considering the interactions of thread placements, NUMA effects, and compiler passes. CERE proposes a novel piecewise approach that accelerates searching the parameter space and enables an hybrid compilation where each region uses the best set of local parameters. It outperforms traditional monolithic tuning.

CERE codelets predict the impact of thread placement and compiler optimization with a mean accuracy of 94.7% over the NAS 3.0 benchmarks. On the RTM industrial proto-application, CERE achieved a `1.11`× execution speedup through compiler pass selection. The search was `200`× faster thanks to codelet tuning. Detailed accuracy and acceleration reports are available at `https://benchmark-subsetting.github.io/autotuning-results/`.

**Acknowledgments:** The research leading to these results has received funding under the Mont-Blanc project from the European Union's Horizon 2020 research and innovation program under grant agreement No 671697.

## References

1. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: International symposium on Code Generation and Op-

timization, IEEE (2004) 75–86
2. Kisuki, T., Knijnenburg, P.M., O'Boyle, M.F., Bodin, F., Wijshoff, H.A.: A feasibility study in iterative compilation. In: High Performance Computing, Springer (1999) 121–132
3. Mazouz, A., Touati, S.A.A., Barthou, D.: Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In: High Performance Computing and Simulation (HPCS), IEEE (2011) 273–279
4. Rountree, B., Lownenthal, D.K., de Supinski, B.R., Schulz, M., Freeh, V.W., Bletsch, T.: Adagio: making dvs practical for complex hpc applications. In: Proceedings of the conference on Supercomputing, ACM/IEEE (2009) 460–469
5. Triantafyllis, S., Vachharajani, M., Vachharajani, N., August, D.I.: Compiler optimization-space exploration. In: Code Generation and Optimization, 2003. CGO 2003. International Symposium on, IEEE (2003) 204–215
6. Ladd, S.R.: Acovea: Analysis of compiler options via evolutionary algorithm (2007)
7. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: SIGPLAN Notices. Volume 34., ACM (1999) 1–9
8. Hoste, K., Eeckhout, L.: Cole: compiler optimization level exploration. In: Code generation and optimization, ACM (2008) 165–174
9. de Oliveira Castro, P., Petit, E., Farjallah, A., Jalby, W.: Adaptive sampling for performance characterization of application kernels. Concurrency and Computation: Practice and Experience (2013)
10. Fursin, G., et al.: Milepost gcc: Machine learning enabled self-tuning compiler. International Journal of Parallel Programming **39**(3) (2011) 296–327
11. de Oliveira Castro, P., Akel, C., Petit, E., Popov, M., Jalby, W.: CERE: LLVM Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization. Transactions on Architecture and Code Optimization **12**(1) (2015) 6
12. Popov, M., Akel, C., Conti, F., Jalby, W., de Oliveira Castro, P.: Pcere: Fine-grained parallel benchmark decomposition for scalability prediction. In: International Parallel and Distributed Processing Symposium, IEEE (2015) 1151–1160
13. Kessler, R.E., Hill, M.D., Wood, D.A.: A comparison of trace-sampling techniques for multi-megabyte caches. Transactions on Computers **43**(6) (1994) 664–675
14. Intel: Reference Guide for the Intel(R) C++ Compiler 15.0. `https://software.intel.com/en-us/node/522691`
15. Bailey, D., et al.: The NAS parallel benchmarks summary and preliminary results. In: Proceedings of the conference on Supercomputing, ACM/IEEE (1991) 158–165
16. Popov, M.: NAS 3.0 C OpenMP. `http://benchmark-subsetting.github.io/cNPB`
17. Baysal, E.: Reverse time migration. Geophysics **48**(11) (November 1983) 1514
18. Sherwood, T., Perelman, E., Calder, B.: Basic block distribution analysis to find periodic behavior and simulation points in applications. In: Parallel Architectures and Compilation Techniques, IEEE (2001) 3–14
19. Fursin, G., Cohen, A., O'Boyle, M., Temam, O.: Quick and practical run-time evaluation of multiple program optimizations. T. HiPEAC **1** (2007) 34–53
20. de Oliveira Castro, P., Kashnikov, Y., Akel, C., Popov, M., Jalby, W.: Fine-grained Benchmark Subsetting for System Selection. In: International symposium on Code Generation and Optimization, ACM (2014) 132–142
21. Kulkarni, P.A., Jantz, M.R., Whalley, D.B.: Improving both the performance benefits and speed of optimization phase sequence searches, ACM (2010) 95–104
22. Purini, S., Jain, L.: Finding good optimization sequences covering program space. Transactions on Architecture and Code Optimization **9**(4) (2013) 56