

Piecewise Holistic Autotuning of Parallel Programs with CERE

Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, Pablo de Oliveira Castro

▶ **To cite this version:**

Mihail Popov, Chadi Akel, Yohan Chatelain, William Jalby, Pablo de Oliveira Castro. Piecewise Holistic Autotuning of Parallel Programs with CERE. Concurrency and Computation: Practice and Experience, 2017, 29 (15), pp.e4190. 10.1002/cpe.4190 . hal-01542912v2

HAL Id: hal-01542912
https://hal.uvsq.fr/hal-01542912v2

Submitted on 28 Nov 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Piecewise Holistic Autotuning of Parallel Programs with CERE

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

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

This is the pre-peer reviewed version of the following article: Piecewise Holistic Autotuning of Parallel Programs with CERE, M.Popov, C. Akel, Y. Chatelin, W. Jalby, P. de Oliveira Castro, Concurrency and Computation: Practice and Experience (2017). DOI: 10.1002/cpe.4190 which has been published in final form at http://onlinelibrary.wiley.com/doi/10.1002/cpe.4190/full. This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Self-Archiving.

Abstract. Current architecture complexity requires fine tuning of compiler and runtime parameters to achieve best performance. Autotuning substantially improves default parameters in many scenarios but it is a costly process requiring long iterative evaluations.

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 the CERE tuning of compiler optimizations, number of threads, thread affinity, and scheduling policy on both NUMA and heterogeneous architectures. Over the NAS benchmarks, we achieve an average speedup of 1.08× after tuning. Tuning a codelet is 13× cheaper than whole-program evaluation and predicts the tuning impact with a 94.7% accuracy. Similarly, exploring thread configurations and scheduling policies for a Black-Scholes solver on an heterogeneous big.LITTLE architecture is over 40× faster using CERE.

1 Introduction

This paper is an extended version of the work presented at the 22nd Euro-Par international conference [1]. [3]

The current increase of architecture complexity with multiple cores, heterogeneity, out-of-order execution, complex memory hierarchies, and Non-Uniform Memory Access (NUMA) complicates 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 leads to substantial improvement but is a costly and time consuming process. For example, compilers such as LLVM [2] 3.4 provide more than sixty optimization passes. Passes have different impact depending on their order of execution and can be applied multiple 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 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 [3] 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 [4, 5] explore the efficiency of different thread placement strategies or frequencies. Smart search algorithms [6, 7] through the parameter space reduce the evaluation cost. Genetic algorithms [8, 9] or adaptive learning [10, 11] 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 an 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 necessarily lead to the optimal parameters.

[3] We develop a new multi-process capture technique to accelerate multi-threaded applications capture. We extend both the NUMA capture and the methodology sections by providing additional data and more details about the warmup strategies. A new section discusses how similar codelets can be clustered to accelerate the tuning. Finally, we use CERE to optimize the thread affinity and scheduling on a Juno big.LITTLE heterogeneous architecture.

In this paper, we propose a piecewise exploration framework based on CERE [12] (Codelet Extractor and REplayer) which enhances 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).

We note that regions of code may be executed multiple times in an application lifetime. Invocations sharing a similar execution context have the same performance. Therefore, a single invocation replay is sufficient to characterize them. Codelets exploit this idea to reduce the tuning cost as they can be directly replayed as few times as we want (section 3.3). Moreover, whole regions may share a common performance behavior and be sensitive to the same parameters. By grouping similar regions, CERE can reduce the number of regions to evaluate and therefore further accelerate the tuning process (section 3.4).

CERE not only accelerates the searching process but also improves the tuning performance benefits. The piecewise codelet evaluation allows to find the best parameters for each region. Combining the best parameters for each region within a single binary is called *hybridization* and outperforms traditional *monolithic* tuning (section 3.5).

Using codelets as proxies for autotuning requires that codelets faithfully reproduce the application behavior across the search space. In particular this requires a warmup of the memory state. CERE already implements various warmup strategies. To enable thread placement exploration, we extend these warmup strategies with a new NUMA ownership strategy (section 3.1). We improved the memory capture implementation with a new parallel *tracee* and *tracer* memory capture library based on Ptrace. This implementation significantly accelerates the multi-threaded memory capture (section 3.2).

We demonstrate CERE tuning capabilities over a Reverse Time Migration (RTM) proto-application and the NAS benchmarks (section 4). To test the NUMA ownership strategy, we perform the multi-threaded runs on a NUMA architecture. CERE achieves a 1.11× speedup with a 200× cheaper exploration over RTM. We also use CERE to tune multiple thread placement and scheduling strategies for PARSEC Blackscholes over an heterogeneous architecture. CERE accurately predicts the execution time while remaining 40× faster than full application runs (section 5).

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, OpenMP loop scheduling policy, NUMA effects, heterogeneous cores, and compiler optimization passes.
- The validation of thread and compiler configurations tuning through codelets over the NAS benchmarks, PARSEC Blackscholes, and an industrial RTM proto-application.
- A Ptrace based NUMA aware memory page capture.

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;2;10;3;11	24.4	21.9	28.6	6.9	81.8
c8	0;1;2;3;4;5;6;7	14.4	13.4	14.3	9.1	51.2
h8	0;16;1;17;2;18;3;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).

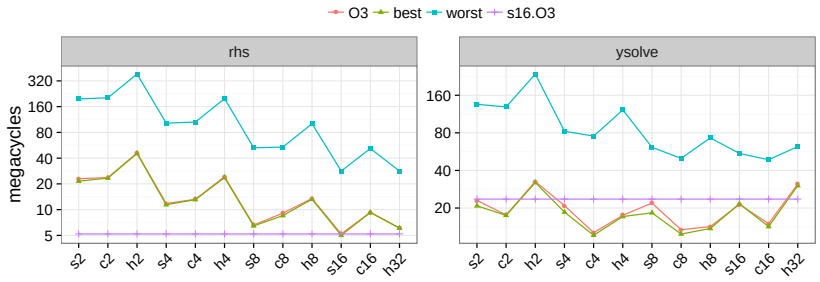


Fig. 1. Tuning exploration for two SP regions. For each affinity, we plot the best, worst, and `-O3` optimization sequences. Custom optimization beats `-O3` for `s2`,`s4`, and `s8` on `ysolve`.

We demonstrate how CERE operates on SP, a Scalar Penta-diagonal solver, from the C version of the NPB 3.0 OpenMP benchmarks [13]. CERE autotuning achieves a 1.82× performance speedup over the standard parameters levels. Thanks to the CERE codelet approach, the exploration time is approximately five times cheaper compared to the whole-program iterative compilation.

CERE starts by profiling SP and automatically selects representative OpenMP regions to tune. `xsolve`, `ysolve`, `zsolve`, and `rhs` are chosen and cover 93% of SP execution time. The coverage of a region is defined as the execution time of the region divided by the execution time of the whole application. CERE uses

integer linear programming to find the minimal codelet set for a target coverage. The approach is detailed by de Oliveira Castro et al. [12].

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.

This study is performed on Sandy Bridge. We explore the interactions between 12 thread configurations combining different number of threads and affinity mappings including scatter, compact, and hyperthread. Scatter distributes the threads as evenly as possible across the entire system. The opposite strategy, compact, assigns the threads to the cores as closely as possible. Hyperthread acts like compact but binds multiple threads to the same physical core to take advantage of virtual cores. We complete this study by evaluating 150 LLVM optimization sequences generated using the random sub-sampling presented in section 4. Combining all the parameters 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`.

CERE makes it possible, through codelet replay, to independently explore each region. Moreover, thanks to CERE replay prediction model presented in section 3.3, it is possible to quickly evaluate the impact of each configuration by using 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 explore 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.

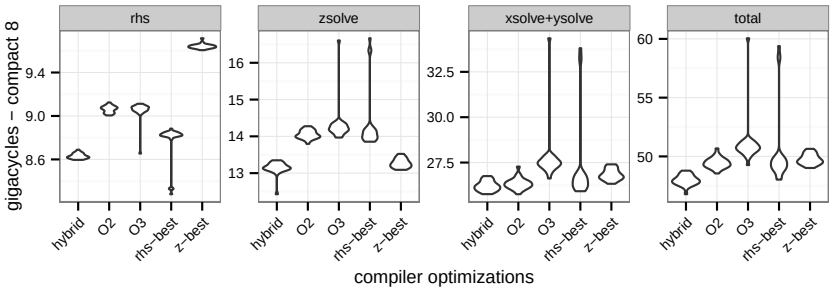


Fig. 2. Violin plot execution time of SP regions using best NUMA affinity. Measures were performed 31 times to ensure reproducibility. When measuring total execution time, hybrid compilation outperforms all other optimization levels, since each region uses the best available optimization sequence.

3 CERE AutoTuner

CERE [12, 13] 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.

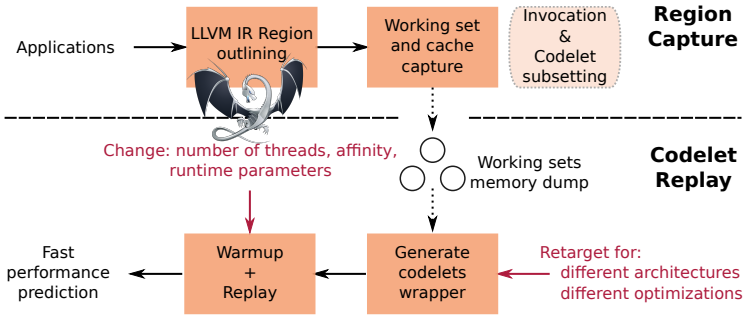


Fig. 3. Codelet capture and replay workflow

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. 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.

One of CERE strong features is allowing change of the number of threads during replay by extracting the `kmpc fork` calls [13]. CERE directly executes the function `kmpc fork` which decides how many threads are spawned. We can change the number of threads as long as the thread configuration is not con-

strained by the capture. A thread configuration is constrained by the capture if applications manually perform serial initializations to allocate memory slots depending on fixed values of number of threads. Fortunately, this design pattern is infrequent and was not used in the considered benchmarks.

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

3.1 NUMA Aware Warmup

A replay has to faithfully reproduce the original invocation context. CERE restores the memory working set of the region and warms up the cache to avoid cold-start bias [14]. 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 remapped to their original addresses. CERE includes two main cache warmup approaches [12] that restore the execution context before running the codelet.

The first is an optimistic warmup strategy that preloads the whole working set into the cache. The strategy assumes that the region working set was already in cache during the original execution. The second, is a page memory tracing technique. It warms the cache by replaying the memory access history at the memory page granularity. The first is faster to perform but less accurate than the second.

In this paper, we present a new orthogonal warmup approach for multi-threaded codelets executed on architectures with multiple NUMA domains. While changing the number of threads is easily done over our test applications, correctly mapping the pages across NUMA domains is a challenge.

Indeed, due to the node local first touch policy in Linux, a page is mapped to the thread, and therefore to its NUMA domain, which first attempts to use it. To guarantee that the codelet replay has the same behavior as the original region, we must ensure that pages are mapped to the same NUMA domains as they have been in the original run. The problem is that pages are not necessarily bound to the same NUMA domains across the different thread affinities. For instance, scatter runtime strategy maximizes the number of NUMA domains while compact minimizes it.

Figure 4 outlines this problem on a 2-NUMA domains 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 domain. Replays accurately predict the execution time as long as the affinity binds the threads to the same NUMA domain. Otherwise, the replay incurs 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 domain of the thread which is the first to touch them.

To ensure a correct NUMA mapping at replay, we must not exceed the number of threads at capture. Also, when the number of threads is changed, we must spread the pages across the replaying threads to correctly remap them across the NUMA domains. We spread the pages according to the following formula:

$$t'(p) = \lceil \frac{n'}{n} t(p) \rceil,$$

where p is the page we are trying to touch. n and n' are respectively the number of threads used during the capture and the replay. $t(p)$ is the thread that first accessed this page during the capture and $t'(p)$ is the thread that must touch the page at replay to correctly warm it up.

The most common NUMA policy is first touch, but there is also the less known interleave policy. Interleave is used at boot-time to avoid overloading the initial boot domain. Interleave evenly spreads the pages across different domains using round robin. It ensures that pages are evenly distributed across all NUMA domains, therefore making the NUMA latency uniform. When using interleave policy, CERE does not use any specific NUMA warmup. There is no guarantee that the pages are assigned to the same domains than in the original run, but the uniform distribution across domains is preserved.

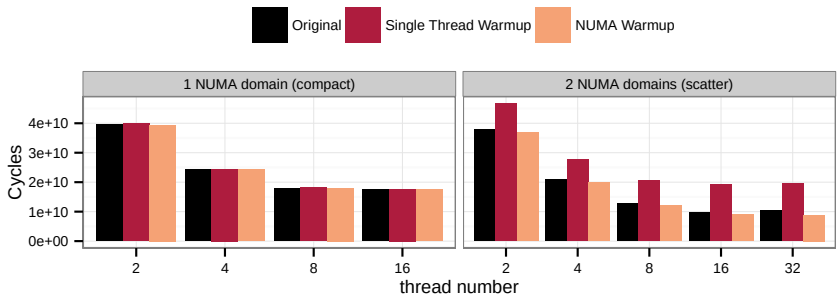


Fig. 4. Prediction accuracy of a single-threaded warmup versus a NUMA aware warmup on BT `xsolve`. Only a NUMA aware warmup is able to predict this region execution time on a multi NUMA domain configuration.

3.2 Multi-Process Capture with Ptrace

CERE captures the memory state at a page level granularity by only saving touched pages. After protecting the whole target application memory, the OS raises a signal when a memory page is touched. This signal is caught by CERE's handlers which unprotect and dump the memory page associated with the signal. Unfortunately, using the same process to execute an application and capture its memory can be problematic for multi-threaded runs.

The memory must not be modified while we attempt to protect it. In multi-threaded capture, threads have to be stopped before the tracing thread protects

memory pages. Otherwise CERE can attempt to protect a segment which is no longer valid. Indeed, a race condition exists: a thread may protect a page which has since been deallocated by another running thread. To avoid the issue the tracing thread should be able to stop all the other threads.

A first solution would be to send a `SIGSTOP` signal to the other threads. Yet, sending `SIGSTOP` to a process stops all its threads, including the thread which actually sent the signal. This will not work since the tracing thread will also be stopped hanging the capture. A second solution would be to use functionalities provided by the Oracle Solaris OS `thr_suspend` and `thr_continue` or Window OS `ResumeThread` and `SuspendThread` which allow stopping and restarting each individual thread. Unfortunately, the POSIX standard does not implement these functions. It is possible to simulate stop/restart functionalities by using a shared mutex, but this requires knowing the spawned threads in advance and modifying their code to include calls to the mutex. Since the CERE capture library does not make assumptions about the underlying program, we cannot apply this solution.

To address this challenge, we separate the studied application from the CERE capturing process. We use the `ptrace` mechanism. Ptrace is a system call which allows a process called `tracer` to monitor another process called `tracee`. Tracer can examine and change the tracee's memory and registers. To follow a thread, the tracer must attach it with ptrace. Since this command is per thread, the tracer must attach each thread of the tracee. (see block `Attach all threads` in Fig. 5). So the capturing process and the application respectively act as tracer and tracee.

When a signal is delivered to the tracee, the kernel stops the process and sends the signal to the tracer. The ptrace API provides a mechanism called `signal injection and suppression`: the tracer can choose to inject or suppress the signal. If the signal is injected, it is sent to the tracee. If the signal is suppressed, it is lost and the tracee remains stopped. We use this mechanism to capture the signal `SIGSEGV` raised when a thread touches a protected page. Protecting or unprotecting tracee's memory pages cannot be done from the tracer since a process can only modify its own memory.

The code for dumping and unprotecting a page is injected by the tracer in the tracee memory with ptrace. Then, the tracer resumes the tracee to execute the injected code, a `SIGTRAP` call at the end of the injected payload returns the focus to the tracer. (See block `Memory capture` Fig. 5)

Figure 5 details the new ptrace capture which is composed of four successive phases:

1. `Attach all threads:` the tracer attaches tracee threads with the ptrace attach command. Then it sends a SIGSTOP to each tracee to stop it. The tracer checks that the SIGSTOP has been received for each tracee. Once all the tracee threads are stopped the tracer is ready for the second phase.
2. `Memory Protection Mechanism:` the tracer protects the whole memory of the tracees by injecting a protecting assembly payload and restarts the threads. If a thread was already stopped before receiving the tracer SIGSTOP, the queued SIGSTOP signal must be cleared at restart to avoid a deadlock.

3. `NUMA first touch and page trace` starts once all the memory is protected. It captures the `tid` of the first thread to touch each page. It also keeps a trace of the most recently touched pages that is used to warmup the cache state at replay.
4. `Memory capture` starts when the region to capture is reached. CERE re-protects the whole memory and starts executing the region. It dumps all touched pages that are accessed until the region ends.

Figure 6 shows the performance comparison between the old single process capture and the new ptrace capture on the BT benchmark. BT is composed of four representative regions. For each region, CERE captures a representative invocation (see section 3.3). To evaluate the capture cost, we measured the execution time required to capture these invocations. On this benchmark, the ptrace based capture is 64.86× faster than the old memory capture. We note that to accelerate the capture process, CERE interrupts it as soon as the targeted invocation is executed. Since the full application is not executed, the capture process might be faster than the original execution.

The ptrace based capture speedup is achieved when the NUMA first touch capture is active. The old capture immediately re-protects each page because the whole memory must be protected when a codelet to capture is reached. This implies that the same page should be unprotected and protected several times during the NUMA capture phase, increasing the overhead. On the other side, the new NUMA capture pays the protection overhead only once, since it is able to protect the whole memory with multiple thread running thanks to its thread stopping strategy.

3.3 Piecewise Optimization with Codelets

Regions within an application are not sensitive to the same optimizations: SP `rhs` and `zsolve` regions in section 2 from the motivating example 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. 12) or regions. IS illustrates how codelets can be use to focus the tuning over a specific region. IS is composed of two regions that respectively generates a random sequence of numbers and sorts them. The benchmark purpose is to measure the sorting time. Nevertheless, the random number generation represents 60% of the application execution time. Through a codelet, CERE extracts the sorting region and tunes it without executing the rest of the application, thus avoiding the initialization overhead.

Codelets also accelerate the tuning process for each region. Regions also have performance variations across different invocations. By using the CLustering LARge Applications algorithm [15] (CLARA), CERE groups the invocations sharing a similar execution time into clusters that capture different performance

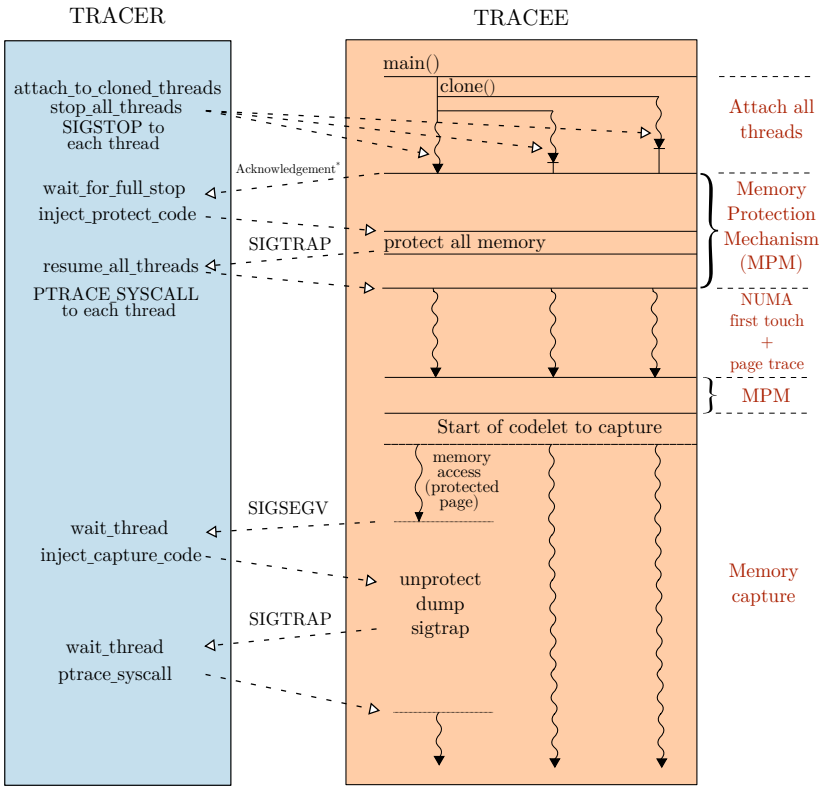


Fig. 5. Multithreaded capture mechanism with ptrace. Capture is split into two distinct processes : the CERE capturing method as the tracer the studied application as the tracee.

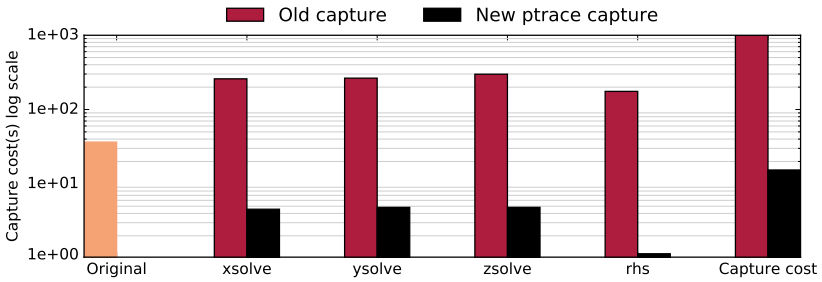


Fig. 6. Speed comparison between of old memory capture versus new memory capture on BT. We used 8 threads on 4 physical i7-4470 cores. The original execution column corresponds to the execution time of BT. xsolve, ysolve, zsolve, and rhs represent the execution time required to capture the first invocation of these regions with either the old or the new capture. The last column sums the capture cost of all the regions for the two strategies. Capturing BT memory context is faster than executing the whole application because we interrupt the execution of each capture as soon as the invocation of interest is executed.

classes. Once the performance classes are identified, CERE selects one representative invocation per class. The representative is the invocation that is closest to the median performance of all the invocations inside the cluster.

CERE accelerate the tuning process by only replaying representative invocations. In particular, CERE extrapolates the full region performance by summing the contribution of each performance class. The contribution of a performance class is defined as its representative invocation execution time multiplied by the number of invocations within that class. Figure 7 demonstrates how CERE replays the parallel region `Resid` MG executed with four threads and compiled with `-O0`. CERE clusters the 42 invocations into 3 performance classes (top left) and selects a representative invocation per performance class to replay. By only replaying these three invocations (top right), CERE can extrapolate the execution time of the whole region.

Figure 7 also presents the same region compiled with `-O3` and executed with two threads. We observe that invocations remain in the same performance classes (down left). Therefore, changing a parameter has the same impact over all the invocations within the same performance class. This is the fundamental assumption of CERE execution time prediction. So, by replaying the same three representative invocations (down right), CERE predicts the region execution for this new configuration.

Figure 8 shows how CERE applies this methodology to tune a region across different compiler optimizations. `ysolve` has 400 invocations with a similar execution time that CERE detects as a single performance class. Since CERE only executes one representative invocation, tuning the region is 149× cheaper with a codelet than running the full SP benchmark.

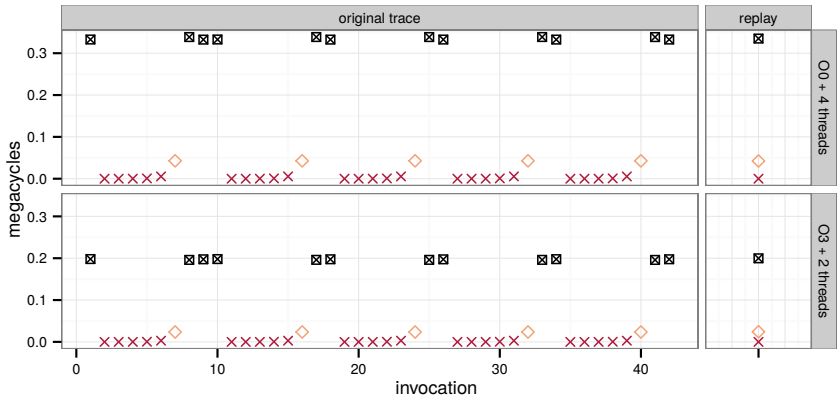


Fig. 7. MG `resid` invocations execution time on Sandy Bridge over `-O3` and `-O0` with respectively 2 and 4 threads. Each representative invocation predicts its performance class execution time.

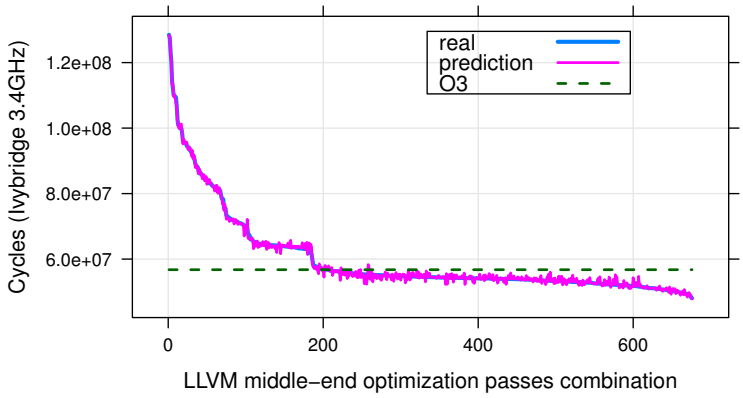


Fig. 8. NAS SP `ysolve` region execution time across 1000 schedules of random compiler combinations passes based on `O3`. Compilation sequences are sorted according to their original execution time. We remove compilation sequences that produce the same binary. **`real`** and **`predicted`** respectively represents the original execution time and the codelet prediction based on representative invocations replay. The codelet faithfully predicts the region execution time across the different compiler optimizations.

3.4 Finding Common Optimization Sequences in Codelet Clusters

The codelet search can also take advantage of similar and repeated computation patterns within the applications to accelerate the tuning. For instance, two linear algebra solvers, despite using different algorithms, will share common computation patterns such as vector copy loops, dot product computations, or matrix vector multiplications. We expect that these similar patterns are impacted in the same way by similar compiler optimizations [16, 11].

CERE takes advantage of codes with such similar patterns by extracting them as codelets. Instead of evaluating an optimization across all the codes, CERE evaluates the optimization once with a selected codelet. Then, it extrapolates its impact over the other similar codelets. Figure 9 illustrates how clustering codelets accelerates the evaluation process of new optimizations.

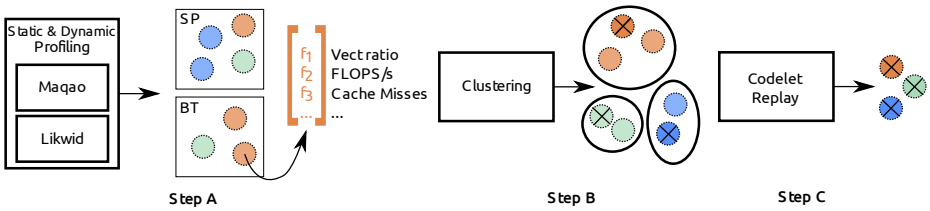


Fig. 9. Clustering codelets for fast compiler optimization tuning. Step A: perform static and dynamic analysis to capture codelets metrics vectors. Step B: by using the proximity between metrics vectors, CERE clusters similar codelets and selects one representative per cluster. Step C: CERE extracts and replays the representatives as standalone codelets.

To detect similar computation patterns, CERE relies on an hybrid approach combining both static and dynamic performance metrics with clustering. First, we profile each region that is a codelet candidate for extraction. We extract the static metrics with MAQAO Code Quality Analyzer [17]. The dynamic metrics are provided by reading the hardware performance counters with Likwid [18].

CERE combines the performance metrics into vectors that are associated to each region. These metric vectors are used as performance signatures to detect similar regions. MAQAO and Likwid provide over 80 different metrics. Irrelevant metrics add noise that degrades CERE predictions. Grouping codelets that are not sensitive to the same optimizations causes CERE to miss-predict the impact of a compiler optimization. Therefore, it is necessary to wisely select metrics, keeping only those that adequately capture the program behavior. Table 2 presents the performance metrics used by CERE while section 4.4 explains how they were selected.

Once a metric vector is associated to each region, we group the regions sharing similar vectors into clusters. In particular, CERE computes euclidean distances across the vectors and uses the hierarchical clustering with Ward's criterion [19]. The final number of clusters is selected with the Elbow method [20].

MAQAO static metrics	Likwid dynamic metrics
- Bytes stored per cycle assuming L1 hits	- Floating point rate in MFLOPS.s^{-1}
- Data dependencies stalls	- L2 bandwidth in MB.s^{-1}
- Estimated IPC assuming only L1 hits	- L3 miss rate
- Number of SD instructions	- Memory bandwidth in MB.s^{-1}
- Pressure in dispatch port P1	
- Ratio between ADD+SUB/MUL	
- Vectorization ratio for Multiplications (FP +INT)	
- Vectorization ratio for Other (FP)	
- Vectorization ratio for Other (INT)	

Table 2. Performance metric set used to cluster regions.

Finally, CERE selects a representative per cluster and extract it as a codelet. A representative must adequately capture the performance metrics of the cluster: we choose the codelet closest to the cluster centroid. The centroid reduces the subsetting approximations since it is the closest point to the cluster average values.

Codelets from the same cluster share the same metrics and should react in the same way to compiler changes. Therefore, by measuring a single representative per cluster, we can extrapolate the performance of all its siblings. This method has been validated for architecture selection by de Oliveira Castro et al. [21] and we extend it to compiler optimizations in this paper. In particular, we demonstrate that codelets from the same cluster are sensitive to similar compiler optimizations in section 4.4.

3.5 Hybrid Compilation

As stated in section 3.3, CERE tunes each codelet separately to outperform monolithic approaches through hybridization. Figure 10 presents how the hybrid tuning operates in two phases. First, the piecewise tuning finds the best compiler optimizations for each loop and OpenMP region. Second, the best found optimizations are applied to each 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 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.

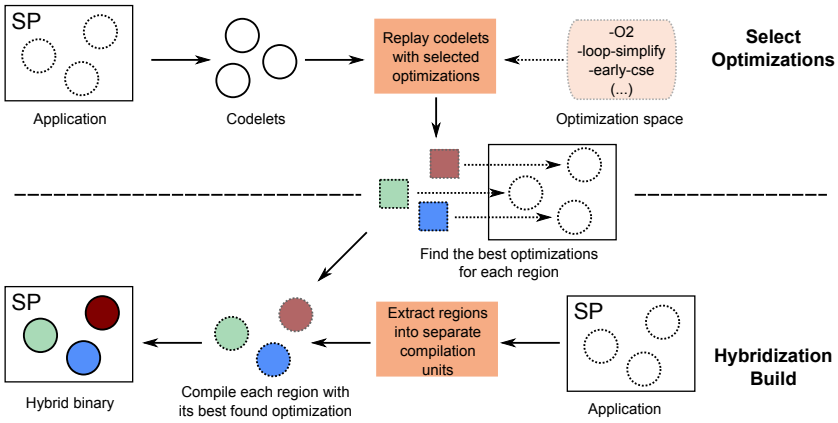


Fig. 10. Hybrid compilation process. Through codelets, CERE finds the best compiler sequence for each region. Then, each region is compiled with its best compiler sequence in the original application.

A limitation of the approach is that an extracted region in a new compilation unit may not be sensitive to the same optimizations. Since the compilation context is different in the new compilation unit, the same compiler optimization can generate different codes between the original and the extracted region. Nevertheless, in our experiments, this did not happen.

Another limitation is that our approach does not model the interactions between regions [22, 23]. In this paper, we assume that we can apply compiler optimizations to each region separately without impacting the others. Detecting when two regions can be separately optimized without impacting each other is a challenge. Our intuition is that regions that are far from each other during the execution should have few interactions.

4 Thread and Compiler Parameters Tuning Validation

This section validates the usage of codelets as proxies to tune compiler and runtime parameters. Codelets capture most of the application hotspots [12]. This section shows 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. So, we quantify the similarity between codelet and application executions. We also evaluate CERE speedup and tuning benefits over standard monolithic approaches.

4.1 Experimental Setup

We used two different Intel CPU micro-architectures for this validation: 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. On the other side, the Ivy Bridge was used for the compiler exploration.

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 [24] notation to characterize the thread placement. Cores ranked between 0 and 7 reference the physical cores of the first NUMA domain while cores between 8 and 15 reference the physical cores of the second NUMA domain. Similarly, cores from 16 to 23 and from 24 to 31 reference the hyper-threaded cores of respectively the first and the second NUMA domain.

The exploration was performed on LLVM 3.4 using a random pass selection. We use LLVM opt and llc to respectively change 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 [25] and C OpenMP parallel [26] benchmarks (respectively NAS SER and NPB) with CLASS A datasets and on a Reverse Time Migration [27] (RTM) proto-application. Since we execute both the original and the codelet versions for each region, we selected CLASS A datasets to perform more experiments in a reasonable amount of time. To improve the accuracy, we use the CERE page tracing strategy to warmup the codelets used for serial compiler passes tuning.

4.2 Number of Threads and Affinity Tuning

This section presents the thread affinity tuning results. CERE page memory capture was performed on a `16 threads scatter` run. Table 3 evaluates CERE thread affinities replay accuracy and reduction factor over NAS OpenMP regions.

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 a realistic warmup that replays each representative invocation four times to restore its execution context. We performed four meta-repetitions to ensure performance stability and accuracy. These repetitions are not amortized on EP and MG.

EP is composed of a big parallel region executed once in the original execution. The region covers the whole application execution time: there are no other invocations or regions in EP that CERE can avoid to accelerate the tuning.

MG regions have multiple performance variations across their different invocations. To accurately predict each region execution time, CERE replays multiple representative invocations which slows down the tuning.

Also, we note that as we increase the data sets, the warmup 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. 11). CERE mispredict some thread configurations for CG and MG. These issues only appear for high number of threads configurations. 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.

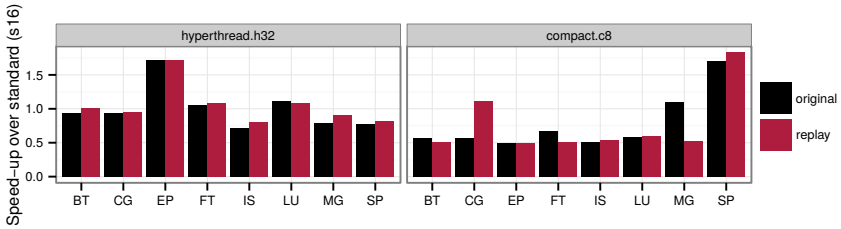


Fig. 11. Original and CERE predicted speedup for two thread configurations. Replay speedup is the ratio between the replayed target and the replayed standard configuration. CERE accurately predicts the best thread affinities in six out of eight benchmarks. For CG and MG, we miss-predict configurations that use all the physical cores.

4.3 Compiler Passes Tuning and Hybridization

Table 3 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 12 presents the number of explored compiler sequences required 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

Benchmarks	Compiler passes #Regions	Accuracy (%)	Reduction factor	Thread affinity #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 3. 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.

be improved with more sophisticated techniques for passes selection such as genetic algorithms [7] which would also benefit from the piecewise approach.

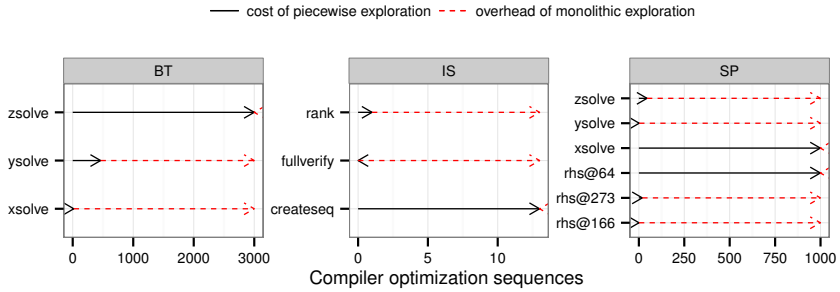


Fig. 12. Compiler sequences required to get a speedup over 1.04× per region. CERE evaluates the sequences in the same order for all the regions. Exploring regions separately is cheaper because we stop tuning a region as soon as the speedup is reached.

CERE outperforms the standard `-O3` over `BT`, `SP`, and `IS` with an average speedup of 1.06× (see Fig. 13). `IS` random generator and sorting algorithm do not benefit from the same optimizations which explains the significant difference between the hybrid and the monolithic approaches. Hybrid binaries based on original or replay explorations have the same performances which ensure that we do not miss any optimizations through the codelets.

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`. The two regions are executed multiple times one after the other. Since each region is compiled in its compilation unit, there is no code generation interaction between them. So, we suspect some memory interactions.

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. Detailed accuracy and reduction factor reports over the NAS benchmarks are available at `https://benchmark-subsetting.github.io/autotuning-results/`.

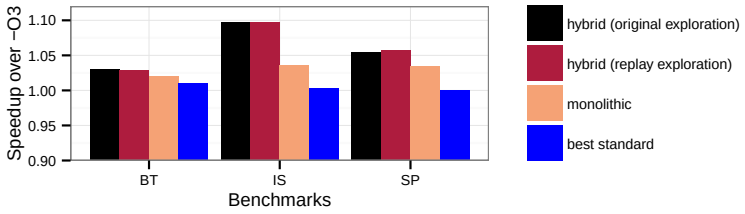


Fig. 13. Speedups over `-O3`. We only observe speedups from the iterative search over `BT`, `SP`, and `IS`. Best standard is the more efficient default optimization (either `-O1`, `-O2`, or `-O3`). Monolithic is best whole program sequence optimization. Hybrid binaries are build upon optimizations found either with codelets or with original application runs.

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 [10]. 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`.

4.4 Codelets Clustering

This section presents how codelets clustering can be used to further accelerate the tuning process. Codelets from the same cluster share the same metrics and should react in the same way to compiler changes. Therefore, by measuring a single representative per cluster, we can extrapolate the performance of all its siblings.

Figure 14 presents the impact of the best found compilation sequence for each region applied over the other regions. We consider the NAS clusters that were originally designed for architecture selection [21]. We expect that the best optimization of a codelet within a cluster will also benefit the other codelets within that cluster. Except for clusters `C3` and `C4`, the white squares show that in general the best optimization for a codelet also benefits the other codelets in that cluster. Each cluster contains between two or three codelets. Since a single

codelet per cluster is replayed, this approach provides an average additional speedup of 2.28× over the tuning. Codelets clustering is attractive when we use a lot of programs because executing multiple applications increases the chances of introducing redundancies. Therefore, this speedup can be further increased by considering more applications.

It is interesting to notice that this clustering was originally designed to catch architectural changes [21]. Applications were decomposed into codelets and profiled over a Nehalem `L5609`. Instead of executing whole applications on new architectures to evaluate them, only representative codelets were measured. Yet and as demonstrated by Fig. 14, the architectural clustering remains suitable to gather codes according to their compiler optimization sensitivity.

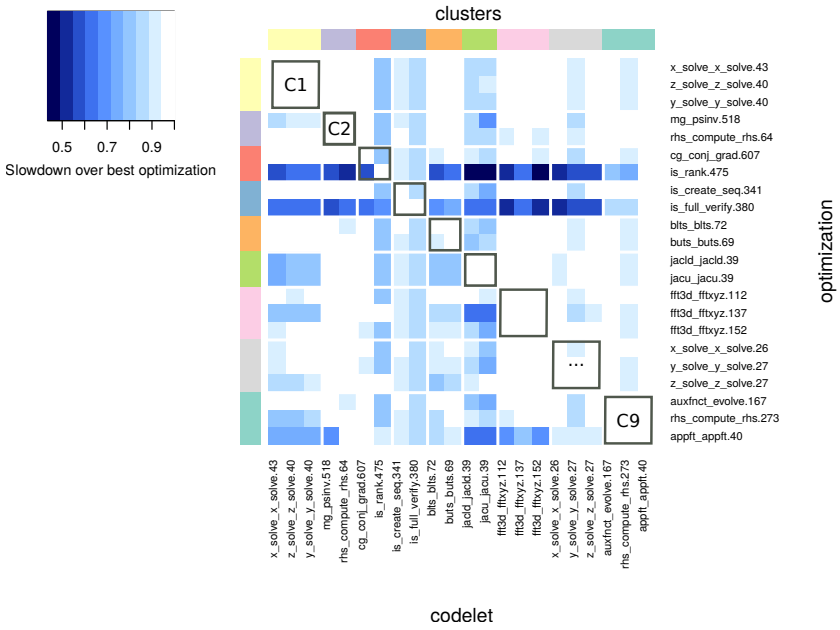


Fig. 14. Optimization sequence evaluation across clusters of similar codelets. To build the color matrix, we apply to each codelet (horizontal axis) the best optimization sequence for every other codelet (vertical axis). The color of each cell represents the slowdown over the best sequence found. The square `C1` represents the compilation passes of the first cluster applied to the regions within `C1`.

5 Exploring runtime parameters in heterogeneous architectures

Tuning through CERE codelets is flexible and can be extended to new domains. In this section we consider the problem of mapping and tuning a parallel applica-

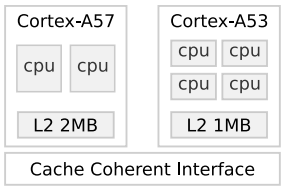


Fig. 15. Juno big.LITTLE ARM architecture

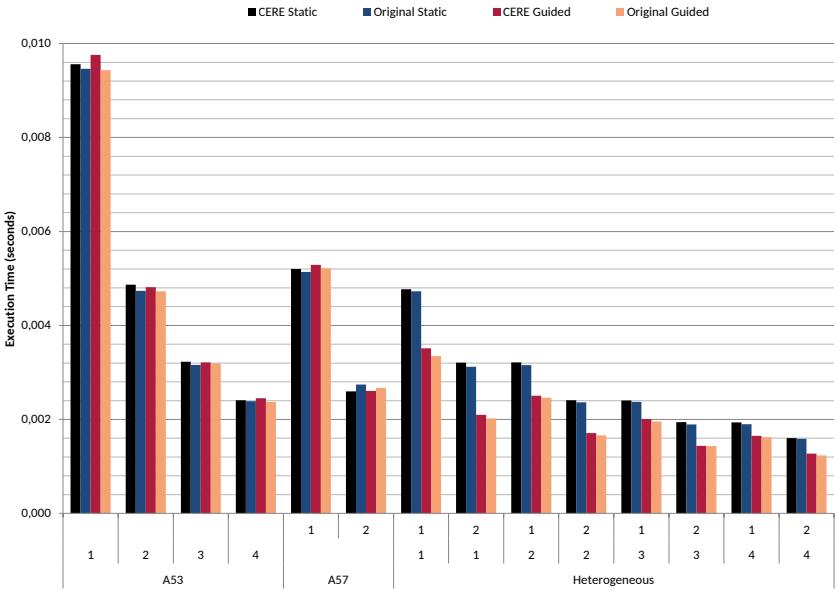


Fig. 16. Execution time of PARSEC Blackscholes 64K in a Juno board. The horizontal axis shows the thread mapping across the four A53 and two A57 cores. The numbers in the first (respectively second) line are the number of threads mapped on A57 (respectively A53) cores. The first two categories consider homogeneous mappings and the last category considers heterogeneous mappings. For each mapping, we both consider static and guided scheduling strategies. For each scheduling, we validate the time estimated by CERE to the time measured in the original benchmark.

tion over an heterogeneous architecture such as the ARM big.LITTLE Juno [28] development board. The architecture, as shown in Fig. 15, combines two *clusters*: one *big* dual-core A57 and one *little* A53 quad-core. Each cluster has its own L2 cache but they can share data through a cache-coherency interface.

Mapping a parallel application to this architecture is challenging because of the compute imbalance between the A53 and A57 and the difficulty to estimate the communication cost between clusters. Through CERE codelets, one can quickly test different mapping and scheduling strategies to find the best configuration. We demonstrate this by exploring PARSEC Blackscholes configurations on the Juno board. In this study, we focus on execution time and do not consider energy measurements.

PARSEC Blackscholes computes option pricing by solving a Partial Differential Equation. PARSEC OpenMP implementation is embarrassingly parallel except for the data initialization phase. We use CERE to capture the main parallel region found at Blackscholes.m4.cpp line 368. To achieve this we had to port CERE to AArch64; which required multiple changes in the capture library[4].

Once codelets were extracted, we systematically explored the available thread affinities. The horizontal axis of Figure 16 shows the fourteen considered mappings: the first four are homogeneous executions on the A53; the next two are homogeneous executions on the A57; and the final eight are heterogeneous mappings. We used the OpenMP default static scheduling strategy which divides loops iterations into equally sized chunks across the different threads. The time of each execution was measured using the `cntvct_el0` cycles register [29]. To ensure that CERE estimates were correct, we validated each run execution time against the execution time of the original benchmark.

Running Blackscholes on the fourteen configurations took 1.42 seconds when using the CERE codelet and 60.53 seconds when running the original benchmark. Despite this significant speedup, the CERE estimates are very accurate across all configurations.

When an homogeneous cluster is used, either A53 or A57, we see that Blackscholes linearly scales as expected from an embarrassingly parallel benchmark. On the other side, performances on heterogeneous configurations are limited by the work imbalance. Let us consider heterogeneous mappings with three threads. We observe similar performance when using two A53 cores or two A57 cores. Since the workload is equally divided across the different cores, performances are limited by the workload running on the A53 cluster. Similarly, A53 cores limit performance across the other heterogeneous mappings with two and four threads.

To take advantage of the A53 cores, we propose to switch the scheduling policy for loop iterations from static to the OpenMP guided. Instead of equally dividing the loop iterations across the cores, the guided policy considers a work queue of loop iterations grouped into chunks. When a thread finishes his chunk, it retrieves the next chunk from the top of the queue. While this OpenMP

[4] The AArch64 Linux ABI obsoletes some system calls and we had to rewrite some architecture-specific sections of code.

policy improves work-balancing by considering the target processors, it also introduces an overhead. To partially reduce this overhead while preserving work-balancing, guided starts with large chunks and reduces them through the execution. Figure 16 demonstrates the benefits of the guided scheduling over the default OpenMP policy. Using guided scheduling achieves a speedup of 1.29× compared to the default policy over the best mapping strategy. CERE remains faithful to the original executions across the different guided mappings while quickly finding the best scheduling strategy.

This preliminary study on the big.LITTLE ARM architecture demonstrates that CERE autotuning capabilities can easily be applied to problems involving heterogeneous architectures by simultaneously considering different number of threads, thread mapping strategies, and scheduling polices. As future work, we can test applications that require to change the OpenMP parameters for each specific region.

6 Related Work

We classify our work into three categories: design space exploration, fine grained tuning with application reduction, and code isolation. They all share a common objective: improving the tuning process of applications.

6.1 Design Space Exploration

A common method to accelerate the compiler tuning process is to guide the search exploration though machine learning techniques such as Genetic Algorithms (GA) [7, 9, 8]. Cooper et al. [8] applied GA for compiler exploration on embedded applications. In particular, they show that GA more quickly converges over satisfying results than random selection for code reduction. Host et al. [9] extend the usage of the GA through Pareto frontiers to target multi-objective compiler exploration such as reducing the code size, the execution time or the compilation time.

An additional approach to guide the parameter exploration is through code clustering. As discussed in section 3.4, similar regions of code or applications can be clustered together to accelerate the tuning. The idea is to associate the best compiler optimization to each cluster. Common GA with iterative compilation is usually used to find these best optimizations. To optimize a new application, we classify it among the clusters and associate the compiler optimization of the selected cluster: we avoid the evaluation of the whole parameter space. The challenge of these approaches is to select the metrics for the clustering: they must actually gather codes that are both sensitive to the same optimizations.

There are different propositions for the clustering criteria. Fursin et al. [11] with Milepost GCC propose static performance metrics to perform the clustering. Martins et al. [30] quantify the similarity between functions through data mining applied to a symbolic representation of the code. Ashouri et al. [31] extend these metrics by combining static and dynamic architecture independent metrics into an hybrid characterization.

Tuning compiler optimizations through codelet replays is an orthogonal approach to these tuning strategies. Codelets do not focus on the search space but rather accelerate the evaluation of each exploration point. Nevertheless, section 4.4 shows how codelets can be clustered with hybrid metrics to additionally prune the exploration space.

Instead of focusing on the code properties, other methods target the compiler optimizations in order to also reduce the number of evaluations. For example, some studies [32, 33] statically determine the impact of some compiler optimizations. Purini et al. [34] propose to search general sets of compilation sequences instead of a unique sequence. Through LLVM iterative compilation runs, they find general optimization sequences that should work well on any given program. CERE codelets could be used as proxies to quickly tune the good optimizations set instead of exploring all the optimizations.

6.2 Application Reduction and Fine Grained Tuning

Applications execution change over time in ways that are often structured as sequences of a small number of reoccurring behaviors, and which are called phases [35]. Each interval of execution labeled as a phase is expected to yield some distinct execution properties, e.g. performance metrics, compared to the other phases. Popular methods to define such phases include basic block vectors [36], performance metrics [35], or parallel synchronizations [37].

Usual benchmark reduction techniques take advantage of these phases to reduce the simulation cost [36]. They cannot be directly used for compiler tuning as they operate on the assembly. Fursin et al. [38] take advantage of the application phases by producing multiple versions of the same region where each version is associated to a compiler sequence to evaluate. When the application is executed, each invocation may use a specific compiler optimization, allowing the evaluation of multiple sequences with a single run. However, they do not use any code isolation techniques so they cannot focus the search as we demonstrate in Fig. 12. This is problematic when a region of interest has a few invocations compared to the others. Moreover their cache warmup system is limited because it does not use a memory trace and is based on the execution of previous iterations. Similarly, Kulkarni et al. [39] 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.

One limitation of our work is that we ignore the adjacent regions dependencies regarding the effect of optimizations. To address this, the work of Curtis et al. [23] proposes an exploration technique to avoid neighboring optimizations with negative cache effects.

Another point to consider is the data set sensitivity of the hybrid compilation. Fine grained optimization increases the performance benefits over monolithic tuning and therefore the chances of optimizing the code for a specific data set. Chen et al. [22] show that increasing the number of trained data sets increases

the overall performance on multiple applications. As future work, we can test the hybridization across different data sets.

The main difference between these related works and CERE is that CERE relies on code isolation. Unlike whole application tuners, extracted codelets can directly target and optimize the representative phases without execution constraints.

6.3 Code Isolation

Multiple approaches have been propose for sequential code isolation [12], but to the best of our knowledge, only Liao et al. [40] extract tunable kernels out of parallel OpenMP programs. They isolate parallel `for` loops at source level. Unfortunately, some transformations used during source isolation may hinder compiler optimization passes [41]. Moreover, outlining a source loop from a parallel region removes the loop from the lexical extent of the parallel region and alters the semantics of the program because the scope of OpenMP data clauses (private, shared, or reduction) is lost. The source outlining approach requires an additional step that repairs the lost scopes. On the contrary, CERE IR level outlining is simpler because it is done after OpenMP data clauses expansion.

7 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, heterogeneous architectures and compiler optimization passes. CERE proposes a novel piecewise approach that accelerates exploring the parameter space and enables an hybrid compilation where each region uses the best set of local parameters. It outperforms traditional monolithic tuning. Using a clustering approach, we show that it is possible to further reduce the exploration cost by applying the best optimization found for a codelet to other similar codelets in the same cluster.

CERE predicts the impact of thread placement and compiler optimization with a mean accuracy of 94.7% over the NAS 3.0 benchmarks. On an RTM proto-application, CERE achieves a 1.11× execution speedup through compiler pass selection. The search is 200× faster thanks to codelet tuning. Finally, tuning Blackscholes thread configurations on an heterogeneous architecture was 42× faster with codelets than running the whole application.

The Ptrace based NUMA capture and the hybrid compiler are released in CERE v0.2.2 which is available at `https://github.com/benchmark-subsetting/cere/`.

Acknowledgments: The authors thank the anonymous reviewers for their feedback. 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. Popov, M., Akel, C., Jalby, W., Castro, P.d.O.: Piecewise holistic autotuning of compiler and runtime parameters. In Kaklamanis, C., Papatheodorou, T.S., Spirakis, P.G., eds.: Euro-Par 2016 Parallel Processing - 22nd International Conference. Volume 9833 of Lecture Notes in Computer Science., Springer (2016) 238–250
2. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: Code Generation and Optimization, 2004. CGO 2004. International Symposium on, IEEE (2004) 75–86
3. Kisuki, T., Knijnenburg, P., OBoyle, M., Wijshoff, H.: Iterative compilation in program optimization. In: Proc. CPC10 (Compilers for Parallel Computers). (2000) 35–44
4. Mazouz, A., Barthou, D., et al.: 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), 2011 International Conference on, IEEE (2011) 273–279
5. 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 23rd international conference on Supercomputing, ACM (2009) 460–469
6. 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
7. Ladd, S.R.: Acovea: Analysis of compiler options via evolutionary algorithm (2007)
8. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: ACM SIGPLAN Notices. Volume 34., ACM (1999) 1–9
9. Hoste, K., Eeckhout, L.: Cole: compiler optimization level exploration. In: Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization, ACM (2008) 165–174
10. de Oliveira Castro, P., Petit, E., Farjallah, A., Jalby, W.: Adaptive sampling for performance characterization of application kernels. Concurrency and Computation: Practice and Experience **25**(17) (2013) 2345–2362
11. Fursin, G., et al.: Milepost gcc: Machine learning enabled self-tuning compiler. International Journal of Parallel Programming **39**(3) (2011) 296–327
12. De Oliveira Castro, P., Akel, C., Petit, E., Popov, M., Jalby, W.: Cere: Llvm-based codelet extractor and replayer for piecewise benchmarking and optimization. ACM Transactions on Architecture and Code Optimization (TACO) **12**(1) (2015) 6
13. Popov, M., Akel, C., Conti, F., Jalby, W., de Oliveira Castro, P.: Pcere: Fine-grained parallel benchmark decomposition for scalability prediction. In: Parallel and Distributed Processing Symposium (IPDPS), 2015 IEEE International, IEEE (2015) 1151–1160
14. Kessler, R.E., Hill, M.D., Wood, D.A.: A comparison of trace-sampling techniques for multi-megabyte caches. IEEE Transactions on Computers **43**(6) (1994) 664–675
15. Kaufman, L., Rousseeuw, P.J.: Finding groups in data: an introduction to cluster analysis. Volume 344. John Wiley & Sons (2009)
16. Agakov, F., Bonilla, E., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M.F., Thomson, J., Toussaint, M., Williams, C.K.: Using machine learning to focus iterative optimization. In: Proceedings of the International Symposium on Code Generation and Optimization, IEEE Computer Society (2006) 295–305

17. Charif-Rubial, A.S., Oseret, E., Noudohouenou, J., Jalby, W., Lartigue, G.: Cqa: A code quality analyzer tool at binary level. In: High Performance Computing (HiPC), 2014 21st International Conference on, IEEE (2014) 1–10
18. Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: Parallel Processing Workshops (ICPPW), 2010 39th International Conference on, IEEE (2010) 207–216
19. Ward, J.H.: Hierarchical grouping to optimize an objective function. Journal of the American Statistical Association **58**(301) (1963) 236–244
20. Thorndike, R.: Who belongs in the family? Psychometrika **18**(4) (1953) 267–276
21. de Oliveira Castro, P., Kashnikov, Y., Akel, C., Popov, M., Jalby, W.: Fine-grained benchmark subsetting for system selection. In: Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization, ACM (2014) 132
22. Chen, Y., Huang, Y., Eeckhout, L., Fursin, G., Peng, L., Temam, O., Wu, C.: Evaluating iterative optimization across 1000 data sets. In: Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10), Toronto, Canada (2010)
23. Curtis-Maury, M., Blagojevic, F., Antonopoulos, C.D., Nikolopoulos, D.S.: Prediction-based power-performance adaptation of multithreaded scientific codes. IEEE Transactions on Parallel and Distributed Systems **19**(10) (2008) 1396–1410
24. Intel: Reference Guide for the Intel(R) C++ Compiler 15.0. `https://software.intel.com/en-us/node/522691`
25. Bailey, D., et al.: The NAS parallel benchmarks summary and preliminary results. In: Proceedings of the conference on Supercomputing, ACM/IEEE (1991) 158–165
26. Popov, M.: NAS 3.0 C OpenMP. `http://benchmark-subsetting.github.io/cNPB`
27. Baysal, E.: Reverse time migration. Geophysics **48**(11) (November 1983) 1514
28. ARM: Juno ARM Development Platform Datasheet (2014)
29. ARM: Arm cortex-a57 mpcore processor technical reference manual revision r1p1 (2016)
30. Martins, L.G., Nobre, R., Cardoso, J.M., Delbem, A.C., Marques, E.: Clustering-based selection for the exploration of compiler optimization sequences. ACM Transactions on Architecture and Code Optimization (TACO) **13**(1) (2016) 8
31. Ashouri, A.H., Mariani, G., Palermo, G., Park, E., Cavazos, J., Silvano, C.: Cobayn: Compiler autotuning framework using bayesian networks. ACM Transactions on Architecture and Code Optimization (TACO) **13**(2) (2016) 21
32. Kulkarni, P.A., Whalley, D.B., Tyson, G.S., Davidson, J.W.: In search of near-optimal optimization phase orderings. In: ACM SIGPLAN Notices. Volume 41., ACM (2006) 83–92
33. Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S., Subramanian, D., Torczon, L., Waterman, T.: Acme: adaptive compilation made efficient. In: ACM SIGPLAN Notices. Volume 40., ACM (2005) 69–77
34. Purini, S., Jain, L.: Finding good optimization sequences covering program space. ACM Transactions on Architecture and Code Optimization (TACO) **9**(4) (2013) 56
35. Eeckhout, L., Sampson, J., Calder, B.: Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In: Workload Characterization Symposium, 2005. Proceedings of the IEEE International, IEEE (2005) 2–12
36. 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, 2001. Proceedings. 2001 International Conference on, IEEE (2001) 3–14

37. Carlson, T.E., Heirman, W., Van Craeynest, K., Eeckhout, L.: Barrierpoint: Sampled simulation of multi-threaded applications. In: Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS). (2014)
38. Fursin, G., Cohen, A., OBoyle, M., Temam, O.: Quick and practical run-time evaluation of multiple program optimizations. In: Transactions on High-Performance Embedded Architectures and Compilers I. Springer (2007) 34–53
39. Kulkarni, P.A., Jantz, M.R., Whalley, D.B.: Improving both the performance benefits and speed of optimization phase sequence searches. In: ACM Sigplan Notices. Volume 45., ACM (2010) 95–104
40. Liao, C., Quinlan, D.J., Vuduc, R., Panas, T.: Effective source-to-source outlining to support whole program empirical optimization. In: Languages and Compilers for Parallel Computing. Springer (2010) 308–322
41. Akel, C., Kashnikov, Y., de Oliveira Castro, P., Jalby, W.: Is Source-code Isolation Viable for Performance Characterization? In: International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI), IEEE Computer Society (2013)