CERE: LLVM-Based Codelet Extractor and REplayer for Piecewise Benchmarking and Optimization

PABLO DE OLIVEIRA CASTRO, Université de Versailles Saint-Quentin-en-Yvelines and Exascale Computing Research
CHADI AKEL, Exascale Computing Research
ERIC PETIT and MIHAIL POPOV, Université de Versailles Saint-Quentin-en-Yvelines
WILLIAM JALBY, Exascale Computing Research

This article presents Codelet Extractor and REplayer (CERE), an open-source framework for code isolation. CERE finds and extracts the hotspots of an application as isolated fragments of code, called *codelets*. Codelets can be modified, compiled, run, and measured independently from the original application. Code isolation reduces benchmarking cost and allows piecewise optimization of an application. Unlike previous approaches, CERE isolates codes at the compiler Intermediate Representation (IR) level. Therefore CERE is language agnostic and supports many input languages such as C, C++, Fortran, and D. CERE automatically detects codelets invocations that have the same performance behavior. Then, it selects a reduced set of representative codelets and invocations, much faster to replay, which still captures accurately the original application. In addition, CERE supports recompiling and retargeting the extracted codelets. Therefore, CERE can be used for cross-architecture performance prediction or piecewise code optimization. On the SPEC 2006 FP benchmarks, CERE codelets cover 90.9% and accurately replay 66.3% of the execution time. We use CERE codelets in a realistic study to evaluate three different architectures on the NAS benchmarks. CERE accurately estimates each architecture performance and is 7.3× to 46.6× cheaper than running the full benchmark.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Measurement Techniques, Modeling Techniques; B.8.2 [**Performance and Reliability**]: Performance Analysis and Design Aids

General Terms: Performance, Measurement

Additional Key Words and Phrases: Program replay, checkpoint restart, iterative optimization, performance prediction

ACM Reference Format:
Pablo de Oliveira Castro, Chadi Akel, Eric Petit, Mihail Popov, and William Jalby. 2015. CERE: LLVM-based codelet extractor and REplayer for piecewise benchmarking and optimization. ACM Trans. Architec. Code Optim. 12, 1, Article 6 (April 2015), 24 pages.
DOI: http://dx.doi.org/10.1145/2724717

1. INTRODUCTION

Performance evaluation for optimization, system benchmarking, or compiler evaluation is time and resource consuming. The expensive cost limits the number of iterations

New paper, not an extension of a conference paper. This work has been conducted in part in the Exascale Computing Research laboratory, thanks to the support of CEA, GENCI, Intel, and UVSQ; and in part in the ITEA2 Coloc project. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the CEA, GENCI, Intel, or UVSQ.
Authors' addresses: P. de Oliveira Castro, C. Akel, E. Petit, M. Popov, and W. Jalby, Université de Versailles Saint-Quentin-en-Yvelines, 78035 Versailles, France; emails: {pablo.oliveira, eric.petit, mihail.popov}@uvsq.fr, {chadi.akel, william.jalby}@exascale-computing.eu.

DOI: http://dx.doi.org/10.1145/2724717

engineers can perform in a given budget. Different approaches to overcome this limitation have been proposed by the community, such as analytical models [Hong and Kim 2010; Marin and Mellor-Crummey 2004], machine learning [Cavazos et al. 2006; Petit et al. 2012], checkpoint restart [Haine et al. 2014], or simulations [Sherwood et al. 2001]. An interesting and versatile approach is code isolation [Lee and Hall 2005; Petit et al. 2006; Liao et al. 2010; Akel et al. 2013]. Usually, in scientific applications, the hotspots represent a small fraction of the total source lines [Knuth 1971]. Code isolation finds and extracts the hotspots of an application as stand-alone fragments of code, called *codelets*. Codelets can be compiled and replayed independently from the original application. For each codelet, the isolation process captures the memory working set and the relevant machine state such as the cache content to achieve realistic replays.

Breaking an application into independent codelets provides multiple benefits. Executing isolated codelets instead of whole applications is faster and enables piecewise evaluation and optimization of an application. Indeed, different codelets may expose different performance bottlenecks and react differently to optimizations. With code isolation they can be individually modified to evaluate the payoff of new optimizations and tune performance at a fine-grain level. Fast benchmarking and system selection [de Oliveira Castro et al. 2014] can be achieved by replaying a reduced set of carefully selected codelets.

Effective code isolation raises multiple challenges. First, to be practical, isolation must support many programming languages, applications, and optimizations. Second, codelets should be replayable on a variety of target architectures. Third, to achieve accurate performance measures, the memory working set and cache state must be captured and restored before each replay. Tracing the memory is a complex and costly process which must be tuned to get a good trade-off between capture overhead and accuracy of replay. Fourth, different invocations of the same codelet may have different performance behaviors, which depend on the working set and cache state of each invocation. Nevertheless, we observe that the invocations can be clustered in a reduced number of representative performance classes. As a result, replaying one codelet independently from the rest of the application can be several orders of magnitude faster than to replay the full application. As demonstrated on the use case of Section 4.3, this speedup brings a substantial advantage for piecewise benchmarking and optimization.

In this article, we propose CERE, an Intermediate Representation (IR) level code extractor framework based on LLVM. CERE extracts and replays the application source loops as codelets. Our contribution is composed of an instrumenter, a clustering approach to find representative invocations and representative codelets, a working set capture mechanism operating at the system memory page granularity, and a realistic access history cache warmup. CERE is made available under the MIT open-source license.

Figure 1 presents the full CERE pipeline. CERE takes as an input the source files of an application or a benchmark suite. All the languages supported by the LLVM front-ends (C, C++, Fortran, D, etc.) are accepted. The source loops are outlined and instrumented with profiling probes to identify the application hotspots. We filter out short loops that contribute less than 1% to the execution time. An optional codelet similarity analysis [de Oliveira Castro et al. 2014] selects a minimal set of representative codelets. Then, a clustering algorithm analyzes the performance trace of each codelet to find a representative subset of invocations. The memory and cache state of each selected invocation is then captured and dumped to disk. The output of this process is a set of representative codelets and invocations, which can be redistributed, recompiled, and replayed on different systems and architectures. The codelet set can be used as a proxy for original application in optimization or benchmarking studies.

We evaluate CERE on the NAS [Bailey et al. 1991] and the SPEC 2006 FP [Henning 2006] benchmarks. CERE's codelets capture 90.9% of the SPEC benchmarks runtime

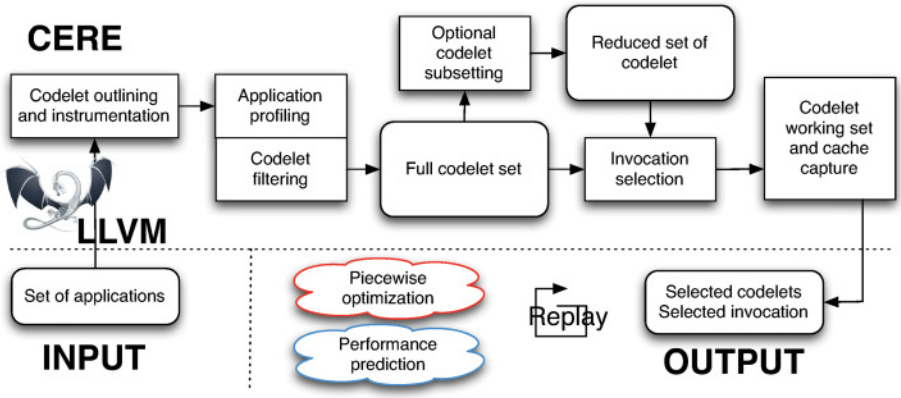


Fig. 1. CERE usage diagram. Applications are partitioned into a set of codelets, which may be pruned using different criteria. A set of representative invocations are selected and captured. The codelets can then be replayed with different options and on different targets to do piecewise optimization or predict performance. (LLVM logo courtesy of Apple Inc.)

and accurately replay 66.3% of their execution time. Section 4.2 demonstrates how CERE codelets can be used to quickly compare the performance of three different architectures on the NAS benchmarks. CERE accurately estimates each architecture performance and is 7.3× to 46.6× cheaper than running the full benchmark. When working on a single kernel, the cost of replay is even lower; for example, the industrial test case of Section 4.3 shows a 237× reduction.

Section 2 presents the state of the art for code isolation, memory capture, and cache warmup approaches. Section 3 gives an overview of the CERE framework. Section 4 evaluates CERE on the NAS and SPEC benchmarks, and showcases CERE on two use cases. First, it shows how CERE can accelerate system benchmarking by finding a small subset of representative codelets. Second, it shows how to use CERE to speed up compiler autotuning on a depth-imaging proto-application.

2. BACKGROUND

In this section, we review previous works on codelet extraction, working set capture, cache capture, and practical applications of codelets.

2.1. Codelet Extraction

Code isolation and replay raise several challenges. The first one is choosing the right granularity for code isolation. Two possibilities have been proposed in previous works: assembly isolation and source code isolation. CERE explores a new level: IR isolation.

Assembly isolation [Sherwood et al. 2001, 2002; Lafage and Seznec 2001] extracts codelets as blocks of assembly instructions. Simpoint [Sherwood et al. 2001] successfully speeds up architecture simulation by sampling a limited number of assembly codelets. Yet assembly isolation is not practical for performance tuning because the assembly code cannot be recompiled with different performance flags or easily retargeted to a new architecture. The extraction software is tied to a specific instruction set architecture. It is also difficult to map assembly codelets to source code regions. However, this approach is language agnostic and resilient to the compiler effect: what you extract is what is executed.

Source code isolation [Akel et al. 2013; Lee and Hall 2005; Petit et al. 2006; Liao et al. 2010] is portable and can be easily used to tune compiler options [Kashnikov et al. 2013] or to select the best architecture [de Oliveira Castro et al. 2014]. Furthermore,

because extraction occurs at source level, before compiler transformations, the performance information gathered during replay can be easily mapped to the source high-level constructs. Unfortunately, source code isolation requires a specific parser and extraction process for each language. Therefore supporting multiple languages is extremely costly because writing a robust extraction pass for complex languages, such as C++, is technically challenging. Finally, one must ensure that the source level extraction process does not alter the performance behavior of the original hotspot. Indeed, some of the transformations used during source isolation may hinder compiler optimization passes [Akel et al. 2013; Liao et al. 2010].

In our work, we explore code isolation at the IR level, which provides a good trade-off between assembly and source code isolation. We choose to target the LLVM [Lattner and Adve 2004] compiler IR. CERE extraction is therefore tied to the LLVM compiler but it supports all of LLVM front-ends and back-ends with no extra engineering cost. Extracting codelets at the IR level is much simpler than at the source code level, which requires parsing complex input languages. It also facilitates the process of instrumenting the code, capturing the memory, and outlining the codelet thanks to the powerful integrated flow analysis passes as detailed in Section 3.

IR codelets provide many performance tuning opportunities. For instance, the codelet can be replayed using different LLVM optimization passes or versions, enabling compiler flag autotuning [Kashnikov et al. 2013]. By leveraging the available LLVM code generation back-ends, codelets can be replayed on different architectures to facilitate system codesign (see Section 4.2).

2.2. Memory Capture and Cache Warmup

Memory capture raises two challenges: capturing the codelet working set, and the cache hot set.

Memory Capture. Before replaying a codelet, the memory state from the original execution must be restored. This ensures that the replayed execution will be equivalent to the original one, even considering data dependent branching code.

Multiple techniques exist to checkpoint the original memory state. Code isolator [Lee and Hall 2005] analyzes the static dataflow of the original application to determine which data structures need to be captured. This method produces small dumps because only the required data are captured, but cannot deal with pointer aliasing. Astex [Petit et al. 2006] captures the convex hulls of the memory accesses. However, it does not preserve the data layout information and does not remap the memory at the same addresses during replay. Therefore pointer-based structures such as linked lists are not supported.

Codelet Finder [CAPS 2013; Akel et al. 2013] takes a full snapshot of the original application address space. A full memory dump is very large but handles the pointer aliasing problem since the full memory is recorded. It also preserves the relative alignment and offsets among data structures. Nevertheless, a full snapshot of the application memory for each codelet can be prohibitive in terms of memory and replay time.

In this article we propose a page level granularity snapshot. Using the memory protection mechanism we capture the memory pages containing the working set. During replay we remap this set of pages at their original addresses. This ensures that the dump remains small and fast. Furthermore, the replay works even with complex pointer aliasing, because the memory layout is preserved.

Cache Capture. Capturing the memory working set of the original execution ensures that during the replay, the data accessed are the same as during the original run. However, it is not enough to guarantee that the replay and original run have the same

execution time. Indeed, to faithfully capture the performance of the original region it is necessary to warm up the system to match as close as possible the original context. This issue is referred to as the cold start bias.

Usual techniques [Kessler et al. 1994] mitigate cold start bias by modeling the warmup effects during a window of time preceding the region of interest. Multiple heuristics [Conte et al. 1996; Haskins Jr and Skadron 2003] have been proposed to optimally determine the window's size.

Two main approaches have been proposed in the literature for cache state warmup in code isolation. The first approach is to warm up the cache by running a few executions of the codelet itself [Petit et al. 2006; CAPS 2013; Akel et al. 2013]. The rationale is that the hotspots forming the codelets are loop based and thus can be warmed up by their own previous iterations. This heuristic proves to be efficient in many cases [de Oliveira Castro et al. 2014; Akel et al. 2013]. The second, more accurate approach, warms up the cache by replaying the history of the memory accesses in a simulator [Sherwood et al. 2001] or using a warmup routine [Lee and Hall 2005]. These techniques require one to trace memory accesses, which is costly and incurs significant slowdowns [Gao et al. 2005].

In this article, we propose two warmup approaches. The first is an optimistic warmup strategy that preloads the whole working set into the cache. The second, is a page memory tracing technique, which warms the cache by replaying memory access history at the memory page granularity.

2.3. Invocation and Codelet Subsetting

The last challenge in code isolation is reducing the replay and codelet capture cost. The replay and capture cost is related to two quantities: the number of codelets and the number of invocations to capture and replay. When codelets or invocations have similar behaviors, it is desirable to only capture and replay a representative subset.

A single region in a program may be called thousands of times with different working sets and cache states. Capturing each individual invocation is prohibitive. Akel et al. [2013], Lee and Hall [2005], and Petit et al. [2006] allow the user to manually choose which invocations should be captured and replayed. Sherwood et al. [2002] identify similar computation phases by computing the distance between the region's basic block vectors. Other works [Eeckhout et al. 2005; Hoste et al. 2006; Hoste and Eeckhout 2006, 2007; Phansalkar et al. 2007; de Oliveira Castro et al. 2014] use performance feature vectors as a measure of region similarity.

CERE includes both an invocation clustering approach that automatically selects representative invocations and a codelet clustering approach that selects representative codelets. Combined, these reduction techniques significantly lower the number of required capture and replay runs.

3. CERE: CODELET EXTRACTOR AND REPLAYER

3.1. IR Capture and Replay Overview

CERE (Codelet Extractor and REplayer) targets the LLVM IR. IR provides multiple advantages over source or assembly code isolation techniques as discussed in Section 2.1.

Because it operates at the IR level, CERE can use any LLVM front-ends. For example, CERE has been tested on all NAS and SPEC 2006 FP programs. While C and C++ benchmarks used the Clang front-end, Fortran programs used the GCC gfortran front-end through the dragonegg plugin [Sands 2009]. CERE also works on less mainstream languages. For example, CERE successfully extracts codelets from D [Alexandrescu 2010] applications compiled with the LLVM D front-end, LDC.

Table I. Codelet Capture and Replay Main Steps

#	Step	Output

1	**Front-end:** Transform the C, C++, Fortran, D input program into LLVM IR (uses Clang, dragonegg, or LDC).	`original:`
		`%0 = load i32* %i, align 4`
		`%1 = load i32* %s.addr, align 4`
		`%cmp = icmp slt i32 %0, %1`
		`br i1 %cmp, ; loop branch here`
		`label %for.body,`
		`label %for.exitStub ...`

2	**Outline:** Outline the region to extract. Flow analysis is used to compute all live-in and live-out values which are passed as arguments. (see Section 3.3).	`define internal void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr) {`
		`%0 = load i32* %i, align 4`
		`...`
		`ret void }`
		`original:`
		`call void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr)`

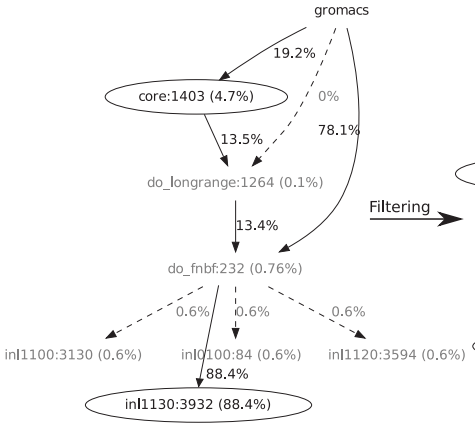
3	**Capture:** Insert calls to CERE capture library. Run the instrumented binary to capture the run-time state. (see Sections 3.4 and 3.5).	`define internal void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr) {`
		`call void @start_capture(i32* %i,`
		`i32* %s.addr, i32** %a.addr)`
		`%0 = load i32* %i, align 4`
		`...`
		`call void @end_capture()`
		`ret void }`

4	**Replay:** Generate minimal replay wrapper that calls the outlined region. Compile and run replay possibly with new optimization options or on a different architecture. (see Section 3.6).	`define i32 @main(i32 %argc, i8** %argv){`
		`; Allocate clone variables`
		`%i = alloca i32`
		`%s.addr = alloca i32`
		`%a.addr = alloca i32*`
		`; Restore arguments and memory`
		`call void @restore(...)`
		`; Call outlined region`
		`call void @outlined(i32* %i,`
		`i32* %s.addr, i32** %a.addr`
		`; Anti-deadcode for live-out values`
		`call void @antideadcode(i32* %i)}`

Table I presents CERE's capture and replay process of a selected codelet. In Step 1, the input program is compiled to LLVM IR.

In Step 2, the region to be captured is outlined in a separate function using the CodeExtractor LLVM pass. CodeExtractor does a flow analysis to detect all the live-in and live-out dependencies of the region to extract [Mikushin et al. 2013]. This pass simplifies the codelet extraction process, since it extracts the region code in its own function. The codelet region is outlined in a new function. Finally CodeExtractor inserts a call to the outlined function in the original code. The dependencies are preserved by passing the live-in and live-out values through function arguments. CodeExtractor is also the starting point for our portable memory capture mechanism discussed in Section 3.4.

Step 3 generates the instrumented binary for memory capture. It inserts special calls to our capture library before and after the outlined region in the original application. The calls are used to trigger the memory and cache warmup state captures, described in Sections 3.4 and 3.5. The instrumented binary execution generates a set of dump



Codelet	Error (%)
do_longrange	2.3
do_fnbf	1.1
inl1130	0.1
inl0100	10.1
inl1120	4.6
inl1100	7.5
core	2.5

Fig. 2. *(Left)* CERE call graph, before and after filtering, for SPEC 2006 gromacs. Each node represents a captured codelet. The percentage inside the node is the codelet's self time. Edges represent calls to other codelets; the edge percentage is the time spent in calls to those nested codelets. *(Right)* Replay percentage error of gromacs codelets using Working Set warmup.

files that can be used during replay to restore the memory state and to warm up the caches. The aim is to ensure that the replay context closely mimics the original execution context.

Step 4 is the replay mechanism. It generates a wrapper to directly call the outlined region. This wrapper performs important steps to restore the original execution environment, such as variable cloning, cache, and memory restoration. The replay IR code can be compiled with different optimization flags to find the best performance configuration. Or it can be compiled with different back-ends to evaluate the performance on multiple targets. The replay process is detailed in Section 3.6.

3.2. Application Partitioning

To find interesting codelets for performance optimization, CERE concentrates on the application hotspots. In scientific applications, performance is mainly concentrated on loops. Therefore, CERE considers all the loops of the original program as potential candidates to be extracted as codelets. Then, CERE profiles the candidate loops and keeps the ones significantly contributing to the total execution time.

CERE provides two loop level profiler modes. First, a low overhead sampling profiler based on the Google Performance Tools library [gpe v221]. Second, an instrumentation profiler, which is slower but more precise. When using sampling, CERE outlines the loops before executing the application; all the outlined loops are profiled using Google's performance toolkit. When using the instrumentation mode, probes to capture the time stamp counter are inserted directly before and after the loop.

Despite our efforts to restore the original execution environment through warmup code reinlining, and variable cloning (see Sections 3.5 and 3.6), the codelet replay sometimes does not match the original loop performance. Clearly, those *ill-behaved* codelets cannot be used as a performance proxy in benchmarking or optimization studies. Therefore CERE runs a sanity check where it replays and profiles each codelet to ensure that only valid codelets are returned to the user. The tolerated discrepancy threshold can be configured. Its sensitivity is presented in Figure 3. In this article we consider that codelets match the original performance when the replay error is under 15%.

After collecting profile data, CERE produces an annotated call graph such as the graph in Figure 2. This call graph is then pruned by removing the loops contributing

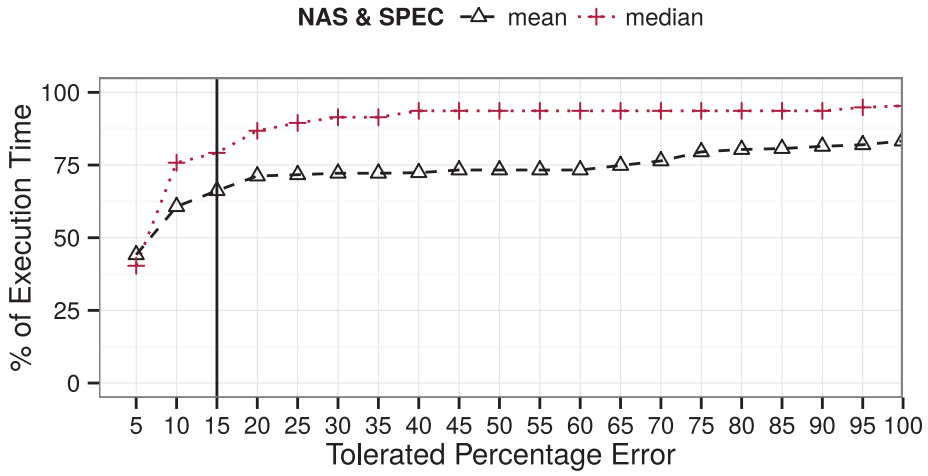


Fig. 3. Mean and median captured execution time as a function of the tolerated replay error. The NAS and SPEC 2006 FP benchmarks. The mean is lower than the median due to the IO-intensive and short kernel benchmarks described in Section 4.1, which skew the distribution.

for less than 1% to the total execution time. Furthermore, if an *ill-behaved* codelet is detected, CERE also removes it from the call graph. When removing a loop from the call graph, we propagate its *self time* to its parent codelets. In our example, the time from the three `inl` removed loops is propagated to their caller `do_fnbf`. In the example of Figure 2, since all the codelets match the original execution time, none would be removed.

Once the removal process is over, CERE extracts all the remaining loops as standalone codelets.

The previous selection algorithm extracts all the well-behaved codelets whose contribution to the program execution time is over a given threshold. To trade coverage for replay time, for example, when using codelets to accelerate system benchmarking, the user wants the minimal set of codelets that can be quickly replayed while simultaneously capturing the application performance accurately. For this purpose, CERE includes a codelet *selector* that uses integer linear programming to find an optimal codelet set. It is similar to the tuning selection algorithm proposed by Pan and Eigenmann [2006]. In the example in Figure 2 it would drop codelets `do_fnbf` and `core`, losing less than 7.35% coverage but significantly reducing the replay cost.

3.3. Codelet Checkpoint-Restart Strategy

Traditional checkpoint techniques [Duell 2005] can save the state of a program at any given point. A full dump of the memory and of the register banks including the program counter allows one to restart the program after capture. Yet, this approach requires that the replayed code keeps the same code layout and uses exactly the same registers as during the capture. Traditional checkpointing is therefore not suited to test compiler optimizations that may remap registers or change code layout. Also it limits codelet portability to architectures sharing the same Application Binary Interface (ABI) and register layout.

Codelet-based piecewise iterative optimization and architecture selection require a portable checkpoint-restart strategy. The outlining pass (Step 2 in Table I) wraps and isolates the region of interest inside a separate function. Because the region now follows a function call, we can guarantee that the accessed data is either in memory or is passed as arguments to the outlined function.

This enables us to simplify the memory capture process: only the memory and arguments to the outlined function must be recorded. Also, the outlined function prototype acts as a clean interface that enables us to recompile and apply transformations to the codelet before replay. Because no assumptions about the register layout are made, codelets are portable across architectures that do not change the memory layout, such as word size and endianness. Our tests have shown, for example, that our codelet replayer allows one to recompile changing optimization flags, capturing on -O0 but replaying on -O3, or changing architectures, capturing on Core Duo and replaying on Atom.

Codelet portability has been extensively tested and works across six different Intel CPU generations (Atom, Core 2 Duo, Nehalem, Sandy Bridge, Ivy Bridge, and Haswell) running various 64-bit Linux distributions on the NAS and SPEC codelets.

We also tested codelet portability between an Intel Core i3 running 32-bit Linux and an embedded target, an ARM1176JZF-S on a Raspberry Pi Model B+ running 32-bit Linux. This test was conducted on a simple benchmark summing the elements of a large integer array. The capture was performed on the Core i3 system and could be faithfully replayed on the ARM embedded target.

In a second experiment the capture was done on the same Intel Core i3, but this time the system was 64-bit Linux; therefore some of the dumped pages were over the 32-bit address space limit. The replay on the ARM system failed because addresses over 32bits overflowed. This example illustrates the limits of CERE: portability does not work out of the box for systems with different memory address sizes. Nevertheless, in this case we were able to overcome this limitation by manually remapping the memory dump to fit the 32-bit address space by masking the address' upper bits. After the manual remapping, we were able to replay the benchmark in the ARM1176JZF-S processor.

The outlining process guarantees that codelets captured once can be distributed and replayed many times on multiple architectures.

3.4. Capturing the Memory

CERE captures codelet's working sets by intercepting accesses to the memory pages. Page level capture combines the advantages of the full memory dump in Codelet Finder [Akel et al. 2013] with the advantages of the dataflow capture in Code Isolator [Lee and Hall 2005] or Astex [Petit et al. 2006]. First, CERE guarantees that all the memory locations accessed by the original program are dumped, including aliases that are not handled with static analysis. The set of captured pages contain the full original working set. Second, because only the touched pages are saved, the memory dump is the smallest page-granularity overapproximation. Therefore, it can be easily stored and distributed.

Figure 4 shows the memory dump process. First, all the memory pages of the process are protected and a special segmentation fault handler is set. Each time a protected page is accessed, a segmentation fault occurs and triggers the handler. The handler dumps the touched memory page to disk and unprotects it before continuing the original program execution.

It is important to protect all newly allocated memory. If memory is allocated but returned to the user unprotected, the tracer misses the access to the memory segment. We catch all calls to the memory allocation library, such as malloc, realloc, or memalign using the `LD_PRELOAD` mechanism. However, some special memory sections must not be protected, such as the pages containing the code of the tracing library and the segmentation fault handler itself. Therefore CERE carefully avoids protecting its own pages and system specific memory segments.

Figure 5 compares the average dump size for the NAS benchmark codelets for two techniques: CERE's page-granularity dump and Codelet Finder's full dump. As can be

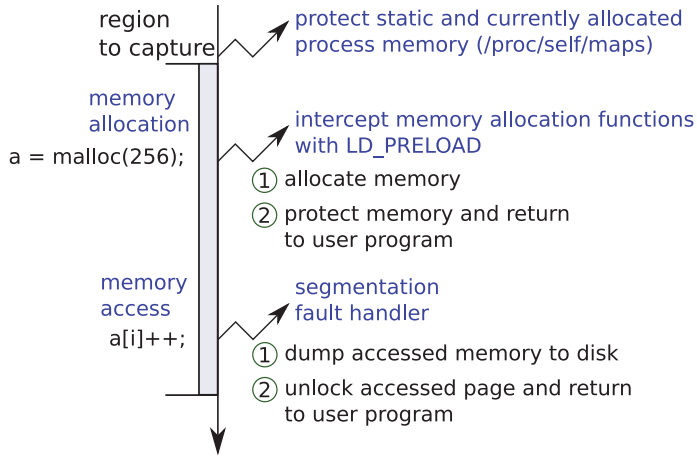


Fig. 4. The memory dump process operates at page granularity. Each page accessed is dumped by intercepting the first touch using memory protection support.

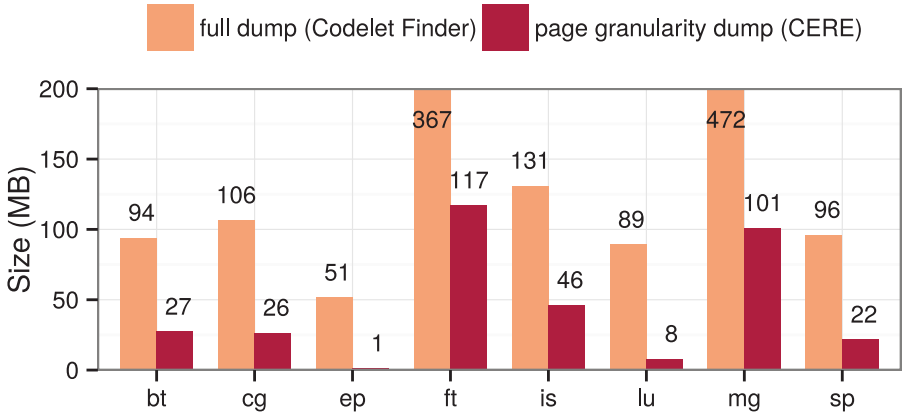


Fig. 5. Comparison between the page capture and full dump size on NAS.A benchmarks. CERE page granularity dump only contains the pages accessed by a codelet. Therefore it is much smaller than a full memory dump.

seen, the page-granularity dump is 3 to 51 times smaller than a full dump. With this technique CERE extracts light portable codelets from industrial application with large working sets.

3.5. Capturing the Cache State

In this section, we address the problem of cache warmup for codelet replay previously discussed in Section 2.2. CERE includes three warmup strategies: *Cold*, *Working Set*, and *Page Trace*.

The *Cold* strategy does not do any warmup before executing the codelet. It is therefore inaccurate but has no overhead. It can be used on long codelets for which the cold start bias is negligible.

The *Working Set* strategy prefetches the full working set of the codelet before its execution. It is an optimistic strategy that assumes that the codelet working set was already in cache in the original execution.

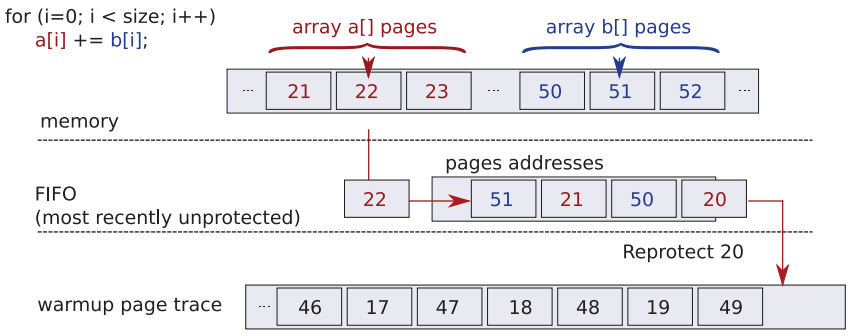


Fig. 6. Cache page tracer on a simple codelet adding two arrays. Each page access is logged. Recently unprotected pages are kept in a FIFO with N slots (here $N = 4$). Once evicted from the FIFO, the pages are protected again.

The *Page Trace* strategy mitigates cold start bias by replaying a memory trace at a page level granularity. It is less accurate than a full memory trace warmup, but much faster. It provides a good trade-off between cost of codelet capture and replay accuracy. The technique is similar to the page tracing technique in Burton and Kelly [2006].

Our page tracer is implemented on top of the memory dump process described in Section 3.4: all the memory pages are protected, and a special segmentation fault handler intercepts accesses to memory. The difference is that unlike the memory dump in which only the first touch to a page is important, the page tracer should capture all the memory accesses to a page.

An exact, but costly, technique involves reprotecting each page after each access. Because this page is immediately reprotected, further accesses to the page will provoke a segmentation fault and will be logged by the tracer. The slowdown is too high for our purposes.

To reduce the cost of the technique, we keep the most N recently accessed pages unprotected. The tracer uses a First-In First-Out (FIFO) to track the recently accessed pages. Each time an access to a page is detected, the page is unprotected and added to the FIFO. The oldest page is popped from the FIFO, reprotected, and added to the page access log. Figure 6 illustrates this approach on a codelet that adds two arrays together.

If a codelet simultaneously accesses less than N separate memory streams, the FIFO ensures that a page remains unprotected for all the consecutive streamed accesses. Assuming a stride-one access, the page tracer handler is only invoked every 4096 bytes (for 4K pages). Therefore, we choose N higher than the number of separate memory streams accessed by most loops. Kashnikov et al. [2013] show that most application loops use less than 16 simultaneous streams. In our experiments we choose $N = 64$. Nevertheless, by keeping the most recent N pages unprotected, our trace is less accurate. In the code of Figure 6, for instance, each cell of array a is accessed twice (it is first read then written to), but the page tracer only sees the first access. When interpreting the trace, one must keep this inaccuracy in mind: the trace presents which pages were accessed but neither how many times nor the precise ordering.

Figure 7 compares the three warmup techniques implemented in CERE on the NAS benchmarks. The *tolerated error* is the maximum percentage difference between the original execution time and the replayed execution time. The plot shows the percentage of execution time of NAS codelets replayed with an error smaller than the *tolerated error*. For example, if we use the *Cold* strategy, 70% of the execution time can be replayed with an error under 15%.

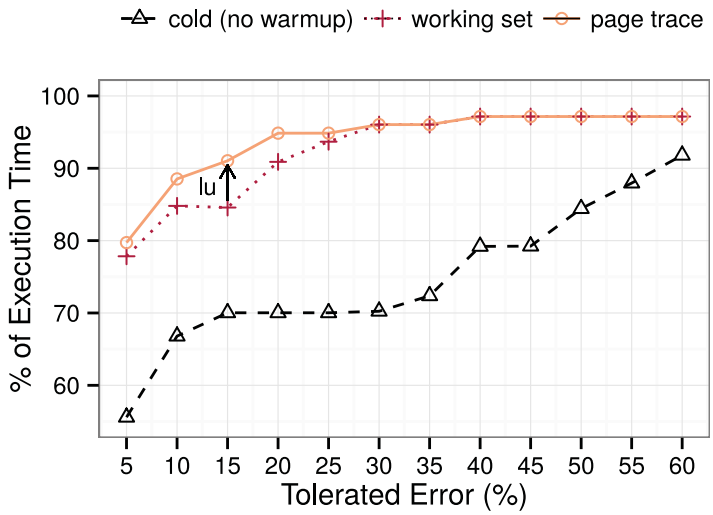
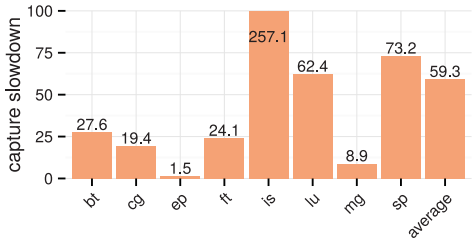


Fig. 7. Comparison of the three cache warmup techniques included in CERE on NAS codelets. The plot shows the percentage of execution time as a function of the replay error. Page Trace and Working Set warmup achieve the best results. Page Trace is more accurate than Working Set on the LU benchmark.



(a) CERE capture overhead (Class A).

	ATOM 3.25	PIN 1.71	Dyninst 4.0
cg.a	98.82	222.67	896.86
ft.a	44.22	127.64	1054.70
lu.a	80.72	153.46	$\gg$301.4
mg.a	107.69	168.61	989.53
sp.a	67.56	93.04	$\gg$203.66

(b) Overhead of other memory tracers.

Fig. 8. CERE capture overhead. For each plot we measure the slowdown of a full capture run against the original application run. (The overhead takes into account the cost of writing the memory dumps and logs to disk and of tracing the memory accesses during the whole execution.) We compare to the overhead of other memory tracing tools as reported by Gao et al. [2005]. Gao et al. did not measure bt, is, and ep.

Figure 8 shows the overhead of the capture run for the NAS benchmarks. For each benchmark we compute the slowdown between the original runtime and a full capture run. This measure includes initialization of the capture library, writing the dumped pages and the memory trace logs to disk for all the codelets in the application. `IS` is particularly slow because one of its codelets accesses memory randomly. This rapidly fills the pages FIFO and slows down the tracer. Figure 8 also compares CERE capture cost with the overhead of other memory tracing tools. CERE overhead is similar to ATOM 3.25 overhead and lower than PIN 1.71 and Dyninst 4.0 overhead.

When the user is only interested in a single codelet, CERE includes a single-capture mode, which is much faster. The capture library fast-forwards the execution and starts the memory tracer and memory protection when the execution is reaching the zone of interest, but leaving a big-enough window to capture the cache warmup log. Conte et al. [1996] and Haskins Jr and Skadron [2003] propose multiple techniques to determine the best window size.

We observe that the *Working Set* and *Page Trace* strategies significantly improve the replay accuracy. On the NAS codelets, the *Page Trace* strategy is slightly better than that of the *Working Set*. The improvement comes from LU codelets whose irregular

accesses are better captured by the *Page Trace* warmup. Section 4.3 shows another example where the *Page Trace* warmup is more accurate than the *Working Set* optimistic warmup.

3.6. Replay

Once the memory and cache state are captured, a codelet can be replayed. Because codelet replay is fast, it is useful to quickly evaluate the impact of moving to a different architecture or changing compiler optimizations.

To replay a codelet, CERE generates the special wrapper shown in Step 4 of Table I. First, it allocates clone variables for the input and output flow dependencies to the outlined region. Second, it restores memory and cache state. Finally, it calls the outlined region.

A first remark is that the outlined region results are not used when returning from the call to the codelet. Therefore, LLVM dead code elimination pass is free to fully optimize by removing this call. Clearly that is not our purpose. Therefore, during replay we insert, for each live-out variable, a special `antideadcode` call. It is an empty external function that forces LLVM to keep the codelet's code, even when using highly aggressive optimization levels such as `-O3`.

A second remark is that the outlining compilation pass dereferences the input and output dependencies. By passing the variables by reference, it is easy to preserve the values modifications during the codelet execution. This is a classic technique in code outliners [Lee and Hall 2005; Liao et al. 2010], which has the unfortunate side effect of disabling many compiler optimizations. In many codelets, dereferencing makes codelet replay slower and therefore unfit to be used as performance proxies of the original code. We solve this problem in three steps. First, we tag each dereferenced pointer with the IR attribute *NoAlias*, which informs LLVM that the dereferenced pointer is not aliased. This is known because the extra dereference is created by the CERE outliner and used only once during replay. Second, we tag the outlined function itself with the attribute *AlwaysInline*, which forces LLVM to reinline the function in the replay wrapper. Third, LLVM alias analysis optimization pass removes the extra layer of dereference. In Section 4 the effect of reinlining and marking cloned variables as NoAlias are measured. These two techniques improve replay accuracy in 11 applications without degrading the other benchmarks

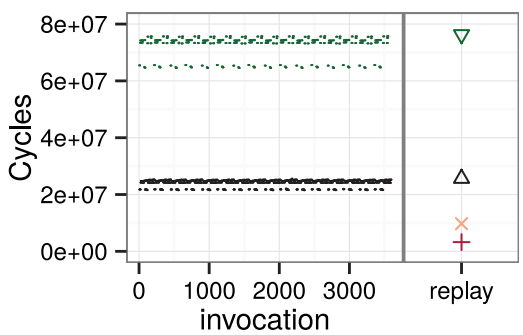
One could think that the outlining step is unnecessary since it is reverted later on by LLVM inliner pass. But as explained in Section 3.3, the outlining step guarantees that CERE finds a safe checkpoint to capture the context just before a procedure call.

Once the replay wrapper is generated, it is compiled and possibly optimized depending on the optimization flags selected by the user. To generate the final replay binary, CERE uses a custom link script that reserves the virtual memory segments occupied by the working set pages during the memory capture. This step is needed so that CERE can preserve the original memory layout.

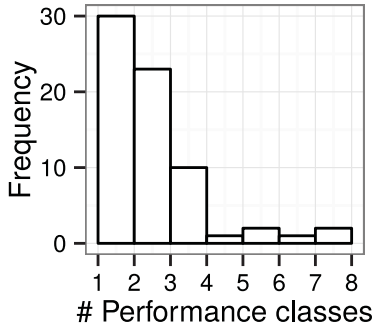
3.7. Invocation Selection

Inside an application, the same codelet may be called multiple times. In many codes two invocations of the same codelet may have different execution times. This is due to the different working sets or initial conditions.

For example, the codelet `make_ft@shell2.F90:1133` extracted from `tonto` is one of the steps of a specialized fast Fourier transformation. In the original application, this loop is called 3587 times with different workloads. Figure 9(a) shows its execution trace. A cluster analysis of the invocations reveal that they can be sorted into four performance behaviors, which are represented with different colors in the figure. Other codelets such as `flux_lam@flux.f:58` extracted from `bwaves`, have a constant workload size but the first invocation is slower because of cache warmup effects.



(a) SPEC tonto `make_ft@shell2.F90:1133` execution trace.



(b) Distribution of the number of representative performance classes across all NAS benchmarks.

Fig. 9. Working set reduction: (a) A clustering analysis of tonto's trace detects four different performance behaviors depending on the workload. The initial 3587 invocations are captured with only four representative replays. (b) Most of the NAS codelets can be captured with less than four representative working sets.

To accurately replay a codelet, we must capture each different invocation state. When the number of invocations is high, this process becomes costly both in time and space. Fortunately, applications exhibit some regularity; and most of the time the invocations can be reduced to a few representative classes. Figure 9(b) shows the distribution of the number of different classes across all NAS benchmarks. One can note that most of the codelets can be captured with less than four representatives. The fact that the performance of a program can be reduced to a few number of representatives has been observed in other domains such as value profiling [Khan et al. 2008; Petit and Bodin 2010; Calder et al. 1997] and iterative compilation [Chen et al. 2010].

To automatically detect the performance classes and generate a set of representative captures, we use a clustering algorithm. Some of the codelets in our benchmarks have large traces with more than 10^9 invocations and cannot be analyzed using traditional clustering algorithms such as hierarchical clustering or *K*-means. To be able to efficiently process such large datasets CERE uses CLARA (CLustering LARge Applications) algorithm [Kaufman and Rousseeuw 2009]. CLARA relies on sampling to reduce the cost of clustering. It extracts a random sample from the original dataset and finds the cluster medoids. The sampling process is repeated to reduce the bias in the medoid selection. Finally, each point of the original dataset is assigned to the nearest medoid's cluster.

Once the performance classes are identified, CERE selects one representative invocation per class. CERE selects the invocation closest to the medoid of the cluster, in other words, the invocation closely matching the median performance of all the invocations inside the cluster. When replaying the benchmark, CERE extrapolates the full benchmark performance by weighting each representative replay time according to the contribution of its performance class in the original execution.

Thanks to invocations reduction, CERE is able to accelerate performance evaluation considerably because only a representative subset of the invocations is replayed: for example, only two out of 10,000 invocations are replayed for codelet `updateTestEv@soplex.c:204` in SPEC 2006 soplex benchmark.

3.8. Codelet Subsetting

A second reduction in the number of replays can be achieved by detecting and exploiting similar and repeated computation patterns. Benchmark suites and applications naturally contain redundant computation patterns across different benchmarks. For

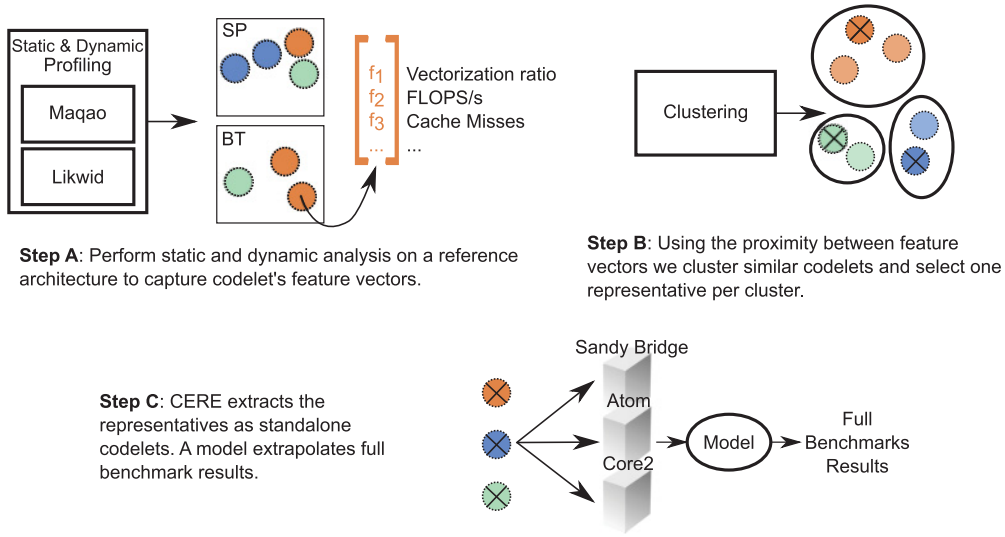


Fig. 10. Overview of the benchmark reduction method for execution time prediction. The reduction method can be extended to support compiler or optimization evaluation.

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.

The method presented in Figure 10 detects repeated computation patterns, and keeps only one representative copy of each, reducing a suite of benchmarks to an essential set of microbenchmarks. Because only duplicated patterns are removed, the important performance features of the original benchmarks are preserved. The method is presented in detail in de Oliveira Castro et al. [2014].

Step A statically analyzes and profiles each codelet on an architecture chosen as a reference. We measure both static and dynamic features. Static features are extracted by the MAQAO code quality analyzer [Djoudi et al. 2005; Kashnikov et al. 2013], which provides detailed low-level performance metrics. Dynamic features are provided by Likwid 3.0 [Treibig et al. 2010], which reads the hardware performance counters. Each codelet is tagged with a feature vector that gathers MAQAO and Likwid measurements. The feature vector is used as a performance signature to detect similar codelets.

Step B groups codelets sharing similar feature vectors into clusters.

Step C selects a representative for each cluster and extracts it as a stand-alone CERE codelet.

Codelets from the same cluster share the same features and should react in the same way to architecture or compiler changes. For example, memory-bound codelets will benefit from faster caches, whereas highly vectorized codelets will benefit from wider vectors. Therefore, by measuring a single representative per cluster we can extrapolate the performance of all its siblings. Given an initial benchmark suite, our method produces a set of reduced benchmarks that can be used in place of the original one for system selection. This method is evaluated in Section 4.2.

4. EVALUATION OF CERE

In Section 4.1, we evaluate CERE capture coverage and replay accuracy on the NAS 3.0 serial benchmarks and the SPEC 2006 FP benchmarks. All the benchmarks in both test suites are used in our evaluation, therefore CERE was tested on 26 different

Table II. Test Architectures

	Atom	Core 2	Nehalem	Sandy Bridge	Ivy Bridge	Haswell
CPU	D510	E7500	L5609	E31240	i7-3770	i7-4770
Frequency (GHz)	1.66	2.93	1.86	3.30	3.40	3.40
Cores	2	2	4	4	4	4
L1 cache (KB)	2×56	2×64	4×64	4×64	4×64	4×64
L2 cache (KB)	2×512	3 MB	4×256	4×256	4×256	4×256
L3 cache (MB)	-	-	12	8	8	8
Ram (GB)	4	4	8	6	16	16

benchmarks in total. NAS benchmarks were tested on class *A* and *B* datasets. SPEC benchmarks were tested on *ref* datasets.

In Section 4.2, we demonstrate how CERE can be used to significantly accelerate the benchmarking process during system selection. To achieve this, we replicate the experiments presented in de Oliveira Castro et al. [2014] using CERE.

The experiments were performed on the machines described in Table II. They belong to six different Intel CPU generations (Atom, Core 2 Duo, Nehalem, Sandy Bridge, Ivy Bridge, and Haswell) and possess quite distinct memory hierarchies. These machines were selected to validate that CERE replay process is portable across architectures.

We compared the replay times of the NAS codelets with memory captures done on Core 2 and Haswell and observed no difference. We conclude that the architecture used for capturing the memory has no significant impact on replay accuracy. Yet for completeness, the reader should note that the final memory capture dumps used in the following experiments were performed on the Core 2 machine for the NAS benchmarks and on Haswell for the SPEC benchmarks.

The experiments were performed with CERE v0.1.0. C and C++ benchmarks were compiled with Clang 3.3 and Fortran benchmarks were compiled with GCC 4.6 through dragonegg.

4.1. Coverage and Replay Accuracy

As discussed in Section 3.2, we consider that a codelet is accurately replayed if its replay performance is within 15% of the original execution time.

Performance is measured using the Time Stamp Counter, which provides a precision around 200 cycles. To ensure that the error upper bound due to measurement noise remains approximately 10% for all codelets, we removed codelets whose execution time was less than 2000 cycles per invocation.

Figure 11 shows for the NAS and SPEC 2006 FP benchmarks the percentage of execution time captured by codelets and the percentage of execution time that could be accurately replayed. On average, the extracted codelets cover 97.3% of the execution time in NAS and 76.6% in SPEC.

On NAS, both coverage and replay accuracy are very high. MG matching is a bit lower (65.1%) than the other benchmarks because of two borderline codelets with replay errors at 16.8% and 18.5%. With a *tolerated error* of 20%, we would have reached 95% coverage.

NAS codelets were replayed in two different architectures to show that CERE reliably supports multiple architectures. The small differences in coverage between Haswell and Core 2 are due to the changes in contribution of codelets to the execution time; for example, CG spends relatively more time on I/Os on Haswell architecture.

SPEC FP results are evaluated on the Haswell architecture. Eleven out of 18 benchmarks have high coverage and replay accuracy, over 75%. Here is a list of the problems affecting the seven remaining benchmarks:

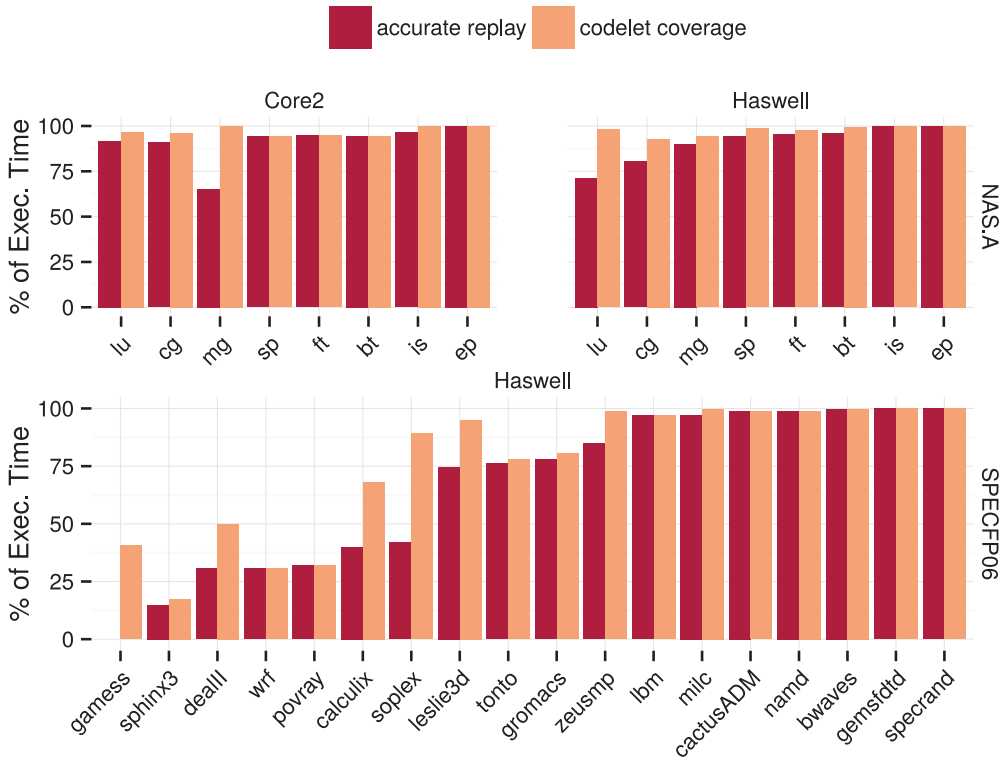


Fig. 11. Evaluation of CERE on NAS and SPEC FP 2006. The coverage is the percentage of the execution time captured by codelets. The accurate replay is the percentage of execution time replayed with an error less than 15%.

sphinx3, wrf, povray, and calculix have low coverage because most of the time is spent in I/O operations. The current version of CERE does not capture codelets performing I/O because the dump does not preserve file descriptors state. However, 100% of captured codelets match.

gamess and dealII have low coverage because most of the performance is spent in loops taking less than 2000 cycles, which were not considered.

gamess has low matching because the only remaining codelet, covering 40% of the execution time, is not accurately replayed. It is due to a warmup bug, which is being investigated.

calculix has low matching because of a borderline codelet `isortii`, which has a replay error of 16% but accounts for 10% of the running time. It is a sort function that is very sensitive to warmup effects.

soplex has low matching because CERE fails, due to a capture bug, to replay its main codelet covering 47.4% of the execution time.

Figure 12 shows that the reinlining and NoAlias-tagging performed during the replay compilation pass are beneficial in 11 benchmarks. Overall CERE coverage and accuracy are high in both NAS and SPEC benchmarks, showing that CERE codelets can be efficiently used as performance proxies for many applications.

CERE has higher replay accuracy than the state-of-the-art code isolator tool, Codelet Finder. On NAS, Codelet Finder accurately replays 69% [Akel et al. 2013] of the execution time, whereas CERE replays 90.9%. On SPEC, Codelet Finder has very low replay accuracy or fails to extract codelets for many benchmarks (the 2013 version

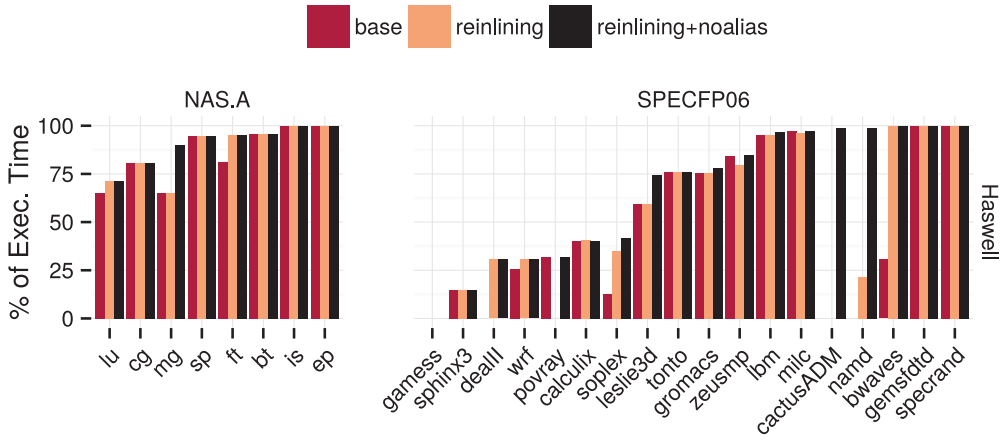


Fig. 12. Percentage of execution time accurately replayed (error $< 15\%$) on the NAS and SPEC FP benchmarks with different replay configurations. Reinlining and explicitly marking cloned variables as NoAlias improve replay accuracy in 11 benchmarks.

of Codelet Finder hangs on gamess, gromacs, cactus, calculix, tonto, specrand, and wrf), whereas CERE accurately replays 66.3% of the SPEC execution time.

CERE includes a report generator that automatically captures the execution traces, selects representative invocations, and computes coverage and replay accuracy of a given set of benchmarks. The reports can be visualized in any modern web browser. The user clicks on any captured codelet in the call graph to see its invocation clustering and replay accuracy statistics. The reports for all NAS and SPECG benchmarks can be viewed at http://benchmark-subsetting.github.io/cere/.

4.2. Benchmark Reduction Use Case

Codelets can be used as fast performance regression test suites for compilers in a continuous integration process. They could also be used as reduced benchmarks for performance studies when testing multiple architectures. Selecting the best computing system for a set of applications is a costly process that requires benchmarking the applications on the different systems. We propose to reduce the benchmarking cost by extracting a set of representative CERE codelets capturing the performance characteristics of the original applications.

We applied the codelet subsetting presented in Section 3.8 to the NAS.B codelets. By clustering similar codelets, 18 representative codelets were selected and extracted using CERE. Then they were replayed in three different architectures: Atom, Core 2, and Sandy Bridge (see Table II). This experiment replicates with CERE a similar study [de Oliveira Castro et al. 2014] that used Codelet Finder instead.

Figure 13 compares the speedup computed using CERE replays to the real speedup measured by running the full benchmark suite. The performance predictions are very close, but CERE replays are 7.3× to 46.6× cheaper than running the full benchmarks.

Table III details the benchmark reduction cost achieved by only replaying the selected representative codelets. We observe that the *Working Set* warmup is much faster than the *Page Trace* warmup that has the overhead of replaying the memory access history. In this particular experiment, the prediction is the same for both warmup techniques, therefore we recommend using *Working Set* warmup, which is much faster. The benchmark cost reduction and prediction accuracy are comparable to the results achieved using Codelet Finder codelets [de Oliveira Castro et al. 2014].

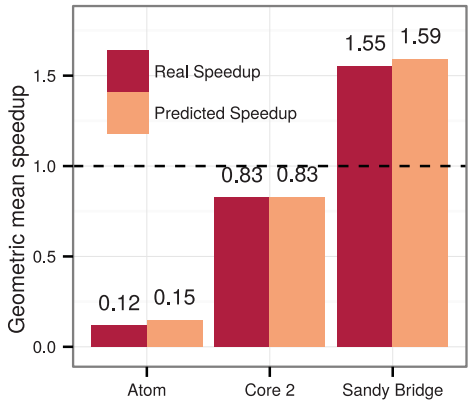


Fig. 13. NAS geometric mean speedup on three architectures. Baseline is a NAS run on Nehalem compiled with `icc 12.1.0 -O3 -xsse4.2`. The predicted speedup is computed by using the replay performance of 18 CERE representative codelets using *Working Set* warmup.

Table III. Benchmarking Acceleration by Replaying only the Representatives
CERE replays are 7.3× to 46.6× faster than running the whole NAS.B suite. CERE benchmark acceleration is comparable to the results achieved with Codelet Finder by de Oliveira Castro et al. [2014].

	CERE		Codelet Finder
(r)2-3 Warmup mode	Working Set	Page Trace	Working Set
Core 2	×30.5	×9.9	×24.7
Atom	×46.6	×10.7	×44.3
Sandy Bridge	×18.3	×7.3	×22.5

4.3. Compiler Tuning Use Case

In the previous section we showed CERE fast architecture evaluation through benchmark reduction. Yet CERE is not limited to architecture benchmarking; it can also be used for quick compiler autotuning. We showcase CERE autotuning capabilities in a Reverse Time Migration (RTM) [Baysal 1983] proto-application used in geophysical depth-imaging applications. The proto-application used here was kindly provided by Asma Farjallah and developed through a collaboration between Total and the University of Versailles Saint-Quentin-en-Yvelines; it implements the finite-difference time-domain (FDTD) step used to solve the wave propagation equation in an isotropic medium. A full description and characterization of the FDTD proto-application is provided by de Oliveira Castro et al. [2013].

The FDTD proto-application is dominated by one Jacobi stencil computation that represents 91.1% of the total running time. In the original application it is called 3,000,000 times. CERE is able to extract the Jacobi stencil codelet. The experiments in this section were performed on Ivy Bridge.

Table IV compares the predicted replay execution times using the three warmup techniques. This codelet is sensible to warmup: *Cold* warmup and *Working Set* warmup are not highly accurate because the replays over- and underestimate memory access costs. In this case, the more realistic *Page Trace* warmup gives the more accurate results. For this reason, the following experiments were all performed using *Page Trace* warmup.

Not only is CERE replay accurate, but it is also much faster than running the original program. Thanks to the invocation selection algorithm presented in Section 3.7, CERE is able to accurately capture the original 3,000,000 invocations behavior using only two representative captures. After accounting for the replay overhead due to warmup and

Table IV. CERE Predicted Execution Times of the FDTD Codelet Compiled with `-O2` using the Three Warmup Techniques

	Original	Page Trace	Working Set	Cold
Execution time (e+11 cycles)	2.34	2.40	2.08	3.13
Prediction error (%)	-	2.56	11.45	25.16

Table V. CERE Performance Predictions for Different Clang Optimization Levels on the FDTD Codelet

	Original (e+11 cycles)	Replay (e+11 cycles)	Prediction error (%)
-O0	2.78	2.88	3.54
-O1	2.33	2.38	2.12
-O2	2.34	2.40	2.25
-O3	2.32	2.42	4.13

Table VI. Evaluation of 3.3 and 3.4 LLVM Versions on the FDTD Codelet using *-O2*

	Original (e+11 cycles)	Replay (e+11 cycles)	Prediction error (%)
LLVM 3.3	2.34	2.40	2.25
LLVM 3.4	2.03	1.92	5.23

performing four metarepetitions for accuracy, the replay only takes 0.3s. Compared to the original proto-application runtime, 71.1s, the replay is 237 times cheaper.

CERE quick replay performance evaluation makes accessible costly techniques such as iterative compilation or compiler auto-tuning. As a preliminary experiment to demonstrate CERE applicability to compiler auto-tuning, we show that it accurately captures the performance when exploring the Clang default optimization levels. Table V summarizes our results. We see there is a significant performance improvement between the `-O0` and `-O1` optimization levels. This performance jump is accurately captured by CERE replays. This opens the opportunity for larger auto-tuning experiment [Kashnikov et al. 2012].

CERE fast replay could also be used to evaluate changes in performance across LLVM compiler versions. The idea is that a codelet captured in LLVM 3.3, can be replayed with a later version of the compiler. Currently in CERE, for this to work the IR must be compatible between the capture and replay LLVM versions. Backwards compatibility of the IR is generally possible between minor LLVM versions, but not a strong guarantee of the LLVM project. We captured the FDTD codelet using LLVM 3.3 and replayed it using both LLVM 3.3 and LLVM 3.4. Table VI shows that CERE replay accurately predicts the real execution times achieved when compiling the FDTD proto-application with the two compiler versions.

CERE codelets offer an alternative to the traditional benchmarks or handcrafted microbenchmarks. It allows one to generate small and fast representative performance test cases.

5. COMPARISON TO RELATED WORK

Our work builds upon two different lines of research: code isolation and sampled simulation of programs. Both share the same objective: accelerating performance evaluation of programs. Code isolation extracts pieces of a program as stand-alone codelets, whereas sampled simulation uses a hardware simulator to replay a small set of representative phases in a program.

A first set of papers [Lee and Hall 2005; Petit et al. 2006; Liao et al. 2010; Akel et al. 2013] study code isolation. Lee and Hall [2005] introduce the concept of code isolation for debugging and iterative performance tuning. Their tool, Code Isolator, leverages the Stanford SUIF compiler to outline and generate codelets. They use Code Isolator

Table VII. Feature Comparison of Code Isolation Tools

	CERE	Code Isolator	Astex	Codelet Finder	SimPoint
Support					
Language	C(++), Fortran, ...	Fortran	C, Fortran	C(++), Fortran	assembly
Extraction	IR	source	source	source	assembly
Indirections	yes	no	no	yes	yes
Replay					
Simulator	yes	yes	yes	yes	yes
Hardware	yes	yes	yes	yes	no
Reduction					
Capture size	reduced	reduced	reduced	full	-
Working set	yes	manual	manual	manual	yes
Codelet	yes	no	no	no	yes

on a finite element application, LS-DYNA, to quickly evaluate the L1 cache misses of the hotspots. Petit et al. [2006] and Liao et al. [2010] use code isolation for automatic kernel tuning and specialization. The tool developed by Petit and Bodin [2010], Astex, uses code isolation to accelerate and facilitate value profiling and code specialization for speculative execution. Akel et al. [2013] evaluate the Codelet Finder tool, developed by Caps Entreprises [CAPS 2013] and study under which conditions codelets preserve the performance characteristics of the original programs.

Sampled simulation identifies and clusters similar program phases to reduce simulation time. Lafage and Seznec [2001] propose a method to find slices of a program that are representative for data cache simulation. It uses hierarchical clustering on two metrics: memory spatial locality and memory temporal locality. SimPoint [Sherwood et al. 2001, 2002] identifies similar program phases by comparing Basic Block Vectors (BBV). Phases are samples of 100M instructions. Simpoint reduces simulation time by removing repeated phases. BBV are program dependent, therefore SimPoint cannot use representatives of one program to predict another. Eeckhout et al. [2005] extend SimPoint by matching interapplication phases using microarchitecture-independent features. SimPoint and its extensions are similar to our work in that they extract representative phases from an application. But SimPoint must be used in a simulator, whereas our method is more versatile since the IR codelets can be recompiled and retargeted and run both on simulators and on real hardware.

Breughe and Eeckhout [2013] propose multiple techniques to choose representative working sets for microprocessor design space exploration. CERE achieves a similar goal through invocation representative selection. The difference is that CERE operates at the codelet level, whereas Breughe and Eeckhout [2013] operate at the application level.

CERE page tracing warmup requires tracing access to memory. Recently Payer et al. [2013] present a very low overhead memory tracing technique, which could be leveraged in CERE to reduce the capture cost. Unfortunately, the technique is only applicable to 32-bit binaries.

Table VII compares the features of the main code isolation tools on multiple criteria. First we compare the supported input languages, the isolation level and the support of indirect memory accesses. Second we consider if the tool allows replay on real hardware or is tied to a simulator. Finally, we examine whether the tool attempts to reduce the capture size, the number of working sets, or the number of representative codelets.

6. CONCLUSION

In this article we present CERE, an LLVM-based codelet extractor and replay framework. CERE finds and extracts the hotspots of an application as *codelets*. Codelets can be modified, compiled, run, and measured independently from the original application.

Code isolation reduces benchmarking cost and allows piecewise optimization. We demonstrate two interesting trade-offs: one about using IR level extraction, one using page-granularity memory capture. Results demonstrate the validity, the portability, and the applicability of our solution.

While limiting the benchmark and working set size, our codelet model authorizes better optimization iteration for the user or autotuning tools. Furthermore, as a lot of effort has been put on the software reliability, CERE is a useful basis to build new codelet applications. We are currently exploring PCERE [Popov et al. 2015], a parallel extension of CERE that is able to capture and replay OpenMP parallel regions.

CERE will soon be released at http://benchmark-subsetting.github.io/cere under an open-source license. Automatically generated reports from our experiments with NAS and SPEC benchmarks are also available at the previously mentioned address.

REFERENCES

Chadi Akel, Yuriy Kashnikov, Pablo de Oliveira Castro, and William Jalby. 2013. Is source-code isolation viable for performance characterization? In *Proceedings of the 2013 42nd International Conference on Parallel Processing Workshops (ICPPW'13)*. IEEE.

A. Alexandrescu. 2010. *The D Programming Language*. Pearson Education.

David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Russell L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, and others. 1991. The NAS parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing'91)*. IEEE, 158–165.

Edip Baysal. 1983. Reverse time migration. *Geophysics* 48, 11 (Nov. 1983), 1514. DOI:http://dx.doi.org/10.1190/1.1441434

Maximilien B. Breughe and Lieven Eeckhout. 2013. Selecting representative benchmark inputs for exploring microprocessor design spaces. *ACM Trans. Architec. Code Optim.* 10, 4 (2013), 37.

Ariel N. Burton and Paul H. J. Kelly. 2006. Performance prediction of paging workloads using lightweight tracing. *Future Gen. Comput. Syst.* 22, 7 (2006), 784–793.

Brad Calder, Peter Feller, and Alan Eustace. 1997. Value profiling. In *Proceedings of the 1997 30th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 259–269.

CAPS. 2013. Codelet Finder. Retrieved from http://www.caps-entreprise.com/.

John Cavazos, Christophe Dubach, Felix Agakov, Edwin Bonilla, Michael F. P. O'Boyle, Grigori Fursin, and Olivier Temam. 2006. Automatic performance model construction for the fast software exploration of new hardware designs. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*. ACM, New York, NY, 24–34.

Yang Chen, Yuanjie Huang, Lieven Eeckhout, Grigori Fursin, Liang Peng, Olivier Temam, and Chengyong Wu. 2010. Evaluating iterative optimization across 1000 data sets. In *Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10)*.

Thomas M. Conte, Mary Ann Hirsch, and Kishore N. Menezes. 1996. Reducing state loss for effective trace sampling of superscalar processors. In *Proceedings of the 1996 IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1996 (ICCD'96)*. IEEE, 468–477.

Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. 2014. Fine-grained benchmark subsetting for system selection. In *Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, New York, NY, 132.

Pablo de Oliveira Castro, Eric Petit, Asma Farjallah, and William Jalby. 2013. Adaptive sampling for performance characterization of application kernels. *Concurrency and Computation: Practice and Experience*. DOI:http://dx.doi.org/10.1002/cpe.3097

Lamia Djoudi, Denis Barthou, Patrick Carribault, Christophe Lemuet, Jean-Thomas Acquaviva, and William Jalby. 2005. Maqao: Modular assembler quality analyzer and optimizer for Itanium 2. In *Proceeding of the 4th Workshop on EPIC Architectures and Compiler Technology, San Jose*.

Jason Duell. 2005. The design and implementation of Berkeley Lab's Linux checkpoint/restart. *Lawrence Berkeley National Laboratory*.

Lieven Eeckhout, John Sampson, and Brad Calder. 2005. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *Proceedings of the 2005 IEEE International Workload Characterization Symposium*. IEEE, 2–12.

gperftools v2.2.1. Google Performance Tools. Retrieved from http://code.google.com/p/gperftools.

Xiaofeng Gao, Michael Laurenzano, Beth Simon, and Allan Snavely. 2005. Reducing overheads for acquiring dynamic memory traces. In *Proceedings of the 2005 IEEE International Workload Characterization Symposium*. IEEE, 46–55.

Christopher Haine, Olivier Aumage, Enguerrand Petit, and Denis Barthou. 2014. Exploring and evaluating array layout restructuration for SIMDization. In *Proceedings of the 27th International Conference on Languages and Compilers for Parallel Computing (LCPC'14)*.

John W. Haskins Jr and Kevin Skadron. 2003. Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'03)*. IEEE, 195–203.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Comput. Architec. News* 34, 4 (2006), 1–17.

Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *ACM SIGARCH Comput. Architec. News* 38, 3 (June 2010), 280–289.

Kenneth Hoste and Lieven Eeckhout. 2006. Comparing benchmarks using key microarchitecture-independent characteristics. In *Proceedings of the 2006 IEEE International Symposium on Workload Characterization*. IEEE, 83–92.

Kenneth Hoste and Lieven Eeckhout. 2007. Microarchitecture-independent workload characterization. *Micro, IEEE* 27, 3 (2007), 63–72.

Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. 2006. Performance prediction based on inherent program similarity. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. ACM, New York, NY, 114–122.

Yuriy Kashnikov, Jean Christophe Beyler, and William Jalby. 2012. Compiler optimizations: Machine Learning versus O3. In *Proceedings of the 25th International Conference on Languages and Compilers for Parallel Computing (LCPC'12)*.

Yuriy Kashnikov, Pablo de Oliveira Castro, Emmanuel Oseret, and William Jalby. 2013. Evaluating architecture and compiler design through static loop analysis. In *Proceedings of the 2013 International Conference on High Performance Computing and Simulation (HPCS'13)*. IEEE, 535–544.

Leonard Kaufman and Peter J. Rousseeuw. 2009. *Finding Groups in Data: An Introduction to Cluster Analysis*. Vol. 344. John Wiley & Sons.

Richard E. Kessler, Mark D. Hill, and David A. Wood. 1994. A comparison of trace-sampling techniques for multi-megabyte caches. *IEEE Trans. Comput.* 43, 6 (1994), 664–675.

Minhaj Ahmad Khan, H.-P. Charles, and Denis Barthou. 2008. An effective automated approach to specialization of code. In *Languages and Compilers for Parallel Computing*. Springer, 308–322.

Donald E. Knuth. 1971. An empirical study of Fortran programs. *Softw.: Pract. Exper.* 1, 2 (1971), 105–133.

Thierry Lafage and André Seznec. 2001. Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream. *Workload Characterization of Emerging Computer Applications*. Springer, 145–163.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization, 2004 (CGO 2004)*. IEEE, 75–86.

Yoon-Ju Lee and Mary Hall. 2005. A code isolator: Isolating code fragments from large programs. In *Languages and Compilers for High Performance Computing*. Springer, 164–178.

Chunhua Liao, Daniel J. Quinlan, Richard Vuduc, and Thomas Panas. 2010. Effective source-to-source outlining to support whole program empirical optimization. In *Languages and Compilers for Parallel Computing*. Springer, 308–322.

Gabriel Marin and John Mellor-Crummey. 2004. Cross-architecture performance predictions for scientific applications using parameterized models. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 32. ACM, New York, NY, 2–13.

Dmitry Mikushin, Nikolay Likhogrud, Eddy Zheng Zhang, and Christopher Bergström. 2013. *KernelGen—The design and Implementation of a Next Generation Compiler Platform for Accelerating Numerical Models on GPUs*. Technical Report 2013/02. University of Lugano, July 2013. Retrieved from http://www.old. inf. usi. ch/file/pub/75/tech_report2013. pdf.

Zhelong Pan and Rudolf Eigenmann. 2006. Fast, automatic, procedure-level performance tuning. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*. ACM Press, New York, NY, 173–181.

Mathias Payer, Enrico Kravina, and Thomas R. Gross. 2013. Lightweight memory tracing. In *Proceedings of the USENIX Annual Technical Conference*. 115–126.

Eric Petit and François Bodin. 2010. Code-Partitioning for a Concise Characterization of Programs for Decoupled Code Tuning. Retrieved from http://hal.archives-ouvertes.fr/hal-00460897.

Eric Petit, Pablo de Oliveira Castro, Tarek Menour, Bettina Krammer, and William Jalby. 2012. Computing-kernels performance prediction using dataflow analysis and microbenchmarking. In *Proceedings of Compilers for Parallel Computers Workshop (CPC'12)*.

Eric Petit, Guillaume Papaure, and François Bodin. 2006. Astex: A hot path based thread extractor for distributed memory system on a chip. In *Proceedings of Compilers for Parallel Computers Workshop (CPC 2006)*.

Aashish Phansalkar, Ajay Joshi, and Lizy K. John. 2007. Analysis of redundancy and application balance in the SPEC CPU2006 benchmark suite. In *ACM SIGARCH Comput. Architec. News*, 35, 2 (May 2007), 412–423.

Mihail Popov, Chadi Akel, Florent Conti, William Jalby, and Pablo de Oliveira Castro. 2015. PCERE: Fine-grained parallel benchmark decomposition for scalability prediction. In *Proceedings of the 29th IEEE International Parallel and Distributed Processing Symposium (IPDPS'15) (to appear)*. IEEE.

D Sands. 2009. Reimplementing llvm-gcc as a gcc plugin. In *Third Annual LLVM Developers Meeting*.

Timothy Sherwood, Erez Perelman, and Brad Calder. 2001. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 3–14.

Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. 2002. Automatically characterizing large scale program behavior. In *ACM SIGARCH Comput. Architec. News* 30, 5 (Dec. 2002), 45–57.

Jan Treibig, Georg Hager, and Gerhard Wellein. 2010. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of the 2010 39th International Conference on Parallel Processing Workshops (ICPPW'10)*. IEEE, 207–216.

Received October 2014; revised January 2015; accepted January 2015