PCERE: Fine-grained Parallel Benchmark Decomposition for Scalability Prediction

Mihail Popov, Chadi Akel, Florent Conti, William Jalby, Pablo de Oliveira Castro
Université de Versailles Saint-Quentin-en-Yvelines
Exascale Computing Research
Versailles, France
{mihail.popov,florent.conti,william.jalby,pablo.oliveira}@uvsq.fr
chadi.akel@exascale-computing.eu

***Abstract*—Evaluating the strong scalability of OpenMP applications is a costly and time-consuming process. It traditionally requires executing the whole application multiple times with different number of threads. We propose the Parallel Codelet Extractor and REplayer (PCERE), a tool to reduce the cost of scalability evaluation. PCERE decomposes applications into small pieces called *codelets*: each codelet maps to an OpenMP parallel region and can be replayed as a standalone program. To accelerate scalability prediction, PCERE replays codelets while varying the number of threads. Prediction speedup comes from two key ideas. First, the number of invocations during replay can be significantly reduced. Invocations that have the same performance are grouped together and a single representative is replayed. Second, sequential parts of the programs do not need to be replayed for each different thread configuration. PCERE codelets can be captured once and replayed accurately on multiple architectures, enabling cross-architecture parallel performance prediction. We evaluate PCERE on a C version of the NAS 3.0 Parallel Benchmarks (NPB). We achieve an average speed-up of 25 $\times$ on evaluating OpenMP applications scalability with an average error of 4.9% (median error of 1.7%).**

***Keywords*—OpenMP applications; program replay; checkpoint restart; parallel code isolation; scalability prediction; cross-architecture performance prediction**

I. Introduction

Parallel applications are costly to measure and optimize. In this paper, we propose a decomposition technique that partitions OpenMP applications into small pieces called *codelets* and uses them to accelerate the benchmarking. Each independent parallel region is extracted as a codelet that can be replayed as a standalone program. Instead of studying the whole application, we focus on each codelet separately.

We propose PCERE, an Intermediate Representation (IR) level parallel code extractor based on the Low Level Virtual Machine (LLVM) [1]. PCERE is an OpenMP extension of the Codelet Extractor and REplayer (CERE) [2]. PCERE automatically isolates the parallel regions of an OpenMP application and allows each one to be replayed separately.

Benchmarking isolated parallel regions instead of whole applications is attractive because the user can concentrate on each codelet separately, with a reduced build and run cost. Codelets can be individually modified to evaluate the payoff of new optimizations or runtime parameters such as the number of threads or the thread affinity. Different codelets may expose different performance bottlenecks, and react differently to optimizations. Isolating codelets allows tuning performance at a fine-grain level.

PCERE codelets can also be used for fast parallel performance prediction. We propose a model that predicts strong scalability of parallel regions and applications in average 25 times faster than by running the original program. This speedup is based on the two following key insights:

- A single OpenMP parallel region may be executed multiple times in an application lifetime. Invocations sharing similar execution contexts have the same performance. Therefore, a single invocation replay is sufficient to characterize the region execution time. Codelets exploit this idea as they can be directly replayed as few times as we want. This method is usable only if the execution time of a parallel region remains the same over different invocations.
- When varying the number of threads, performance of sequential regions is not strongly impacted. With codelets, scalability can be computed by only replaying parallel regions; there is no need to replay the sequential parts with different number of threads.

To predict the strong thread scalability of an application, PCERE partitions it into codelets which are then replayed with varying number of threads. A model based on Amdahl's law predicts the whole application scalability.

Parallel codelets can be replayed on micro-architectures different from the one in which they were captured. Therefore PCERE is able to predict the scalability on different target micro-architectures.

Codelet based performance evaluation is only viable if the codelets faithfully capture the original application behavior. In this paper we show that PCERE faithfully replays the C-only version of the NAS Parallel Benchmarks [3] on different architectures with different number of threads.

The current PCERE version only supports C or C++ OpenMP applications. Most of the NPB 3.0 are written in Fortran. To evaluate PCERE we propose an unofficial C version of the NPB 3.0 OpenMP that is built upon the NPB OMP 2.3 C version from the Omni Compiler Project [4].

The contributions of this paper are:

- PCERE: an open-source framework that extracts and replays parallel regions codelets (Section III)

- A pure C version of the NPB OpenMP (Section IV-A)
- A systematic evaluation of PCERE replay accuracy on the NPB OpenMP on three different architectures with varying number of threads (Section IV-B).
- A fast scalability prediction model based on parallel codelets (Section V).

II. Background

This section discusses the relevant background for parallel code extraction. First, we present the concept of code isolation. Second, we show how compilers turn OpenMP directives into parallel code. Third, we discuss how to apply code isolation to OpenMP programs.

A. Code Isolation

Code isolation was proposed originally by Lee et al [5] as a method to quickly debug and tune large applications. Usually, in scientific applications, the hotspots represent a small fraction of the total source lines [6]. Code isolation finds and extracts the hotspots of an application as standalone *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 contents to achieve realistic replays.

Code isolation has been used to tune compiler options [7] or to accelerate architecture selection [8]. Nevertheless, most of the code isolation frameworks [5], [9], [10], [11], [12] target sequential programs. PCERE extends the concept of code isolation to multi-threaded OpenMP programs. Parallel code isolation is discussed in section VI-A.

B. OpenMP Compiler Support

In OpenMP programs, the application concurrency is described through a set of compiler directives and library calls. For instance, a parallel region can be declared using the directive `#pragma omp parallel`. Figure 1 shows a simple C OpenMP program where each thread prints its thread identifier.

PCERE is based on the LLVM compiler infrastructure [1], which includes preliminary support for OpenMP 4.0 [13] since version 3.4. Compiling and linking an OpenMP application under LLVM requires the following steps:

1) **Translation to IR** The Clang front-end transforms C or C++ code into LLVM IR.
2) **Code Optimization** LLVM optimizations passes are applied to the IR.
3) **Object generation and linking** The IR is transformed into object code and linked with the Intel/LLVM OpenMP runtime library [14].

In most compilers, including GCC and LLVM, parallel directives are expanded in the front-end before doing any code optimization [15]. Usually, the first step in OpenMP expansion is *outlining* parallel regions. To outline a region the compiler moves the region code inside a separate function. The compiler preserves data dependencies by passing live-in and live-out values through the outlined function arguments. Then the original region is replaced by a call that spawns multiple threads running the outlined function.

Figure 1 shows how LLVM expands a simple OpenMP directive and the IR code it produces. LLVM outlines the region code in a `microtask` function. `kmpc fork`, an OpenMP Runtime library function, spawns a pool of threads. Then, every thread runs the outlined `microtask` function which describes the region parallel work.

C. Partitioning OpenMP Programs into Codelets

Parallel codelets should satisfy three important properties. First, each codelet must capture a specific region of code in the original application. Second, it should be possible to change the number of threads and other runtime parameters at replay and it should be possible to replay codelets across different architectures. Third, the set of extracted codelets must faithfully capture the behavior of the original application so that it can be used as a proxy for measuring performance and scalability. In particular, each codelet replay must be deterministic: different replays of the same codelet should execute the same code and have the same performance.

Multi-threaded execution is a well known source of indeterminism: race conditions and synchronization delays between threads may change the order of the operations from one execution to the next. In particular, when multiple threads are running, each one may be executing a different region of code. This makes it difficult to isolate a particular region of code.

To avoid thread indeterminism issues, PCERE codelets start at the beginning of a parallel region and finish at the end of the region. Indeed, the beginning of an OpenMP region is a global synchronization point where all threads positions in the program are known. Capturing codelets at the start of the region has another advantage: it enables changing the number of threads at replay. Indeed, the capture happens just before the call to `kmpc fork` that decides how many threads are spawned.

Another advantage of capturing codelets just before a call to `kmpc fork` is that it simplifies codelet portability. Traditional checkpoint techniques save a full dump of the memory and of the register banks, including the Program Counter. This approach requires that the replayed code keeps the same code layout and uses exactly the same registers as during the capture. It limits codelet portability to architectures sharing the same register layout and does not enable recompiling and replaying the code on a different architecture.

Because codelet capture happens just before a call to `kmpc fork`, the application binary interface guarantees that the codelet working set is either in memory or is passed as arguments to the outlined function. This simplifies 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 allows 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 have similar memory layout, such as word size and endianness. Our tests have shown that our codelet replayer allows to recompile changing optimization flags (capturing on `-O0` but replaying on `-O3`) or changing

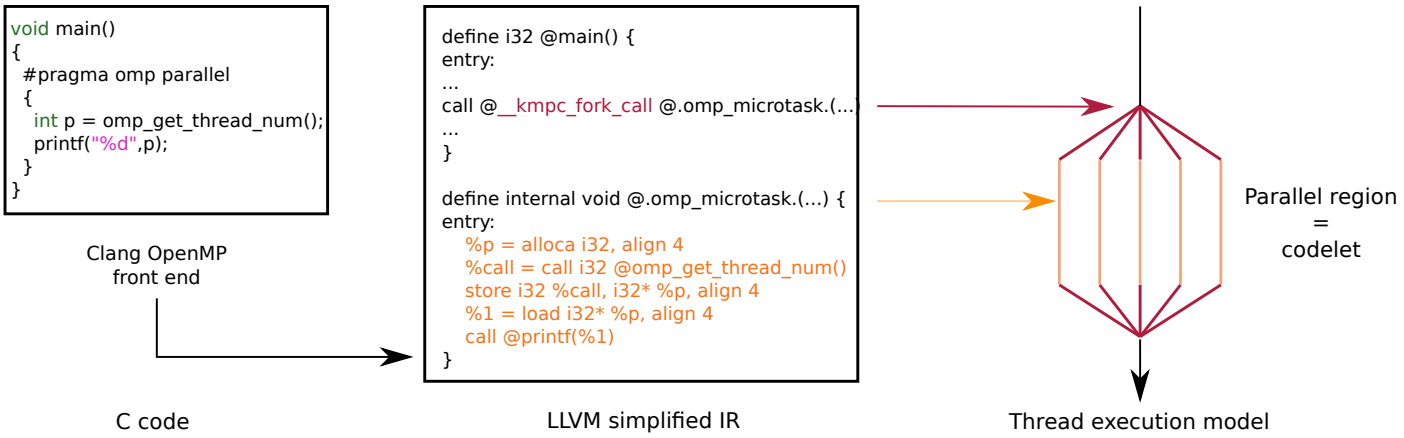

Fig. 1. Clang outlines each C parallel region as an independent IR function:`omp_microtask`. The call to `kmpc_fork` spawns a pool of threads that runs the outlined microtask.

micro-architectures (capturing on Nehalem and replaying on Sandy Bridge, Core2, or Atom).

III. PCERE: Parallel Codelet Extraction and Replay

This section gives an overview of PCERE architecture and details the different technical challenges of codelet extraction and replay.

A. Workflow Overview

Parallel code isolation consists of two main steps: *capture* and *replay*. During the capture, the original program is instrumented with calls to our memory capture library. The execution state is captured at the start of each parallel region. During the replay, the user selects a particular region to replay. PCERE generates a standalone codelet that restores the execution context and jumps to the region of interest.

Figure 2 presents how a parallel region is captured as a codelet and replayed. Both capture and replay require special compilation passes to instrument the code with calls to the capture and restore libraries, and to isolate parallel regions. PCERE leverages the LLVM compiler framework to implement the instrumentation and isolation passes. LLVM provides a rich API for manipulating the IR code, which greatly simplifies the process. PCERE instrumentation and extraction passes operate after OpenMP directive outlining but before the IR optimization passses.

The capture requires executing the application once to get the execution context of all the parallel regions. If the user wants to measure an application on a single architecture with a fixed number of threads, extracting and replaying the codelets does not pay off. It is quicker and more accurate to fully measure the benchmark. But, if the user is interested in comparing the performance of different architectures and thread configurations, the codelet approach is quickly amortized because codelets are only extracted once but replayed many times. The overhead of codelet capture is discussed in section V-A.

B. Capturing and Restoring the Execution Context

Before replaying a codelet, the memory state from the original execution must be restored. This ensures that the replay will be equivalent to the original run, even for data dependent branching code. Three aspects need to be considered at replay:

1) the working set of the parallel region must be restored,
2) the cache must be warmed to avoid cold-start bias [16],
3) the different data sets which induce performance variations across invocations must be considered.

Checkpointing the original memory state must be done before reaching the parallel region. In PCERE the memory is captured just before the `kmpc fork` call. Capturing before the fork allows to change at replay both the number of threads and the architecture. The capture of the thread stack and the Thread Local Storage (TLS) are also simplified as they are handled by the `kmpc fork`.

In the rest of this section, we describe how the working set and cache state are captured and restored, and how PCERE handles the capture of regions where the performance is different across working sets.

a) Working Set Capture:
To capture the working set, PCERE takes a full snapshot of the original application address space. The application is frozen using the `ptrace` system call, then a helper process dumps the memory contents to disk, and returns the control to the original application. Codelet Finder [11], [10], a sequential code isolator tool, uses a similar technique. Full memory dumps are large, but have the advantage of perfectly capturing the memory layout, handling pointer aliasing, and preserving the relative alignment and the offsets among data structures.

When replaying a codelet, PCERE instruments the `main` function so that it immediately jumps to a special `run_codelet` procedure. `run_codelet` restores the memory working state and runs the original parallel region by directly calling `kmpc fork` on the outlined function.

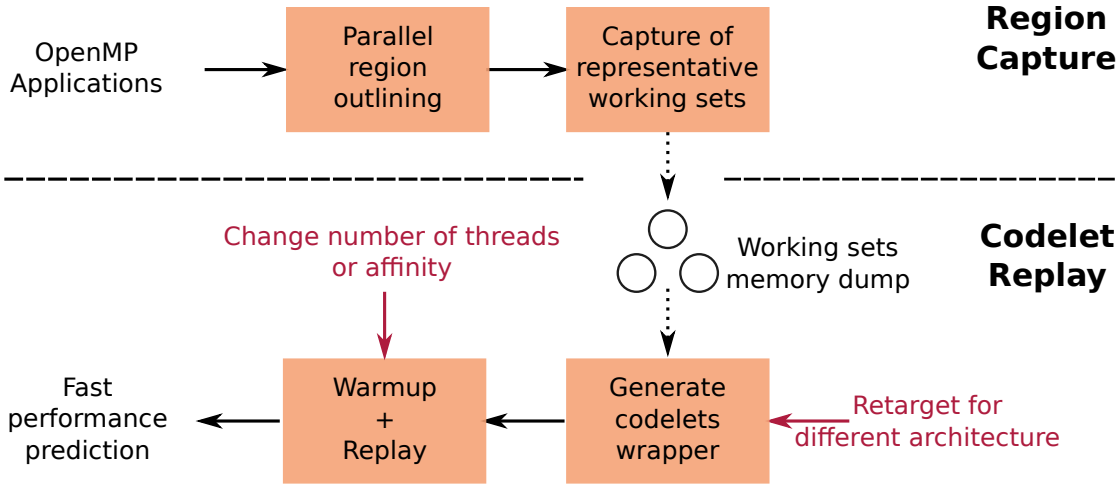

Fig. 2. Codelet capture and replay workflow

b) Cache warmup:
To faithfully capture the performance of the original region it is necessary to warm up the system to match the original context as much as possible. Two main approaches have been proposed in the literature. The first approach is to warm up the cache by running a few warmup executions of the codelet itself [9], [11], [10]. It is an optimistic heuristic that assumes that the codelet working set is hot in the original run.

The second approach warms up the cache by replaying the history of the memory accesses in a simulator [17] or using a warmup routine [5]. These techniques are more accurate but require tracing the memory which is costly and incurs significant slowdowns [18].

The current version of PCERE uses the first approach because it is simpler to implement and has a much lower overhead. Our warmup heuristic distinguishes two types of codelets: frequently-called codelets that have more than four invocations in the original application and infrequently-called codelets.

We assume that the working set of frequently-called codelets is hot in the original application. The rationale is that the first invocations in the original program have warmed up the region working set. Therefore, for frequently-called codelets, our warmup strategy runs the codelet four times before replaying it for real. This number was determined empirically.

For infrequently-called codelets, we assume that the working set is cold in the original run. Therefore we replay them exactly as many times as in the original application.

Our experiments in section IV show that the optimistic warmup is enough to accurately replay most of the NAS parallel benchmark on three different architectures. Yet, in future versions of PCERE we would like to offer more realistic warmup strategies as discussed in section VII.

c) Codelets with different working sets: Most of the codelets have a constant performance across invocations. Therefore, a single working state snapshot is sufficient to accurately replay all the invocations. If performance varies across invocations, this simple capture strategy is insufficient.

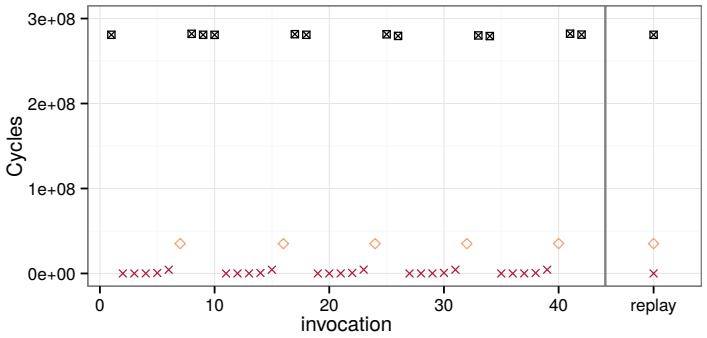

Fig. 3. MG `resid` execution time over the different invocations replayed with 4 threads on Nehalem. `resid` has three distinct performance classes. To faithfully replay the parallel region, PCERE captures one working set per performance class. Here the first, second, and seventh invocations were captured and replayed.

Figure 3 presents the execution time across the different invocations of `resid` which represents more than 50% of MG execution time and is invoked 42 times. We observe that `resid` has three different performance behaviors. Each performance class corresponds to a different working set. Performance classes can be automatically detected with a clustering algorithm such as CLARA [19]. When a codelet has different performance classes, we select a representative invocation per class. During capture, PCERE saves a separate memory dump for each representative invocation. The right box in figure 3 shows the replay time of the three selected representatives in `resid`. To predict the codelet whole replay time, PCERE adds each representative execution time weighted by the number of invocations that fall in its performance class.

C. Lock Support

To fully support replay of codelets using OpenMP lock primitives, a special *lock capture* step is required.

OpenMP uses futex (fast userspace mutexes) calls to implement the lock support on Linux. Each futex requires a kernel space wait queue. System calls are used to request operations on the wait queue from user space. Our memory capture only

TABLE I. THE TEST ARCHITECTURES USED IN THIS PAPER.

	Core2	Nehalem	Sandy Bridge
CPU	E7500	Xeon E5620	E5
Frequency (GHz)	2.93	2.40	2.7
Sockets	1	2	2
Cores per socket	2	4	8
Threads per core	1	1	2
L1 cache (KB)	32	32	32
L2 cache (KB)	3MB	256	256
L3 cache (MB)	-	12	20
Ram (GB)	4	24	64

saves the user space process memory, therefore it does not preserve the state of the futex wait queue.

To properly support OpenMP locks, we need a special lock capture step that detects all the locks accessed by a codelet. This is achieved by intercepting calls to the lock OpenMP library during capture.

Before replaying the codelet, the replay wrapper takes care to properly initialize all the required locks in kernel space.

IV. PCERE EVALUATION

This section evaluates PCERE replay accuracy. The evaluation is done on an unofficial C version of the NAS Parallel Benchmarks OpenMP.

A. Experimental Setup

The NAS Parallel Benchmarks (NPB) are an established OpenMP benchmark suite and a good target to evaluate PCERE prediction accuracy. Yet, most of the NPB are written in Fortran and PCERE is built upon Clang which only supports C and C++ OpenMP programs.

The Omni Compiler Project (OCP) [4] maintains an unofficial C version of NPB 2.3. Unfortunately, it is based on an outdated NPB 2.3 which was released in 1997. In particular, in the NPB 2.3 version the OpenMP parallelism is coarse grained: most of the benchmarks only have one huge parallel region. This makes them a poor choice to evaluate PCERE. To overcome this problem, we have updated the OCP unofficial NAS benchmarks so that they mimic the structure of the NPB 3.0 OpenMP official version. This effort involved carefully porting the changes in the official Fortran version to the unofficial C version. Our 3.0 C version of NPB is publicly available at http://benchmark-subsetting.github.io/cNPB.

The CG benchmark was slightly modified to overcome a bug in LLVM OpenMP frontend. A global barrier was added after `omp for reduction` clauses. According to the OpenMP specification, this synchronization is implicit and mandatory, but the current LLVM OpenMP implementation does not honor it. This issue has since been reported and corrected [20].

In this paper all the benchmarks were compiled at the `-O3` optimization level using Clang 3.4 and linked against the Intel/LLVM runtime. The benchmarks were run using the class A working set sizes. The thread affinity was set to *Scatter* with a static work scheduling.

Table I presents our test architectures. They belong to three different Intel CPU generations (Core2 Duo, Nehalem, and Sandy Bridge) and possess quite distinct memory hierarchies. These machines were selected to validate that PCERE replay process is portable across micro-architectures.

B. Replay Accuracy

The codelet based benchmarking process is only viable if the codelets faithfully capture the original application behavior. In this section we evaluate the *accuracy* of the codelet replay.

Accuracy is defined as the relative difference between the *in vivo* and *in vitro* execution times of a codelet. The in vivo time is the time spent inside the codelet parallel region in the original application. The in vitro time is the time measured during the codelet standalone replay.

To evaluate accuracy, we extracted the full set of codelets from all the NPB applications. As discussed in section II-C, the extracted codelet set maps exactly to the parallel regions of the original OpenMP application. We reduced this full set, by removing the codelets that represent less than 5% of the original execution time. Originally 59 codelets were captured, and after filtering only 25 were kept. Then we measured the in vivo and in vitro execution time for each codelet, and computed the replay accuracy.

For the NPB benchmarks, the number of threads used during capture had no impact on replay: the number of threads can be freely changed at replay. Therefore, the thread number during capture can be chosen arbitrarily. In our experiments we ran capture using a single thread run.

Table II summarizes the replay accuracy over the NPB codelet set measured on Nehalem. PCERE faithfully replays most of the parallel regions: the in vivo and in vitro performance is close. A single working set was enough to faithfully replay the benchmarks parallel regions except for MG for which we needed to extract multiple working sets to accurately replay its in vivo execution time.

Only two codelets are misreplayed: SP `xsolve` and CG `residual norm`. In both benchmarks, the error is due to cache state differences between the in vivo and the in vitro executions. Our optimistic warmup strategy is not accurate enough for these two benchmarks. For instance, CG `residual norm` working set is cold in the original run but incorrectly warmed-up during replay. Section VII briefly discusses how the warmup strategy could be improved.

V. PCERE SCALABILITY PREDICTION

Through fast codelet replay, PCERE is able to quickly estimate the strong scalability of a parallel application. This section evaluates the scalability prediction on the NPB 3.0. Using PCERE scalability evaluation is 25 $\times$ cheaper than measuring the original NPB and maintains a low 1.7% median prediction error.

A. Prediction Model

Traditionally the scalability of an application is evaluated by plotting the application execution time against the number of parallel threads. This requires measuring one application run for each thread configuration.

TABLE II. CODELET REPLAY ACCURACY

Benchmark / Parallel region	Threads 1	2	4	8	weight %
CG					
conj grad iteration loop	11.18	8.05	0.23	2.66	95.8
conj grad residual norm	12.26	11.27	31.756	3.08	66.4
MG					
resid	0.68	0.03	1.19	0.46	53.5
psinv	1.42	1.07	0.57	1.43	22.8
interp	8.76	8.09	8.18	1.75	07.2
rprj3	1.31	3.15	2.09	7.27	05.8
norm2u3	0.12	0.89	1.53	0.14	05.6
zero3	4.03	4.86	5.46	10.50	05.4
EP					
main	0.01	2.00	0.10	1.36	99.99
IS					
main random generator	0.22	0.50	0.52	3.76	80.6
rank	0.28	0.84	1.64	5.30	39.9
SP					
x solve	23.27	17.37	10.59	4.37	33.4
z solve	2.64	1.30	1.86	2.07	30.9
y solve	8.46	5.55	6.66	5.61	32.7
compute rhs	1.61	1.60	2.1	0.57	26.8
BT					
z solve	0.14	3.52	2.5	10.80	33.6
y solve	0.20	3.76	1.82	6.25	32.3
x solve	0.33	0.26	4.16	11.13	30.5
compute rhs	0.84	0.77	0.08	2.34	11.1
FT					
cffts2	0.38	2.58	3.29	3.20	31.3
cffts3	5.12	4.71	5.06	5.27	30.9
cffts1	1.29	0.99	0.11	0.84	30.7
compute indexmap	3.09	5.20	0.32	3.82	06.6
LU					
ssor iteration	0.03	0.86	0.53	0.03	98.7
rhs	0.89	1.09	0.58	2.01	27.4

The table shows the relative error between the in vivo and in vitro execution time for each NPB codelet over different thread configurations. Measures were performed on Nehalem. The weight % column is the contribution of the region to the total running time (the weight changes across thread configurations, here we consider the maximum).

Amdahl's law stipulates that the execution time $T(n)$ of an application being executed on n threads is,

$$T(n) = T(1) \times (S + \frac{1-S}{n}),$$

where S is the fraction of the algorithm that is strictly serial. In other words, increasing the number of threads will only speedup parallel regions. Our scalability prediction model is built upon this assumption. The invariant sequential regions are measured only once. Thanks to codelets, only the parallel regions are replayed for each different number of threads.

Our prediction method has two steps. First, *initialization* extracts the codelet set and measures the sequential part of each application, T_{seq}. Second, *prediction* replays the codelets with different number of threads to predict the scalability of the original applications.

Let C be the set of all the codelets of an application. Our model estimates the application execution time with m threads as,

$$T_{predicted}(m) = T_{seq} + \sum_{c \in C} T_c^{vitro}(m) \times invoc_c,$$

where $invoc_c$ is the number of invocations of codelet c in the original application, and $T_c^{vitro}(m)$ is the in vitro execution time per invocation measured by replaying codelet c with m threads.

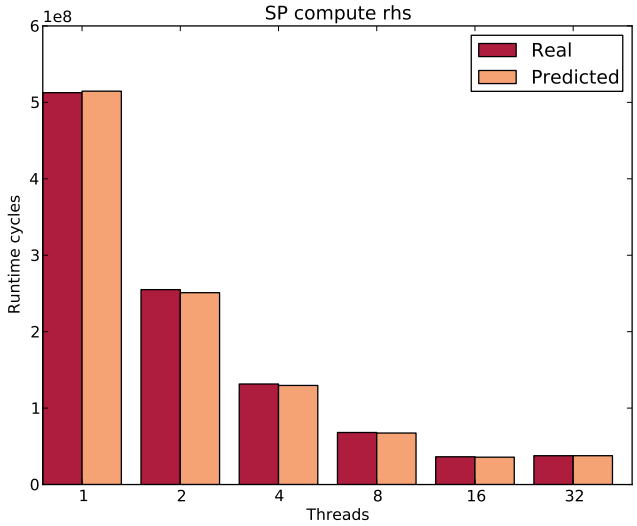

Fig. 4. Real vs. PCERE execution time predictions on Sandy Bridge for the SP compute rhs codelet

PCERE accelerates the benchmarking process because:

1) the sequential part is only measured once during the initialization,
2) codelet replay is faster because the number of invocations of the parallel regions is reduced.

PCERE scalability prediction acceleration is computed as the ratio between the replay cost and the original application execution time. The replay cost is the time needed to replay all of the extracted codelets, it includes warmup time.

We do not take into account the cost of the initialization step into the quantification of PCERE acceleration over the benchmarking process. Indeed, codelets might be reused and the extraction cost is only paid once. Codelets can even be shared among different users, quickly amortizing the initialization cost. For reference, the cost of the initialization step is comparable to a full run of the initial benchmarks, with a small overhead attributable to our additional compiler passes and capture library calls. On the NPB suite BT has the worst initialization overhead: 1.25 $\times$ the original execution time.

The prediction accuracy is quantified for m threads as the relative error between $T_{predicted}(m)$ and $T_{real}(m)$, the real execution time of the application with m threads. The lower this value is, the better our predictions are. The next section evaluates the acceleration and prediction error on the NPB 3.0 on three test architectures.

B. Scalability Prediction Per Architecture

Table III details NPB prediction and benchmarking acceleration over three different architectures. As explained in section IV, codelets were extracted on a single thread.

Overall PCERE scalability prediction is fast and accurate. Table IV summarizes the average accuracy and acceleration over all the NPB. In figure 4 we compare the real and predicted

TABLE IV. AVERAGE SCALABILITY PREDICTION ACCURACY AND ACCELERATION ON NPB

	Core2	Nehalem	Sandy Bridge
Accuracy	1.84%	2.9%	7.4%
Acceleration	25.2	27.4	23.7

Sandy Bridge higher prediction error comes from the inability of PCERE to correctly predict scalability when hyper-threads are used. Not considering hyper-threaded runs lowers Sandy Bridge error to 5.9%.

execution time on SP `compute rhs` codelet which has the biggest execution time per invocation within SP.

Problems are highlighted in gray on table III and explained below:

- EP benchmark shows no acceleration at all with our method. EP sequential part is negligible and its single parallel region is only invoked once. Therefore PCERE replay strategy is not faster than the original run.
- IS has low accuracy on Sandy Bridge. This is because the memory layout in IS depends on the number of threads. A higher number of threads changes the blocking and impacts the execution time of sequential regions as demonstrated in [21]. This violates our base assumption of invariant sequential execution time.
- On Sandy Bridge, replays running with 32 threads show high misprediction errors. The test machine has only 16 physical cores. PCERE warmup and replay strategy is not accurate when hyper-threads are used. We are investigating this issue.

C. Cross Architecture Prediction

PCERE codelets are portable: they can be extracted on an architecture and replayed on another one. Architectures should share the same memory layout, word size and endianness; but the register layout and ISA can be different because our checkpointing happens just before a function call, so only the memory and function arguments need to be captured. Therefore, it is possible to do cross architecture codelet replay and scalability prediction.

To demonstrate cross-architecture portability, we extracted the NPB codelets on Nehalem and replayed them on Sandy Bridge. Table V summarizes the results. Overall accuracy is high, except for CG replays and 32 threads replays.

As before, the misprediction with 32 threads is caused by the inability of PCERE to correctly predict scalability when hyper-threads are used. CG misprediction is caused by our heuristic warmup which, in this case, proves to be insufficient. Indeed our warmup strategy is optimistic: it assumes that the working set is hot in the original execution, which is not true in this scenario. Perspectives for improving the warmup strategy are discussed in section VII.

VI. 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 standalone codelets whereas sampled simulation uses a hardware simulator to replay a small set of representatives phases in a program.

A. Code Isolation

Many sequential code isolation frameworks [5], [9], [10], [11], [2] have been proposed. But to the best of our knowledge, only Liao et al. [22] also study parallel code isolation. Their approach is based on the ROSE [23] compiler infrastructure. Isolation is achieved through a source to source outliner which extracts tunable kernels out of OpenMP programs. They use the isolator to accelerate computation kernel tuning and show how to find the best parameters for the number of OpenMP threads, the schedule policy, and the chunk sizes. Liao et al. isolates parallel `for` loops at the source level. 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, PCERE IR level outlining is simpler because it is done after OpenMP data clauses expansion. Unlike PCERE which is evaluated on all the NPB, Liao et al. outliner is demonstrated on a single OpenMP `for` loop from the SMG2000 benchmark.

Yang et al. [24] proposes a partial execution framework without code isolation based on relative performance between platforms. They manually insert probes around the kernels of the application. The probes allow to stop the execution during replay after a large enough number of invocations has been measured. Like PCERE, this method reduces the number of invocations of the kernels but it is not automatic. The user must hand-tag the computation kernels with source annotations.

B. Sampled Simulation

Sampling techniques identify and cluster similar program phases to reduce simulation time. These techniques are already mature for single threaded applications [25], [26]. Similar phases are usually detected by comparing their Basic Block Vectors (BBV).

Extending sampling techniques to multi-threaded simulations is difficult because of the threads interactions. Wenisch et al. [27] and Van Biesbrouck et al. [28] both propose techniques to accelerate multi-threaded simulations. They both build their model under the assumption that each thread is independent. Therefore, they do not support explicit threads synchronization.

Perelman [29] applies the SimPoint [25] methodology to parallel applications using instruction-based sampling. However, Carlson et al. [30] and Ardestani et al. [31] both show that instructions are not a good proxy for execution time in multi threaded programs. Instead, they propose a time based sampling method. Carlson et al. [30] provide a methodology for sampling multi-threaded workloads with up to a 5.8 $\times$ simulation time reduction over the NPB and Parsec benchmarks with an average absolute error of 3.5%.

Carlson et al. [32] also propose BarrierPoint, a sampling methodology which detects globally synchronizing barriers in

TABLE III. SCALABILITY PREDICTION ON NPB

	CORE2				NEHALEM							
	Accuracy		Acceleration		Accuracy				Acceleration			
Threads	1	2	1	2	1	2	4	8	1	2	4	8
CG	0.25	1.72	28.77	19.55	10.64	5.38	0.38	22.62	40.41	23.07	12.65	8.07
MG	1.94	1.13	0.55	1.14	0.32	0.74	1.25	0.11	1.81	1.94	2.26	2.73
EP	0.11	0.13	1.0	1.0	0.01	0.19	0.1	1.13	1.0	1.0	1.0	0.99
IS	4.16	4.29	1.01	1.23	0.37	1.63	2.08	4.89	1.12	1.12	1.15	1.2
SP	1.22	2.38	92.82	81.58	8.32	5.69	4.49	5.38	103.02	97.52	94.69	92.74
BT	0.87	4.1	36.48	26.41	0.16	3.1	0.59	7.98	38.33	41.05	44.26	49.18
FT	1.26	3.34	1.76	1.22	0.93	0.75	0.05	0.21	1.93	1.96	2.11	2.32
LU	1.72	1.0	54.44	54.2	0.24	1.0	0.63	0.67	52.91	52.53	51.81	50.25

	SANDY BRIDGE											
	Accuracy						Acceleration					
Threads	1	2	4	8	16	32	1	2	4	8	16	32
BT	1.51	2.47	2.64	12.99	18.28	15.66	35.83	39.83	43.86	51.13	55.47	46.99
EP	0.1	0.09	5.06	0.31	2.81	0.7	1.0	1.0	1.05	1.0	1.02	0.98
LU	0.01	2.08	1.96	1.95	0.19	2.43	51.86	50.9	50.67	47.7	45.39	20.78
FT	3.09	1.35	0.91	1.3	0.68	2.52	1.9	1.98	2.17	2.49	2.96	2.77
SP	0.78	0.06	0.14	0.05	3.38	0.67	92.19	88.72	85.89	83.01	75.04	49.21
CG	15.15	3.37	9.51	18.77	13.86	26.62	37.28	17.95	9.64	5.29	4.31	4.82
IS	1.53	6.22	24.83	17.14	30.57	27.57	1.12	1.08	1.61	1.61	1.0	0.99
MG	1.07	6.08	4.24	2.3	4.73	54.24	1.88	1.92	2.32	3.01	3.77	1.7

Overall PCERE accurately predicts the scalability on the three architectures. The average prediction error is 4.9%. The prediction is in average 25 × faster with PCERE. In this experiment a separate initialization step was performed on each architecture. In Sandy Bridge the error is higher on IS and with 32 threads. IS misprediction is due to the fact that changing the number of threads changes the memory layout which impacts the sequential regions violating our model assumptions. Higher misprediction on 32 threads is due to PCERE being less reliable when hyper-threads are used.

TABLE V. CROSS ARCHITECTURE REPLAY ACCURACY

Benchmark	Threads						
Parallel region	1	2	4	8	16	32	weight %
CG							
conj grad iteration loop	7.57	4.44	1.69	1.35	5.44	3.06	95.6
conj grad residual norm	18.86	15.65	6.99	24.3	42.63	36.7	93.5
MG							
resid	1.34	1.64	1.72	1.59	1.03	63.20	56.0
psinv	20.27	3.13	2.09	37.25	10.06	80.41	23.1
zero3	6.98	3.20	8.12	8.23	3.33	71.06	09.2
interp	7.13	6.55	6.22	2.48	7.50	92.03	07.4
rprj3	7.53	6.50	9.86	11.93	15.38	94.72	05.6
EP							
main	0.04	0.10	0.05	0.03	0.26	0.44	99.99
IS							
main random generator	0.63	1.09	0.67	9.59	29.14	1.98	82.4
rank	0.69	0.42	0.24	0.22	1.49	8.24	42.8
SP							
z solve	2.54	0.01	6.02	10.4	3.94	4.77	35.1
x solve	2.80	2.21	4.84	4.25	0.90	9.48	31.9
compute rhs	2.04	1.60	1.30	0.19	1.42	0.53	27.7
y solve	0.51	0.03	5.23	10.94	1.84	3.72	27.1
BT							
z solve	0.30	2.00	1.58	12.51	21.08	24.02	33.7
x solve	0.87	3.97	5.01	12.53	19.78	15.56	33.4
y solve	0.13	1.75	5.58	16.30	22.39	13.05	33.1
compute rhs	0.19	0.99	1.96	1.43	2.30	3.619	09.5
FT							
cffts2	1.66	0.47	1.57	0.80	0.70	0.91	30.5
cffts3	9.49	7.93	7.83	7.58	7.13	2.10	30.4
cffts1	0.03	0.84	1.5	0.08	1.16	0.34	30.2
compute indexmap	0.80	1.39	2.70	0.30	0.83	8.78	10.1
LU							
ssor iteration	1.44	0.01	2.58	2.27	0.32	0.48	76.8
rhs	0.12	0.54	0.26	0.11	0.45	0.73	26.1

In this experiment, codelets were extracted on Nehalem and replayed on Sandy Bridge, enabling quick cross architecture scalability prediction. The table shows the relative error between the in vivo and in vitro execution time for each NPB codelet over different threads configurations. The prediction error is low, except for CG residual norm codelet and 32 threads. CG misprediction is caused by our heuristic warmup which, in this case, proves to be insufficient. Misprediction on 32 threads is again due to PCERE inability to predict thread configurations requiring hyper-threads. The weight % column is the contribution of the region to the total running time (the weight changes across thread configurations, here we consider the maximum).

multi-threaded applications. BarrierPoint estimates total application execution time through detailed simulation of the most representative inter-barrier regions. Regions representativeness is defined with micro-architecture independent information and data signatures. BarrierPoint achieves an average speed up of 24.7 $\times$ over the NPB and Parsec benchmarks with an average error of 0.9%. The speedup is similar to the one achieved by PCERE, yet the accuracy of BarrierPoint is better. Nevertheless, accuracies are not directly comparable since BarrierPoint measures accuracy on a functional simulator whereas PCERE measures accuracy on real hardware.

Sampling techniques are similar to our work in that they extract representative phases from applications and allow accurate replay. Nevertheless, all of these sampling techniques must be used in a simulator, whereas our method is more versatile since it produces IR codelets that can be recompiled and run both on simulators and on real hardware. Another key difference is that BarrierPoint and the other sampling techniques do not allow changing the number of threads at replay. Each thread configuration requires a separate capture. Therefore, unlike PCERE, they cannot be easily used to evaluate parallel scalability.

VII. Limitations and Future Work

Our methodology successfully accelerates performance evaluation of multi-threaded programs on different architectures. In this section we discuss its drawbacks and outline promising future work.

In this paper, PCERE evaluates scalability of the NAS NPB. PCERE evaluation could be extended to real industrial applications.

PCERE fast codelet replay could also be used to accelerate the space exploration of other OpenMP parameters such as the schedule policy, the chunk sizes of parallel regions, or the thread affinity mapping [22]. PCERE could also be used to tune compiler flags for parallel programs [33] since flags can be changed at replay time.

PCERE does not handle extraction of nested parallel regions. This was a deliberate choice that simplifies the implementation of the extraction compiler passes. We hope to remove this limitation in future versions of PCERE.

One of PCERE strong features is allowing change of the number of threads during replay. This is possible for most of the OpenMP applications tested, in particular it is true for all the NPB 3.0. Nevertheless, there are cases in which the replay thread number is constrained by the capture. For instance, `qsomp`, a quick sort benchmark in OMPSCR v2.0 suite [34], can only be replayed on a smaller number of threads than the one used during capture. This is because `qsomp` serial initialization depends on the number of threads. Instead of using the TLS, it manually allocates a reserved slot in a global vector for each thread. If the replay is done with more threads than during capture, there are not enough reserved slots and replay fails. Fortunately, this design pattern is infrequent in OpenMP, which encourages to write thread-number agnostic code. PCERE could automatically detect those problematic cases by tracking calls to `omp_get_max_threads` that are later used as argument to memory allocating functions.

PCERE uses an optimistic warmup strategy to reproduce the original cache state when replaying a region. The main problem of the method is that sometimes its optimistic assumption is not true: the working set is cold in the original program. In those programs, optimistic warmup introduces discrepancies between in vivo and in vitro performance. Future versions of PCERE should offer more realistic warmup techniques, that replay the history of past memory accesses to restore the cache state [16], [35], [36].

Through program similarity analysis, de Oliveira Castro et al. [8] find a minimal set of representative codelets. They cluster together codelets that have the same performance behavior, and keep only one representative copy for each group. Similary, Zhai et al. [37] propose Phantom, a sampling methodology that clusters together similar processes in MPI applications to reduce performance evaluation cost. These benchmark reduction techniques could be extended and applied to PCERE codelets, achieving even higher cost reductions in parallel performance evaluation. However, they require defining a set of program similarity metrics that are relevant in the context of OpenMP parallel regions.

Conclusion

This paper presents a methodology to reduce the cost of strong scalability evaluation. Instead of executing the whole application with different thread configurations, we outline and replay only parallel codelets. Unlike previous work, codelets allow changing the thread number at replay. A single codelet capture can be replayed with different thread configurations.

PCERE reduces the strong scalability evaluation time up to 25 $\times$ with an average prediction error of 4.9%. The prediction method can successfully be used across different architectures. Parallel codelet replay offers other interesting applications such as fast OpenMP runtime parameters auto-tuning.

PCERE will soon be released at http://benchmark-subsetting.github.io/pcere under an open source license. Detailed reports from our experiments with NPB benchmarks are also available at the above address. We propose a C OpenMP version of the 3.0 NAS Parallel Benchmarks. The benchmark suite is available at http://benchmark-subsetting.github.io/cNPB.

Acknowledgments

The authors would like to thank Eric Petit for his insightful comments.

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

References

[1] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*. IEEE, 2004, pp. 75–86.

[2] P. de Oliveira Castro, C. Akel, E. Petit, M. Popov, and J. William, "CERE: LLVM based codelet extractor and REplayer for piecewise benchmarking and optimization," *ACM Transactions on Architecture and Code Optimization (to appear)*, 2015.

[3] D. H. Bailey, *Nas parallel benchmarks*. Springer, 2011.

[4] "Omni Compiler Project Benchmarks." [Online]. Available: http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/

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

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

[7] Y. Kashnikov, P. de Oliveira Castro, E. Oseret, and W. Jalby, "Evaluating architecture and compiler design through static loop analysis," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*. IEEE, 2013, pp. 535–544.

[8] P. de Oliveira Castro, Y. Kashnikov, C. Akel, M. Popov, and W. Jalby, "Fine-grained benchmark subsetting for system selection," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*. ACM, 2014, p. 132.

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

[10] C. Akel, Y. Kashnikov, P. de Oliveira Castro, and W. Jalby, "Is Source-code Isolation Viable for Performance Characterization?" in *International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI)*. IEEE Computer Society, 2013.

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

[12] T. Lafage and A. Seznec, "Choosing representative slices of program execution for microarchitecture simulations: A preliminary application to the data stream," in *Workload characterization of emerging computer applications*. Springer, 2001, pp. 145–163.

[13] A. B. . A. Bataev, "Towards OpenMP Support in LLVM," in *2013 European LLVM Conference*, 2013.

[14] "LLVM OpenMP runtime," https://www.openmprtl.org/, accessed: 2014-09-01.

[15] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "Openuh: An optimizing, portable openmp compiler," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 18, pp. 2317–2332, 2007.

[16] R. E. Kessler, M. D. Hill, and D. A. Wood, "A comparison of trace-sampling techniques for multi-megabyte caches," *Computers, IEEE Transactions on*, vol. 43, no. 6, pp. 664–675, 1994.

[17] T. Sherwood, E. Perelman, and B. Calder, "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, pp. 3–14.

[18] X. Gao, M. Laurenzano, B. Simon, and A. Snavely, "Reducing overheads for acquiring dynamic memory traces," in *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*. IEEE, 2005, pp. 46–55.

[19] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009, vol. 344.

[20] "LLVM implicit barrier issue report," https://github.com/clang-omp/clang/issues/52, accessed: 2015-01-01.

[21] S. N. Natarajan, B. Swamy, A. Seznec *et al.*, "Modeling multi-threaded programs execution time in the many-core era," 2013.

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

[23] D. Quinlan and C. Liao, "The rose source-to-source compiler infrastructure," in *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, vol. 2011, 2011, p. 1.

[24] L. T. Yang, X. Ma, and F. Mueller, "Cross-platform performance prediction of parallel applications using partial execution," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*. IEEE, 2005, pp. 40–40.

[25] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5. ACM, 2002, pp. 45–57.

[26] L. Eeckhout, J. Sampson, and B. Calder, "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, pp. 2–12.

[27] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe, "Simflex: statistical sampling of computer system simulation," *IEEE Micro*, vol. 26, no. 4, pp. 18–31, 2006.

[28] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004, pp. 45–56.

[29] E. Perelman, M. Polito, J.-Y. Bouguet, J. Sampson, B. Calder, and C. Dulong, "Detecting phases in parallel applications on shared memory architectures," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006, pp. 10–pp.

[30] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sampled simulation of multi-threaded applications," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*. IEEE, 2013, pp. 2–12.

[31] E. K. Ardestani and J. Renau, "Esesc: A fast multicore simulator using time-based sampling," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*. IEEE, 2013, pp. 448–459.

[32] T. E. Carlson, W. Heirman, K. Van Craeynest, and L. Eeckhout, "Barrierpoint: Sampled simulation of multi-threaded applications," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

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

[34] A. J. Dorta, C. Rodriguez, and F. de Sande, "The openmp source code repository," in *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*. IEEE, 2005, pp. 244–250.

[35] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Computer Design: VLSI in Computers and Processors, 1996. ICCD'96. Proceedings., 1996 IEEE International Conference on*. IEEE, 1996, pp. 468–477.

[36] J. W. Haskins Jr and K. Skadron, "Memory reference reuse latency: Accelerated warmup for sampled microarchitecture simulation," in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*. IEEE, 2003, pp. 195–203.

[37] J. Zhai, W. Chen, and W. Zheng, "Phantom: predicting performance of parallel applications on large-scale parallel machines using a single node," in *ACM Sigplan Notices*, vol. 45, no. 5. ACM, 2010, pp. 305–314.