Piecewise Holistic Autotuning of Compiler and Runtime Parameters

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

University of Versailles – Exascale Computing Research

August 2016

Context

- Architecture, system, and application complexities increase
- System provides default good enough parameter configurations
  - Compiler optimizations: `-O2`, `-O3`
  - Thread affinity: `scatter`
- Outperforming default parameters leads to substantial benefits but is a costly process
  - Execution driven studies test different configurations
  - Applications have redundancies
  - Executing an application is time consuming
  - The search space is huge
  - Studies reduce the exploration cost by smartly navigating through the search space

Piecewise Exploration

- Codelet Extractor and REplayer (CERE) decomposes applications into small pieces called Codelets
- Each codelet maps a loop or a parallel region and is a standalone executable
- Extract codelets once
- Replay codelets instead of applications with different configurations to avoid redundancies

IS Motivating Example

```
int main()
{
create_seq()
for(i=0;i<11;i++)
  rank()
}
```

- IS benchmark
  - IS `create_seq` covers 40% of the execution time
  - IS `rank` sorting algorithm performs 11 invocations with the same execution time
- Piecewise exploration benefits
  - Avoid `create_seq` execution
  - Evaluate a single invocation of rank
  - IS `rank` and `create_seq` are not sensitive to the same optimizations

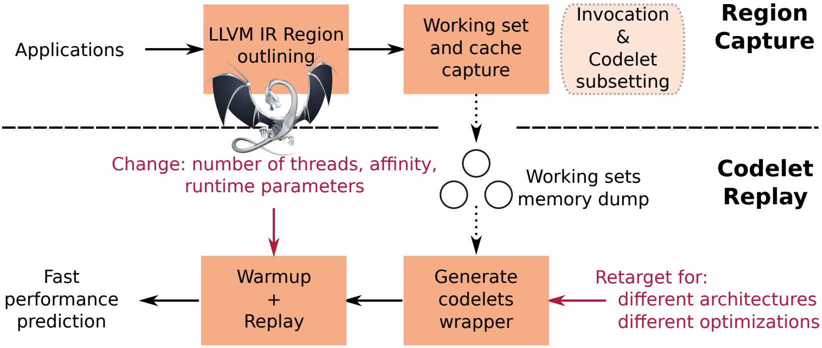
Outline

Codelet Extractor and Replayer (CERE)

Prediction Model

Thread and Compiler Tuning

CERE Workflow



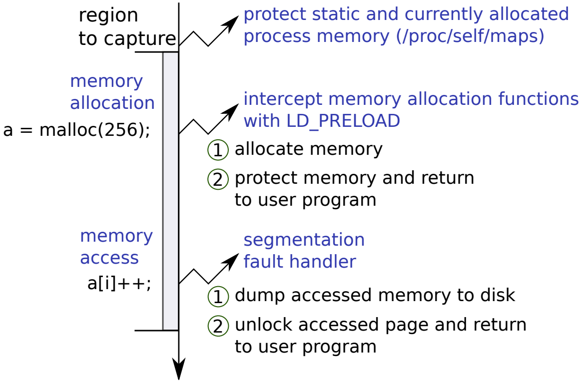
CERE can extract codelets from:

- Hot Loops
- OpenMP non-nested parallel regions

Codelet Capture and Replay

- Codelets are extracted at the LLVM Intermediate Representation level
- The user can recompile each codelet and replay it while changing compile options, runtime parameters, or the target system
- Performance accurate replay requires to capture the cache state
- Semantically accurate replay requires to capture the memory

Memory Page Capture

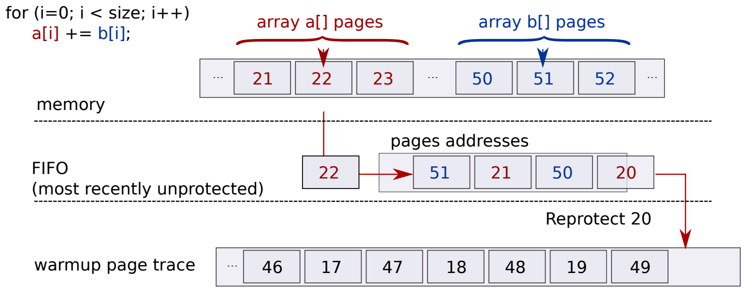


- Capture access at page granularity: coarse but fast
- Small dump footprint: only touched pages are saved

Cache State Capture

- Cold
  - Do not capture cache effects
- Working Set
  - Warms all the working set during replay (Optimistic)
- Page Trace
  - Before replay warms the last N pages accessed to restore a cache state close to the original

CERE Cache Warmup



OpenMP Regions Support

```
void main()
{
 #pragma omp parallel
 {
  int p = omp_get_thread_num();
  printf("%d",p);
 }
}
```

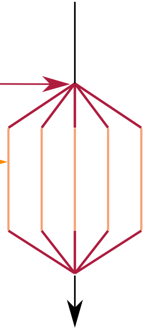
Clang OpenMP
front end

C code

```
define i32 @main() {
entry:
...
call @__kmpc_fork_call @.omp_microtask.(...)
...
}

define internal void @.omp_microtask.(...) {
entry:
   %p = alloca i32, align 4
   %call = call i32 @omp_get_thread_num()
   store i32 %call, i32* %p, align 4
   %1 = load i32* %p, align 4
   call @printf(%1)
}
```

LLVM simplified IR



Thread execution model

Selecting Representative Invocations

- A region can have thousand of invocations
- Performance differs due to different working sets
- Cluster to select representative invocations

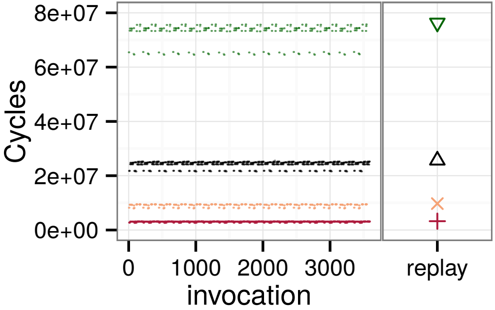
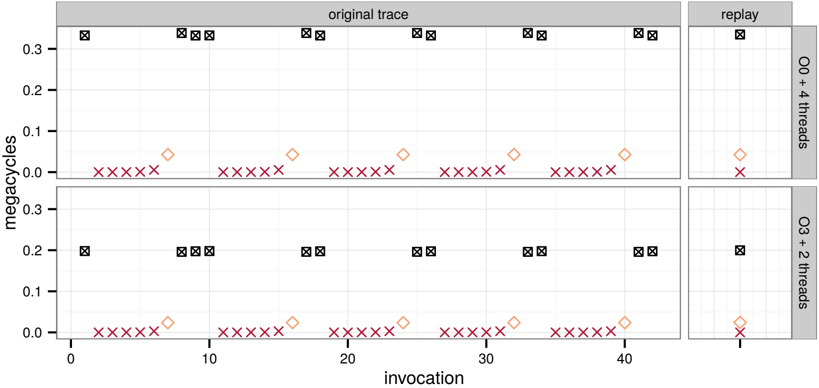


Figure: SPEC tonto `make_ft@shell2.F90:1133` execution trace. 90% of NAS codelets can be reduced to four or less representatives.

Performance Classes Across Parameters



- "MG resid" invocations execution time
- Use three invocations to predict the application execution time
- Parameters do not change the performance classes

NUMA Aware Warmup

- First touch policy: threads allocate the pages that they are the first to touch on their NUMA domain
- Detect the first thread that touches the memory pages
- During warmup the recorded NUMA-domains are restored

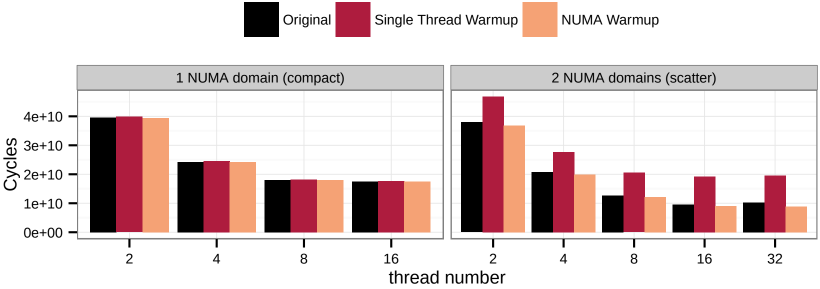


Figure: "BT xsolve" replay

Test Architectures and Applications

- NAS SER and NPB OpenMP 3.0 C version CLASS A
- Blackscholes from the PARSEC benchmarks
- Reverse Time Migration (RTM) proto-application
- Compiler LLVM 3.4

	Sandy Bridge	Ivy Bridge
CPU	E5	i7-3770
Frequency (GHz)	2.7	3.4
Sockets	2	1
Cores per socket	8	4
Threads per core	2	2
L1 cache (KB)	32	32
L2 cache (KB)	256	256
L3 cache (MB)	20	8
Ram (GB)	64	16

Figure: Test architectures

Blackscholes Thread Affinities Exploration

- Different thread affinities to evaluate
  - sn: n scatter threads
  - cn: n compact threads without hyper threading
  - hn: n compact threads with hyper threading

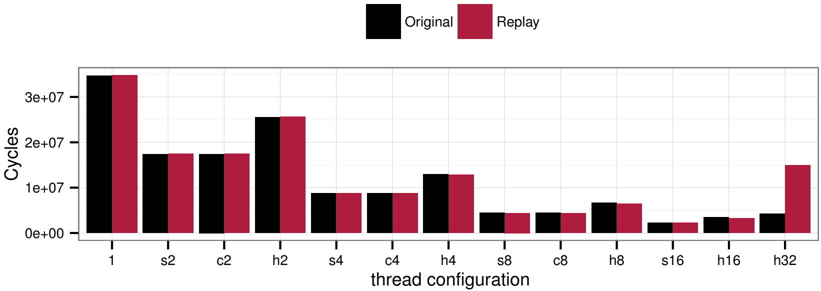


Figure: PARSEC Blackscholes thread configurations search

Outperforming Default Thread Configuration

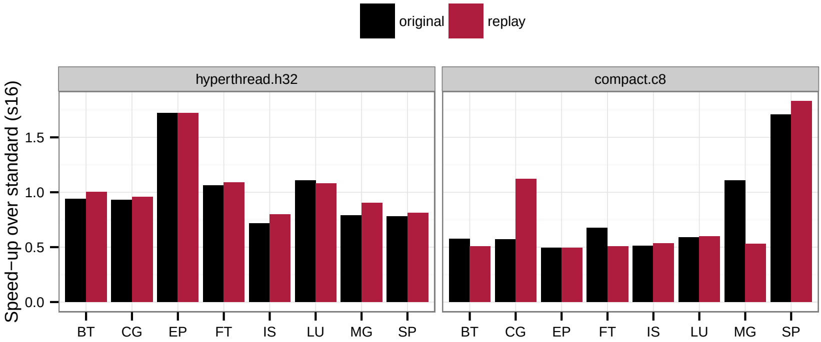


Figure: NAS thread configurations tuning

Autotuning LLVM Middle End Optimizations

- LLVM middle end offers more than 50 optimization passes
- Codelet replay enable per-region fast optimization tuning

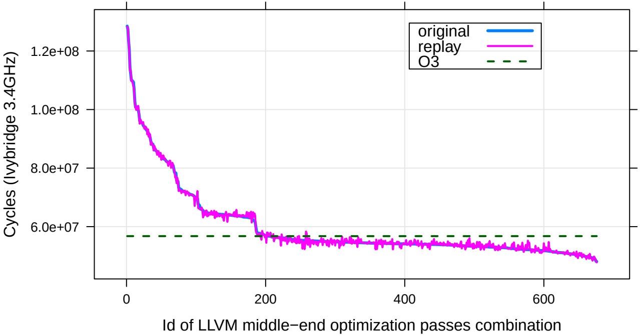
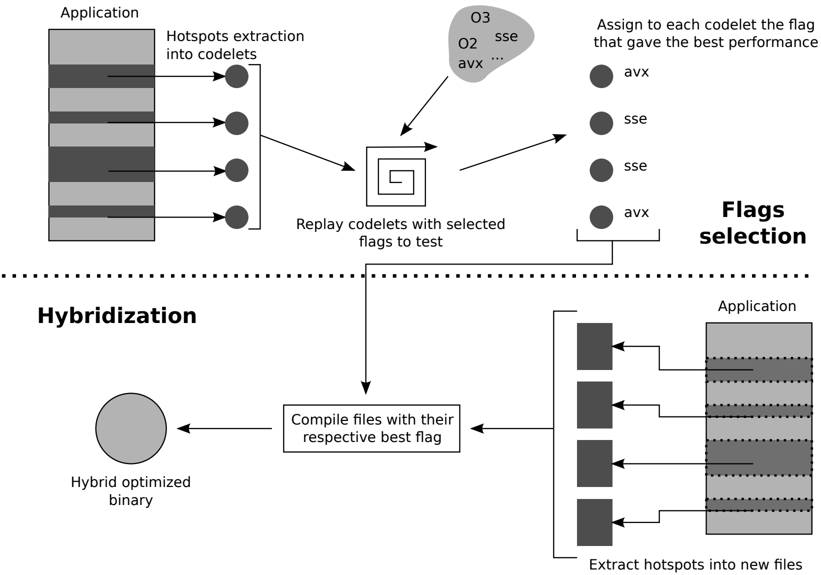


Figure: "SP ysolve" codelet. 1000 schedules of random passes combinations explored based on O3 passes.

CERE 149× cheaper than running the full benchmark
(27× cheaper when tuning codelets covering 75% of SP)

Hybridization



Hybrid Compilation over the NAS

- Four parallel regions of SP cover 93% of the execution time
- No single sequence is the best for all the regions
- Codelets explore parameters for each region separately
- Produce an hybrid where each region is compiled using its best sequence

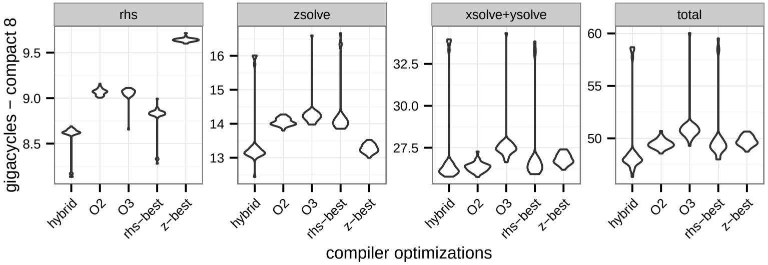
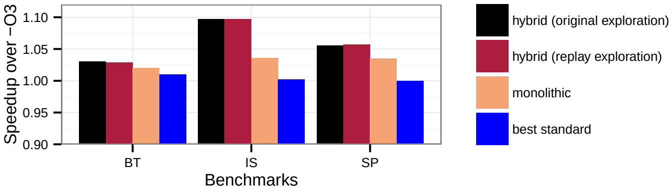


Figure: Hybrid compilation speeds up SP OpenMP 1.06×

Piecewise Exploration Benefits



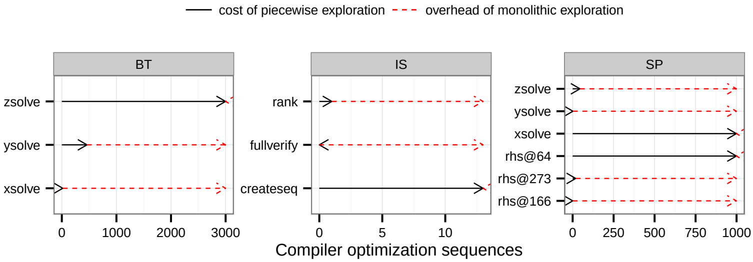


Figure: Piecewise exploration of the NAS SER

Codelets Tuning Results

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

- NAS SER and OpenMP benchmarks average speedup of $1.08\times$
- Tuning a single codelet is $13\times$ faster than full applications
- Codelet average accuracy is 94.6%
- RTM tuning through a codelet is $200\times$ faster and achieves a speedup of $1.11\times$

Related Works

- Kulkarni et al. "Improving Both the Performance Benefits and Speed of Optimization Phase Sequence Searches" (ACM Sigplan Notices 2010)
- Fursin et al. "Quick and practical run-time evaluation of multiple program optimizations" (HiPEAC 2007)
- Fursin et al. "Milepost gcc: Machine learning enabled self-tuning compiler" (Int. J. Parallel Prog. 2011)
- Purini et al. "Finding good optimization sequences covering program space" (TACO 2013)

Conclusion

- Piecewise tuning with codelets
  - Accelerate the exploration process
  - Improve the benefits
- Discussion
  - Some regions are not independent: LU `jacu` and `jacld`
  - Piecewise tuning sensitivity to the data set
- Future Work
  - Combine codelets tuning with GA
  - Use a clustering approach over codelets
  - Improve the parallel warmup strategy
- `https://benchmark-subsetting.github.io/cere/`