Continuous Shape Shifting: Enabling Loop Co-optimization via Near-Free Dynamic Code Rewriting

Animesh Jain, Michael A. Laurenzano, Lingjia Tang and Jason Mars
University of Michigan, Ann Arbor
{anijain, mlaurenz, lingjia, profmars}@umich.edu

***Abstract*—The class of optimizations characterized by manipulating a loop's interaction space for improved cache locality and reuse (i.e, *cache tiling / blocking / strip mine and interchange*) are static optimizations requiring a priori information about the microarchitectural and runtime environment of an application binary. However, particularly in datacenter environments, deployed applications face numerous dynamic environments over their lifetimes. As a result, this class of optimizations can result in sub-optimal performance due to the inability to flexibly adapt iteration spaces as cache conditions change at runtime.**

This paper introduces continuous shape shifiting, a compilation approach that removes the risks of cache tiling optimizations by dynamically rewriting (and reshaping) deployed, running application code. To realize continuous shape shifting, we present ShapeShifter, a framework for continuous monitoring of co-running applications and their runtime environments to reshape loop iteration spaces and pinpoint near-optimal loop tile configurations. Upon identifying a need for reshaping, a new tiling approach is quickly constructed for the application, new code is dynamically generated and is then seamlessly stitched into the running application with near-zero overhead. Our evaluation on a wide spectrum of runtime scenarios demonstrates that ShapeShifter achieves an average of 10-40% performance improvement (up to 2.4×) on real systems depending on the runtime environment compared to an oracle static loop tiling baseline.

I. Introduction

The class of loop optimizations that reshape the iteration space for cache locality and reuse are traditionally static compiler optimizations [1, 2, 3, 4]. With a specification of microarchitectural design and cache topology in a processor, the computation in an application's nested loops is restructured with strip mine and interchange passes into *tiles* – small subsets of the working set that fit into the cache – to take advantage of data reuse and improve the effectiveness of the cache for the computation. As a statically parameterized optimization, tiling requires that the compiler control both the size and shape of the tiles used in the computation. The choice of these parameters is intimately linked to the characteristics of architectural resources available to the application as it runs [5, 6, 7]. However, this class of optimization was conceptualized before the multicore era, which has introduced numerous additional dynamic factors that affect application runtime environments.

The advent of highly dynamic multicore/multiprocessor environments necessitates the rethinking of how cache tiling should be applied and deployed in commercial and production

978-1-5090-3508-3/16/$31.00

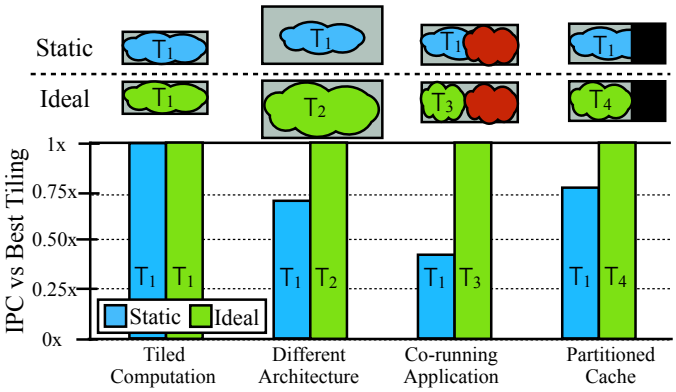

Fig. 1. The optimal tiling for one runtime environment can perform poorly in other environments

contexts. The static assumptions used to aggressively tune the tiling parameters can be easily broken by sources of post-deployment dynamism. This concept is illustrated in Figure 1, which compares two tiling approaches. First, an approach that aggressively tiles for one runtime environment achieves excellent performance in that environment, but may perform poorly in other environments. Second an ideal approach that aggressively tiles for each runtime environment. The figure shows that pre-deployment best tile can result in sub-optimal performance across different runtime environments.

Although there has been some prior work addressing particular challenges that arise from dynamism [8, 9, 10], these works use white-box approaches to target particular sources of inefficiency. Realizing a holistic approach that continuously adapts to numerous, varied sources of post-deployment dynamism requires a black box approach and remains an open problem. In particular, three sources of dynamism must be addressed to realize a loop iteration space specialization that is deployable in modern commercial and production environments. These sources of dynamism include:

1) **Co-runner Dynamism.** Cloud and datacenter operators routinely co-run applications to improve server utilization [11, 12, 13] and multi-program workloads have become a norm on desktop and mobile platforms [14]. The co-runners an application faces will vary in number and character.
2) **Microarchitectural Flexibility.** Processor design has evolved significantly since the original conceptualization

of cache tiling. Now, microarchitectural parameters may change over the course of an application run. For instance, cache way-gating [15], processor power capping [16] and cache partitioning [17, 18] may be used to slow, constrain, or shut down architectural resources in order to limit power consumption or provide performance isolation.

3) **Microarchitectural Diversity.** Datacenters operators typically house numerous architectural implementations [19, 20, 21], and heterogeneous architectures, for example ARM big.LITTLE, are becoming common because of their energy efficiency. Moreover, the target platform for commercial off the shelf (COTS) software is rarely known ahead of deployment, and each platform may have different cache configurations and microarchitectural implementations.

Each of these sources of dynamism impacts the availability of important architectural resources to the application, significantly affecting how cache tiling should be aggressively employed. The set of factors impacting the choice of cache tiling parameters is broad, have complex interactions, and may change many times over the course of a single application run. Handling this myriad factors therefore demands a novel, dynamic solution that can quickly and seamlessly change the tile structure to reflect changes to an application's runtime environment.

To design a cache tiling solution that can encompass these numerous factors, two main challenges emerge:

(i) *the solution should be **accurate**, generating tiles that are customized to take full advantage of cache and delivering significant performance benefits, and*

(ii) *monitoring application code for tiling opportunities and rewriting application code to introduce new tiles must be **low-overhead**, such that the overhead of those activities does not outweigh the benefits of the improved tiles.*

A key insight of this work is to use a rapidly and dynamically constructed environment- and application-specific *black box* model for predicting the performance of a host of tiling options within the immediate environment. This paper introduces **continuous shape shifiting** with *ShapeShifter*, an end-to-end dynamic compilation infrastructure that enables continuous shape shifting and aggressively rewrites running applications in response to runtime dynamism. ShapeShifter uses a lightweight monitoring infrastructure to examine the running applications and the runtime environment to look for opportunities to tile and re-tile the applications in response to changes in the runtime environment. Upon identifying a suitable tile shape based on the dynamically constructed model, ShapeShifter rewrites and re-tiles the application leveraging a low-overhead dynamic compilation capability to divert execution into the aggressively tiled code with near-zero overhead.

In addition to continuous shape shifting, we propose a *co-optimization* algorithm to perform retiling of multiple co-runners simultaneously. It is a challenging problem to find suitable tile shapes for multiple co-runners because optimizing a tile shape for a co-runner can change the optimal tile for an already optimized co-runner. We observed that cache interference often has little to do with tile shape, i.e., different tile shapes of the same tile size produce similar amount of interference to other co-runners. This observation can be leveraged to design an approach that quickly finds suitable tile shapes. This is the first work to consider the effect of this dynamic interference in the presence of multiple co-runners. The specific contributions of this paper are:

- **Study of Impact of Dynamism on Tile shapes –** we study the impact of several important sources of post-deployment dynamism on the efficacy of cache tiling techniques. It shows that static tiling approaches lose significant performance opportunity unexploited amidst various sources of dynamism.
- **Continuous Runtime Tiling with ShapeShifter –** we introduce ShapeShifter, a system that runs continuously, detecting and reshaping tiles within multiple running applications. ShapeShifter has negligible overhead, designed to be suitable for continuous deployment in production and commercial environments.
- **Black-Box Dynamic Tile Generation –** at the core of ShapeShifter is a tile generation algorithm capable of seamlessly handling numerous sources of dynamism. This algorithm centers around a novel black box approach to tile selection based on rapid online model creation for the specific application and runtime environment into which the application will be deployed.
- **Real System Evaluation –** we evaluate ShapeShifter on real systems within a spectrum of runtime environments spanning several architectural platforms, showing that by aggressively retiling application tiles, we are able to achieve an average of 10-40% performance improvement (up to 2.4×) over an oracle that aggressively tiles for a single runtime environment.

II. Motivation and Background

In this section, we investigate the opportunity available in the presence of a solution that can aggressively re-tile application code in the context of three common sources of post-deployment dynamism.

A. Opportunity Analysis

The efficacy of a cache tile depends heavily on the runtime environment, as there are numerous factors in the runtime environment that can impact the availability of cache and other microarchitectural resources. This study focuses on three such sources of dynamism – the impact of co-running with other applications, the impact of changing the amount of cache available to the application, and the impact of microarchitectural diversity. Our baseline is an approach we call *StaticBest* that exhaustively runs a large space of tiling parameters to determine the best tiling configuration for a runtime environment that (1) has no-co-running applications, (2) is for a commodity server processor (AMD Opteron), and (3) for which the application has full use of the 16-way L2 cache. We evaluate on a host of applications from Polybench [22].

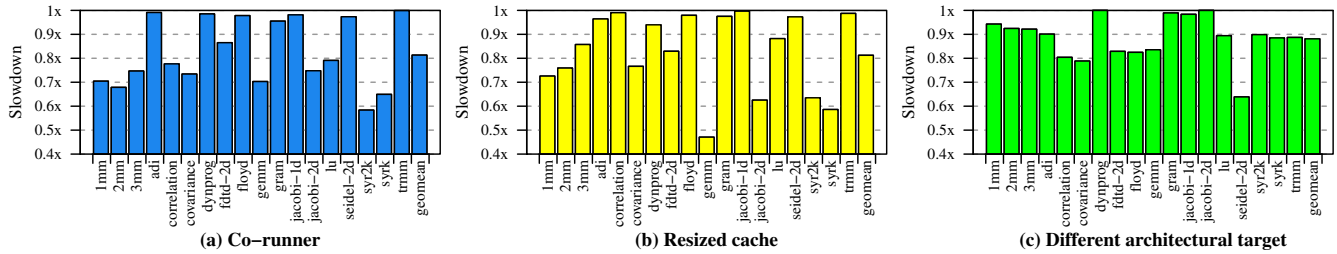

Fig. 2. Suboptimal performance if the application code is not retiled to the application runtime environment

Co-runner Dynamism. We first evaluate the efficacy of StaticBest when the assumption that the application has no co-runners proves to be untrue (Figure 2(a)). For this comparison, we run the applications again against a cache pressure microbenchmark while employing each of the tiling parameterizations used to find StaticBest, then selecting the best performing tiling, which we term AggressiveBest. The difference in performance between StaticBest to AggressiveBest can be interpreted as the slowdown resulting from a failure to tailor the tiling approach to its runtime environment. Figure 2(a) illustrates the resulting slowdown, which is over 19% on average, and up to 41% for syr2k.

Microarchitectural Flexibility. We next evaluate the efficacy of StaticBest when the assumption that the full L2 cache is available to the application is violated. We use Bulldozer's way-locking feature to lock half the ways of the 16-way L2 cache, effectively reducing the cache size by half. The resulting slowdown if the applications are not re-tiled to respond to this microarchitectural change is illustrated in Figure 2(b). In this case, up to a 52% slowdown over the optimized tile is observed for gemm, with an average of 19% across all applications.

Microarchitectural Diversity. Finally, we evaluate the efficacy of the optimized tile if the assumption that the target architecture is an AMD Bulldozer is violated. To do this, we find AggressiveBest when running the applications on an Intel Haswell server and compare the resulting performance to StaticBest on the Haswell server. Like the previous cases, a significant performance opportunity is left unexploited if applications are not re-tiled to reflect this different runtime environment. The maximum resulting slowdown is 37% for seidel-2d and averages 12% across applications.

B. Limitations of Prior Work

We compare ShapeShifter to the most relevant previous work in Table 1 [8, 9, 10]. Both Defensive Tiling [8] and Dynamic Selection of Tile Sizes [10] do not retile multiple co-runners simultaneously. It is a necessary and challenging problem to solve as optimizing the tile for one application can change the best tile for an already optimized co-runner. ShapeShifter has the capability of retiling multiple co-running applications. We provide insights as to how tile shape and size affects the interference between applications. We then present an algorithm built upon that insight to find suitable tiles for co-runners simultaneously. Reactive tiling [9] strives to find the best tiling parameters in the presence of cache partitioning. However, it is evaluated on simulators, which have limited ability to capture industry-standard proprietary features such as prefetcher designs and cache replacement policies. In addition, ShapeShifter supports dynamic compilation providing the opportunity to use wide range of compiler optimizations suitable to the runtime environment.

TABLE I. Comparison between ShapeShifter and prior retiling works

	Defensive Tiling[8]	Reactive Tiling[9]	Dynamic Selection[10]	Shape-Shifter
Retiling multiple co-runners				✓
Rectangular tiles		✓	✓	✓
Black-box model approach		✓	✓	✓
Compilation flexibility				✓
Real system evaluation	✓		✓	✓
Handles co-runner presence	✓			✓
Handles cache-partioning		✓		✓
Handles platform changes				✓

III. System Overview

This section describes the design of ShapeShifter, a dynamic compilation infrastructure that takes advantage of the opportunity to aggressively re-tile running application code to reflect the runtime environment. We discuss the main challenges in designing such an infrastructure, and give an overview of how ShapeShifter overcomes these challenges.

A. Challenges

Accuracy. Realizing a tiling approach that is *universal*, capable of identifying the right tile among a broad range of runtime environments, is a challenging problem. Existing solutions using detailed cache and memory access pattern models are designed to focus on a narrow range of the possible runtime environments. Thus, designing a new mechanism capable of reasoning about cache tiling and correctly identifying the most suitable tiling parameters among many runtime environments is necessary for solving this problem.

Overhead. A dynamic re-tiling system must be left in place continuously throughout application execution, available to monitor the application and environment, and able to take steps to exploit re-tiling opportunities as they arise. Having such a capability that is low overhead is a challenging problem. Classic virtualization-based monitoring and dynamic compilation

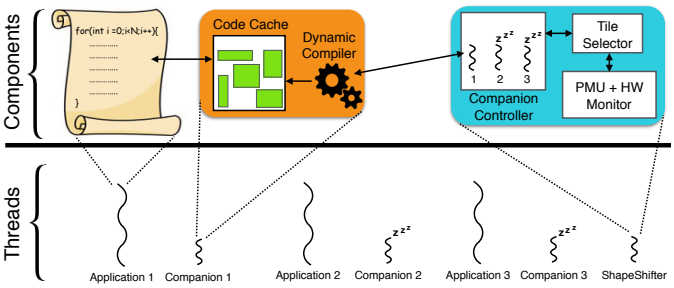

Fig. 3. Dynamic compilation and monitoring infrastructure

infrastructures are ill-suited to this task, as the overhead introduced by those infrastructures can easily outweigh benefits of the optimizations themselves.

B. ShapeShifter System Architecture

ShapeShifter is an end-to-end dynamic system continuously monitoring the running application and runtime environment and looking for opportunities to re-tile the application code. The runtime environment can change because of arrival/departure of co-runners, architectural policy changes and platform changes.

To achieve this dynamic capability, ShapeShifter spawns a runtime thread for each application as soon as it starts execution, as shown in Figure 3. This thread, referred to as *Companion thread*, provides a dynamic compilation capability to its application. ShapeShifter continuously looks for tiling opportunities by using a *Runtime Environment monitor*. When an opportunity is identified, it triggers a *Tile Generator* module that accurately predicts a suitable tile for the current runtime environment. Finally, ShapeShifter instructs Companion thread to introduce the new tiling strategy into the application code.

Companion threads, Tile Generator and Runtime Environment Monitor work in tandem to achieve continuous shape shifting. Here we provide an overview of these components.

Companion Thread. Companion threads provide Dynamic Compilation infrastructure inspired by protean code to introduce re-tiled code into the running application [23]. The key difference between protean code and other traditional heavy-weight dynamic compilation infrastructures is that protean code runs asynchronously to the application, without stalling the application progress. The application continues running the old code variant and switches to the new code variant only when protean code has lazily stitched it into the running application. Therefore, protean code incurs low overhead on the application performance.

Companion threads are woken up only when a tiling opportunity is detected, as illustrated in Figure 3. Because of its minimal interaction with the application, it provides a low-overhead dynamic compilation solution to achieve continuous shape shifting.

Runtime Environment Monitor. One of the key capabilities of ShapeShifter is to detect opportunities to re-tile running application code. This capability takes the form of a *Runtime Environment Monitor* (REM), a lightweight process that occasionally polls the machine state via hardware event monitors and model specific registers (MSRs). It collects performance and cache statistics counters that are used to guide tiling decisions. MSRs often expose useful information about microarchitectural state. This information helps in constructing a view of the application runtime environment. For example, the AMD Bulldozer platform support way-locking in the L2 cache, and MSRs expose the number of L2 cache ways available at any given time. By monitoring the relevant machine state, ShapeShifter can detect changes in the architectural policies at any time.

Tile Generator. Tile Generator is responsible for predicting a suitable tile for the application current runtime environment. It uses the performance and cache statistics collected by REM to generate an online black box linear model. Using this model, ShapeShifter predicts a tile that is optimized for the current runtime conditions. It instructs the Companion thread to generate the corresponding tiled variant and stitch it into the application code.

IV. ShapeShifter Design and Implementation

In this section, we provide description of ShapeShifter runtime system. The different components of ShapeShifter – Companion threads, Tile Generator and Runtime Environment Monitor – work hand-in-hand to identify and take advantage of tiling opportunities. Figure 4 gives an overview of the ShapeShifter runtime system. Whenever REM detects a tiling opportunity, Tile Generator starts constructing an application- and environment-specific tiling performance model. It instructs the Companion thread to stitch a handful of different tile parameterization codes into the application, where each is run for short time. REM collects the performance and cache statistics, referred to as Training data, while these tiles execute. This training data is used by Tile Generator to construct a tiling performance model on the fly. Tile Generator then selects the tiling with the highest modeled IPC and invokes Companion thread to introduce that tiling into the application. We show in Section VI that this tile generation process is highly accurate, choosing tiling strategies close to optimal-tiling performance.

We divide the above process into three parts: Online training (§IV-A), Tile Generation (§IV-B), and Monitored execution (§IV-C). We now describe these three steps in detail.

A. Online Training

Online training is triggered when the REM detects a change in the application runtime environment. In this step, the REM collects training data with the help of the Tile Generator and Companion threads. This training data is then later used to develop a tiling performance model. The process can be further broken down into 2 steps. First, finding a suitable tile size for the application and second, collecting training data.

1) Tile Size Selection: Both tile size and shape are important tile characteristics that impact the performance of a tiled loop nest. Tile size defines the working set of the application. A working set larger than the targeted cache size slows down

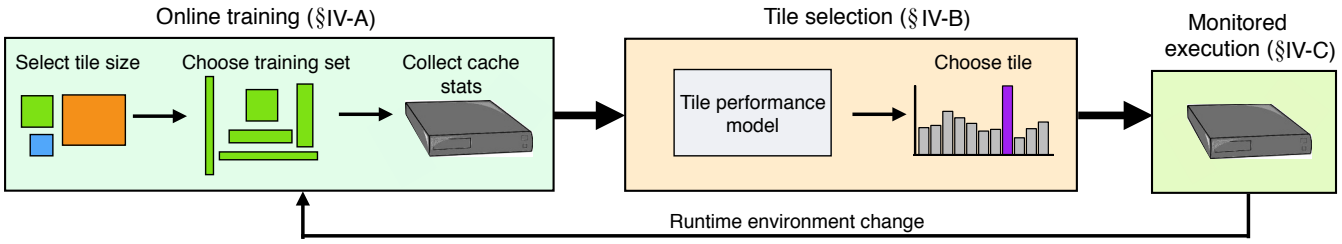

Fig. 4. Tile selection in ShapeShifter is accomplished by running a small training set of tiling parameters, which is used to model the IPC of a large space of tiling then to select the tiling with the highest IPC

the application because some memory requests take longer to finish as they have to go to lower and slower levels of memory hierarchy. On the other hand, a working set much smaller than the cache size does not utilize the data reuse efficiently. This step tackles the problem of finding a suitable tile size for the application, whereas the problem of finding a suitable tile shape is solved by the black box model described in Section IV-B2.

On detecting a change in the application runtime environment, the REM reads in current cache size using software visible registers and MSRs. This information is passed on to Tile Generator that instructs the Companion thread to generate a tile variant consuming a certain portion of the available private cache size. Companion thread executes this tile variant while REM collects the performance and cache statistics during the tile execution. This process is repeated with reduced tile size until the private cache miss rate is below a certain threshold (<2% in our case). This low cache miss rate signifies that the working set of the application now fits in the cache. This produces a tile variant whose tile size is tuned for the application current runtime environment.

2) Collection of Training Data: On finding a suitable tile size, ShapeShifter starts collecting training data to help generate a tiling performance model. In this step, Tile Generator generates a set of training tiles, Companion threads executes these training tiles one by one for short duration while REM collects the performance and cache statistics for each tile. Algorithm 1 provides an overview of this whole process.

Algorithm 1 Online training

```
1:  Input: TileSize
2:  Output: TrainingData
3:  function GetTrainingData(TileSize)
4:      GenTrainingSet(TileSize)                    ▷ Tile Generator
5:      for (i in 1:size(TrainingSet)) do
6:          GenTiledVariant()                       ▷ Dynamic Compiler
7:          DispatchTileToApp()                     ▷ Dynamic Compiler
8:          RunTheTile()
9:          data = CollectPerfMonData()             ▷ REM
10:         TrainingData = TrainingData + data
11:     end for
12:     return TrainingData
13: end function
```

Tile Generator first generates a set of training tiles using the tile size identified in the previous step but with varying tile shapes. There is a broad range of tile shapes to choose from. We classify the tile shapes in 3 categories: Broad tiles – tiles with large number of rows but few columns, Narrow tiles – tiles with few rows but large number of columns, and Intermediate tiles – tiles that are neither broad not narrow. As shown in Figure 4, Tile Generator chooses only a subset of these tiles to profile the application. In total, the training set consists of 5 versions of application - 2 broad-tiled, 2 narrow-tiled, 1 intermediate-tiled.

Tile Generator instructs the Companion thread to introduce training set into the application code. These tiles are then executed one by one while REM collects performance counters during their execution. Specifically, REM collects this information for each tile in the training set: a) number of retired instructions, and b) number of execution cycles . This creates a training database which is later used to develop a tiling performance model.

B. Tile Generator

The training data is now used to develop a model and identify a suitable tiling strategy for the application runtime environment. The runtime environment can change because of various factors like arrival of co-runners, microarchitectural policy changes and platform changes. Figure 5 gives an overview of tile generation. Tile Generator uses the training data collected by REM and creates an online black box model for the current application and runtime environment. The black-box model does not assume any prior knowledge of the application and architecture, and is completely created on the fly using the training data. Since the model has to capture only the current application and runtime environment, a relatively simple model can suffice to achieve high prediction accuracy. This is in contrast to traditional white-box models that are quite complex because they are designed to handle a wide variety of cases.

1) Black-box Development: ShapeShifter uses the online training data to develop a black box model. Our goal is to generate a model that takes a tile T_i as an input and predicts its corresponding performance IPC_i. Tile Generator uses this model to identify a suitable tilting strategy for the application in its runtime environment.

We first define a tile T_i. It consists of three parameters – t_i^1, t_i^2, t_i^3 as we focus on widely used three-dimensional tiling [8, 9, 10, 24]. Thus, a tile T_i can be represented as

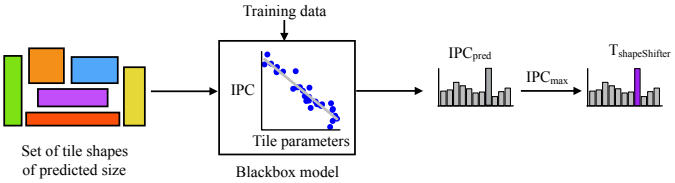

Fig. 5. ShapeShifter tile shape selection - Tile Generator applies the blackbox model on a set of tiles and chooses the tile with the highest predicted IPC

$$T_i =< t_i^1, t_i^2, t_i^3 > \tag{1}$$

The online training data has five training tile parameters and their observed IPC. Tile Generator develops a linear model between these training tile parameters and their corresponding IPC. It applies a linear curve fitting method on these five data points. This model can be formalized in the following manner.

$$f(T_i) \Longrightarrow IPC_i \tag{2}$$

where IPC_i is the modeled IPC for the application.

Note that the model is obtained by applying a linear curve fitting method on just five data points. The overhead of generating a linear model with so few points is minimal. Also note that this model only captures current application runtime environment. Therefore this model needs to be updated if a new runtime environment is encountered.

2) Tile Shape Selection: This step uses the black box model represented by Equation 2 and predicts a suitable tile shape for the application in its current runtime environment. As shown in Figure 5, Tile Generator applies the black box model on a large span of tile shapes consisting a mix of broad, narrow and intermediate tiles. Note that ShapeShifter is not executing these tiles, it is just applying the black box model to predict the IPC of each of the available tile shapes. The tile with the maximum predicted IPC is chosen as the tile for further execution. This tile is referred to as $T_{shapeShifter}$.

Algorithm 2 Tile shape selection

```
1: Input: TrainingData
2: Input: AvailSet
3: Output: ShapeShifterTile
4: function GetShapeShifterTile(TrainingData)
5:     bbModel = GenBBModel(TrainingData)
6:     predIPC = Apply(bbModel, AvailSet)
7:     ShapeShifterTile = maxIPC(predIPC)
8:     return ShapeShifterTile
9: end function
```

Algorithm 2 gives an overview of this step. This step can be represented in the following manner

$$T_{shapeShifter} : IPC_{shapeShifter} = \max_{i \in avail\_tiles} f(T_i) \tag{3}$$

Tile Generator invokes Companion thread to create a new version of application with $T_{shapeShifter}$ parameters. This version is used for execution from now on.

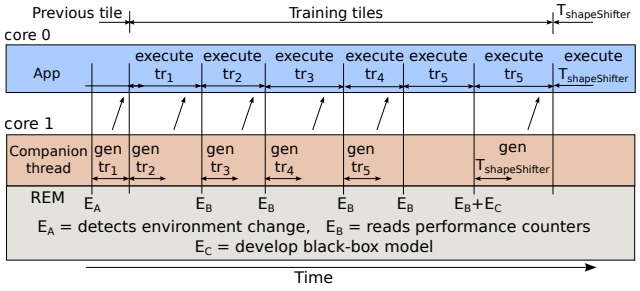

Fig. 6. REM detects the environment change and wakes up companion threads to start training phase leading to creation of ShapeShifter tile

A key point to notice is that the black box model does not have to predict the IPC of each tile accurately. Even ranking the tiles accurately is more than necessary for our purpose. Minimally, ShapeShifter should be able to pick up an acceptable tile when there is a large variation in the IPC of available tile shapes. We show in Section VI that ShapeShifter black box model is highly accurate. It asserts that a simple linear model generated online is sufficient to identify suitable tile parameters across a wide range of runtime environments.

C. Monitored Execution Phase

REM continuously monitors the runtime environment and triggers online training in the presence of a tiling opportunity. For detecting changes in architectural policies and platform, REM periodically polls MSRs and other software visible registers. In order to detect the presence of a co-runner, ShapeShifter uses a simple technique of monitoring cache misses. Arrival of a co-runner typically increases cache miss rate. ShapeShifter assumes the presence of a software/hardware mechanism that provides an estimate of cache size that should be allocated to each co-runner. In the absence of such a mechanism, ShapeShifter assumes that the co-runners consumes half of the available cache capacity.

In addition, ShapeShifter remembers the tile for a particular application and runtime environment. If the same runtime environment shows up later, ShapeShifter uses the stored tile for the application to avoid unnecessary training overhead.

The entire process is shown in Figure 6. In the figure, the application is executing on core 0 while companion threads, REM and Tile Generator are running on core 1. On detecting a change in the runtime environment (event E_A), the Tile Generator starts the training process and instructs Companion thread to generate the training tiles (tr_1-tr_5) and stitch them in the application code. The application runs these training tiles one-by-one while the REM keeps collecting cache and performance statistics for each training tile execution (event E_B). After online training is complete, Tile Generator uses the training data and predicts a suitable tile for the current runtime environment (event E_C). This tile is used for further execution.

In frequently changing environment, REM detects a change while the training is in process. In that case, ShapeShifter

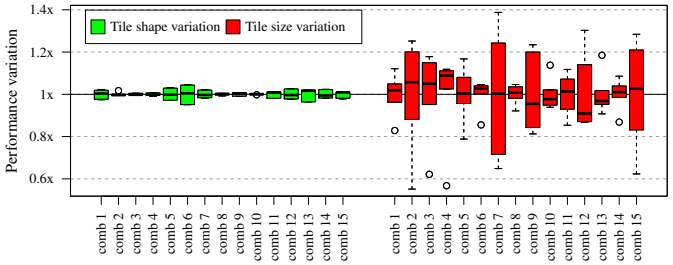

Fig. 7. Interference caused by different tile shapes is similar whereas different tile sizes exert significantly different amount of cache pressure

AMD Bulldozer	Intel Haswell	Intel Atom
Opteron 6272	Xeon E3-1240v3	Atom 330
16 cores	4 cores	2 cores
2.1 GHz	3.4 GHz	1.6 GHz
48K, 4-way, L1 (private) 2M, 16-way, L2 (shared) 12M, 128-way, L3 (shared) Individual way-locking on L2. Experiments use 16 -way/8-way and 4-way unlocked configurations	32K, 8-way, L1 (private) 256K, 8-way, L2 (private) 8M, 16-way, L3 (shared)	24K, 6-way, L1 (private) 512K, 16-way, L2 (shared)

TABLE II. Platforms used in the evaluation

finishes the training and discards REM detection for a certain duration.

V. Loop Co-optimization

Applications are often co-run in datacenter operators to improve server utilization [11, 12]. Also, executing multiple programs are common on desktop and mobile platforms [14]. A universal tiling strategy needs to find suitable tile shape and size for all the co-running applications such that the interference between them is minimized. ShapeShifter provides a capability of capturing this interference and adjust tiles of multiple co-runners to their corresponding effective cache size. We refer to this feature as *co-optimization*.

A major hurdle in achieving effective co-optimization is the search space. Tiling for one application itself has a huge search space. Adding co-runners makes the problem intractable. Applying different tiles in one application creates different runtime environments for the co-running applications, and thus, optimizing tiling for one application can change the best tiling strategy for an already optimized application. This makes tiling for multiple co-runners simultaneously a challenging problem.

An insight that can enable a solution to this problem is that the interference caused by co-runners is largely a function of their tile size, that is, different tile shapes of the same size exert similar amount of cache interference. This is illustrated in Figure 7. In this experiment, we take 15 pairs of co-runners and study performance variation of the first application when (left) only tile shape of the second application is varied while keeping the tile size same and, (right) tile size of the second application is varied. This shows that different tile shapes among a tile size result in similar amount of interference, while different tile sizes result in much larger performance variation. This insight gives us a strong foundation to solve the challenging problem of tiling for multiple co-runners.

Algorithm 3 Co-optimization

```
1:  Input: Apps
2:  function CO-TILING(Apps)
3:      Initialize TileSize[Apps]
4:      for (app in Apps) do                          ▷ Optimize size
5:          T_optSize = FindTileSize(app)
6:          DynComp(app, T_optSize)
7:          TileSize[app] = T_optSize
8:      end for
9:      for (app in Apps) do
10:         Ts = TileSize[app]
11:         trainingData = GetTrainingData(Ts)        ▷ Algo 1
12:         T_shapeShifter = GetShapeShifterTile(
    trainingData)                                     ▷ Algo 2
13:         DynComp(app, T_shapeShifter)
14:     end for
15: end function
```

Algorithm 3 gives an overview of our co-optimization. ShapeShifter first identifies a suitable tile size for all the co-runners as described in Section IV-A1. We refer to the tiles after this step as $T_{optsize}$, as the tiles have been optimized for size. Since the interference is dependent heavily on the tile size and does not change significantly with the tile shape, this step creates a stable runtime environment, whereafter the cache interference does not change significantly as the tile size changes. Therefore, ShapeShifter now optimizes the tile shape of all the co-runners one-by-one. Since the cache interference does not change with tile shapes, optimizing tile shape once for all the co-runners results in suitable tiling strategies. We observed that additional optimization on tile shapes resulted in marginal performance improvements. We evaluate co-optimization in Section VI-C.

VI. Evaluation

A. Methodology

Applications. We evaluate ShapeShifter on the Polybench application suite [22, 25, 26, 27, 28, 29], a collection of linear algebra, stencil computation and data mining algorithms.

Implementation. We used Polly [30], a polyhedral optimizer tool that is integrated into LLVM [31] to perform tiling. We integrated Polly with protean code [23] to implement ShapeShifter. Polly performs cache tiling on LLVM intermediate representation while protean code provides the dynamic compilation capability.

Hardware Platforms. Our evaluation encompasses three design points with different microarchitectural and architectural configurations, as summarized in Table II. These platforms are an AMD Bulldozer, an Intel Haswell and an Intel Atom. The AMD Bulldozer allows way-locking on its 16-way L2 cache, preventing a subset of ways in the cache from being

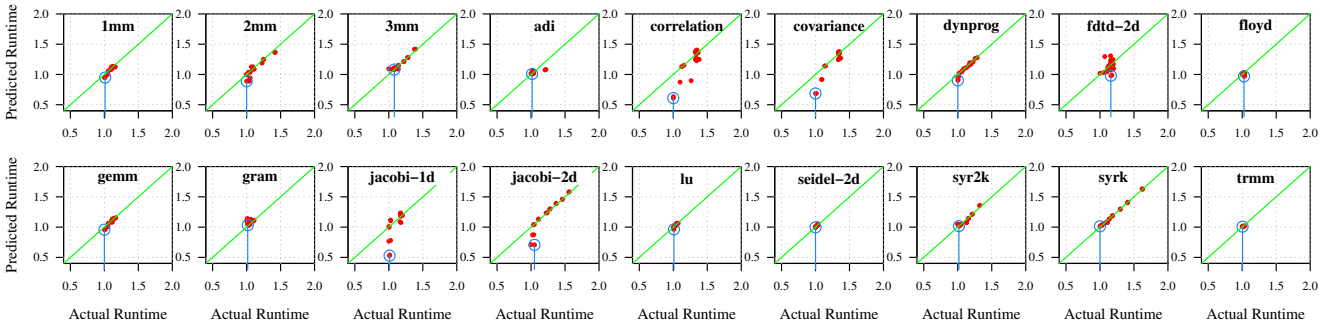

Fig. 8. Actual runtime of applications vs. the runtime modeled by ShapeShifter's dynamic Tile Generator

accessed by any application. We consider three configurations of way-locked L2 in our evaluation: completely unlocked (all 16 ways are active), half locked (8 ways are available) and mostly locked (4 ways are available).

Baselines. Our baseline is the best performing tile on the largest cache across all the machines. We find this tile by statically running an exhaustive search space on the largest cache in our experimental setup. We term this baseline as *Static Best* approach to tiling.

B. Tile Selection Accuracy

This section evaluates the black box modeling technique at the core of the tile selection algorithm. The goal of the model is to map tiling parameters to performance, thus allowing the Tile Generator to choose the tiling strategy with the best performance of the available tiles. For these experiments, we statically compile and perform a run of the application with a host of different tiling strategies, measuring the runtime of each.

The results of this experiment are presented in Figure 8, where each plot shows the modeled vs. actual runtime for a particular benchmark, normalized to the runtime of the fastest tiling strategy. Inside each plot, the position of a particular point on the x-axis gives the actual runtime for a single tiling, while its position on the y-axis gives the modeled runtime from the black box model. As a guide, each plot has a line at x=y to show where perfect predictions (modeled runtime equals actual runtime) would reside. Also in each figure is a circled point, showing the tiling strategy chosen by ShapeShifter's tiling selection algorithm, along with a line that illustrates the actual runtime of that point.

Some applications, such as dynprog, result in precise models where the actual and modeled runtimes track each other closely across tiling parameters. However, a precise model is far beyond what is necessary to select a high performance tiling. To make this more clear, we highlight covariance, where the modeled runtime of the tiling chosen by ShapeShifter is 70% of the actual runtime but the the tiling strategy is still the fastest from among the available options. Similarly in jacobi-2d, there are numerous tiling strategies offering similar high performance and ShapeShifter chooses one from among them.

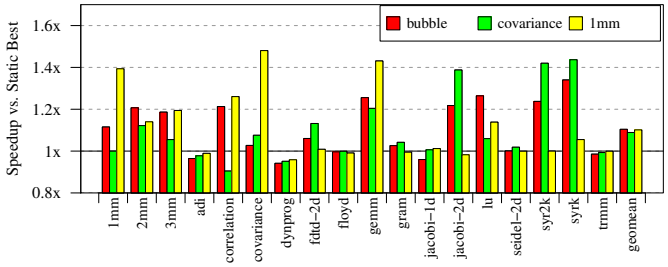

Fig. 9. ShapeShifter adjusts the tiling strategy of an application in presence of diverse co-runners

This stresses the idea that our models do not need to predict absolute performance precisely.

C. Dynamism in Co-runners

In the era of multicore processors, the common case is that multiple applications are run together on a system at the same time. These co-running applications compete for shared resources, which includes caches. In this section, we evaluate how ShapeShifter adapts to runtime environment in the presence of co-runners. These experiments are run on the AMD Bulldozer platform, and we measure co-runners' performance as they run with ShapeShifter. We measure IPC for all the co-running applications and use it compute weighted speedup.

Stable Co-running Workloads. In this set, we conduct three experiments where ShapeShifter is used among 1, 2 or 4 applications.

In the first experiment, we evaluate how co-optimization performs if we limit it to optimize only one co-runner while the other co-runner tile remains unchanged. The results of this experiment are presented in Figure 9, which shows the performance improvement ShapeShifter-tiled application normalized to Static Best in the presence of three different co-runners: (1) the bubble, a microbenchmark designed to place pressure on a specific subset of the cache, which we configure to place pressure on half the L2 cache (1MB), (2) covariance from polybench, and (3) 1mm from polybench. These results demonstrate that by re-tiling application code, ShapeShifter is able to achieve sizable speedups over Static Best, achieving

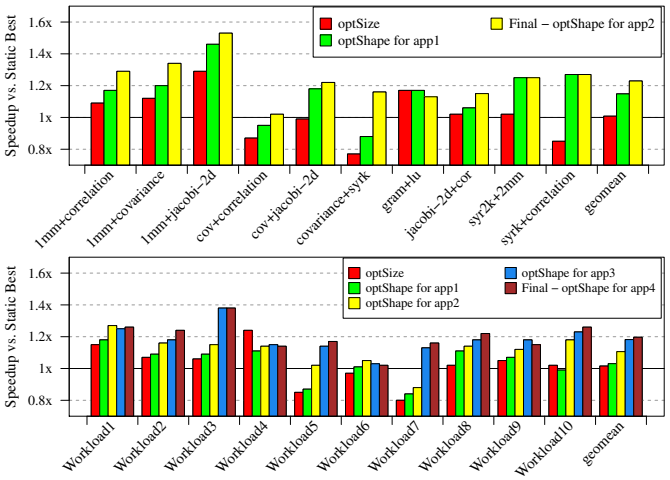

Fig. 10. ShapeShifter co-optimization retiles multiple co-runners resulting in better cache usage

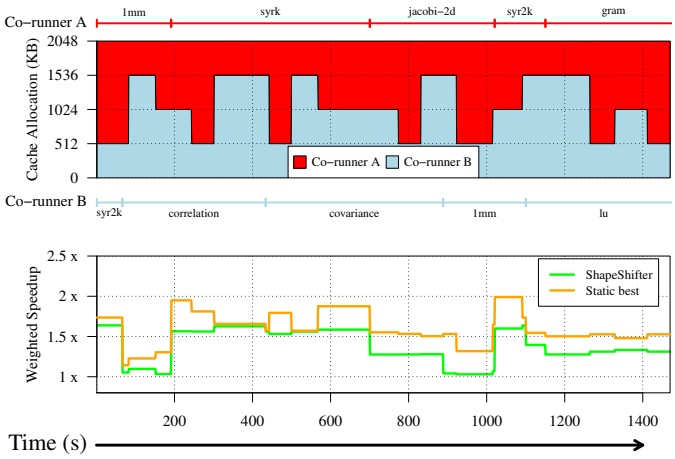

Fig. 11. ShapeShifter co-optimization continuously adjusts co-runner tiles to changing runtime environment

Workload 1	`1mm`, `covariance`, `gram`, `lu`
Workload 2	`jacobi-2d`, `covariance`, `correlation`, `1mm`
Workload 3	`correlation`, `syr2k`, `syrk`, `jacobi-2d`
Workload 4	`gram`, `lu`, `jacobi-2d`, `1mm`
Workload 5	`2mm`, `syr2k`, `covariance`, `1mm`
Workload 6	`jacobi-2d`, `2mm`, `syrk`, `correlation`
Workload 7	`covariance`, `correlation`, `1mm`, `2mm`
Workload 8	`jacobi-2d`, `1mm`, `correlation`, `covariance`
Workload 9	`syrk`, `syr2k`, `covariance`, `correlation`
Workload 10	`1mm`, `correlation`, `jacobi-2d`, `covariance`

TABLE III. Co-runner workloads of 4 applications

performance improvements of up to 1.5× (covariance vs. bubble), and an average improvement of 1.1× on average.

In the second and third experiments, we demonstrate the capability of ShapeShifter co-optimization to accurately select tiling strategies in the presence of two and four co-running applications. The results of this experiment are present in Figure 10. Co-optimization works by first finding the right tile size for each co-runner, then optimizing each application tile shape one-by-one. The figure shows step-by-step speedup during this co-optimization process for two and four co-runners across 10 different workloads (Table III). We observe that ShapeShifter co-optimization achieves performance improvement of up to 1.5× , with an average of 1.2× in both the scenarios. We also experimented with running ShapeShifter after all applications have been optimized once. We observed that the additional benefits were negligible, supporting the key insight that different tile shapes of same tile size does not have a large effect.

Dynamically Changing Workloads. In this experiment, we evaluate ShapeShifter co-optimization on a dynamically changing runtime environment that demonstrates how it adapts to the dynamism. In this experiment, at any given time there are 2 co-runners sharing a cache. These co-runners change with time along with the cache allocated to them as shown in Figure 11(a). ShapeShifter weighted speedup is compared against the weighted speedup obtained by running co-runners aggressively tiled with Static Best strategy.

The result of this experiment are presented in Figure 11(b). We observe that ShapeShifter continuously adapts to changing runtime environment, finding a suitable tiling strategy for both the co-runners at different cache allocations. It results in significant speedup as compared to Static Best tiling strategy.

D. Microarchitectural Factors

In current systems, the architectural/microarchitectural parameters can change during the application execution. Here, we evaluate ShapeShifter on cache resizing and platform changes.

1) Cache Resizing: We begin by exposing applications to diverse situations in which different amount of cache are available. To conduct this experiment, we configure the way-locking feature on the AMD Bulldozer to leave either 8 or 4 ways open, then run the application with ShapeShifter to allow ShapeShifter to realize an aggressive tiling configuration on that microarchitectural configuration. The results of this experiment are presented in Figure 12, which is again normalized to application performance when the application is compiled to employ the Static Best tiling configuration.

We see large performance improvements over the Static Best strategy. When 8 ways are available to the application, ShapeShifter achieves performance improvements of 1.2× on average and up to 1.7×. This contrast becomes more stark when only 4 ways are available to the application, where an even larger gap exists between the optimal tiling strategies between the 4-way and 16-way configurations. In this case, ShapeShifter achieves a speedup of 1.4× over Static Best, with a maximum speedup of 2.4× on gemm.

2) Microarchitectural Diversity: We next examine how ShapeShifter deals with significant microarchitectural diversity, applying it on applications running on Intel Haswell and Intel Atom platforms. Re-tiling occurs in ShapeShifter

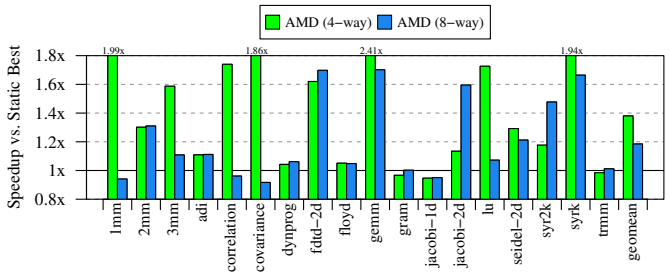

Fig. 12. ShapeShifter demonstrates significant speedup by retiling for available cache size

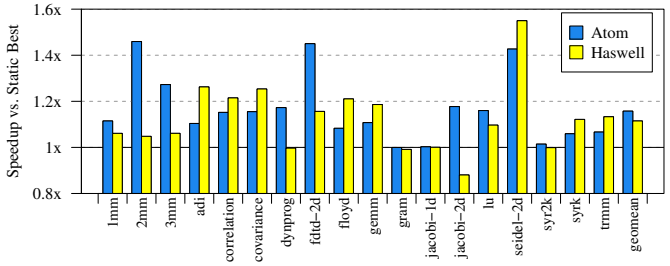

Fig. 13. ShapeShifter shows sizable speedup by retiling for different microarchitectures

as the application begins execution, arriving at an aggressive tiling strategy for the specific microarchitecture. As a point of comparison, we also measure the performance when running applications that are tiled using the Static Best strategy on the Haswell and Atom.

The results of the experiment are presented in Figure 13, phrased as the performance improvement achieved by ShapeShifter over Static Best. These results demonstrate the effectiveness of ShapeShifter at developing aggressive tiling strategies across multiple microarchitectures, with a performance improvement of up to 1.5× when running seidel-2d on the Haswell system, an average performance improvement of 1.1× on Haswell, and a 1.1× average performance improvement on Atom.

E. Overhead Analysis

We now present the ShapeShifter runtime overhead. Companion threads use a small amount of compute and memory resources to dynamically compile new versions of code. This causes interference to the primary application which can in turn lead to slowdown. We term this slowdown due to interference as *dynamic compilation* overhead. In addition, whenever application is redirected to newly compiled tile, it suffers an I-cache warmup phase. We refer to this overhead as *code redirection* overhead. Finally, there is the online *training* overhead. In this section, we provide quantitative analysis of these overheads.

Dynamic compilation. In order to calculate just the dynamic compilation overhead, we design a stress test experiment where Companion thread continuously generates new tile variants without redirecting the application to the generated

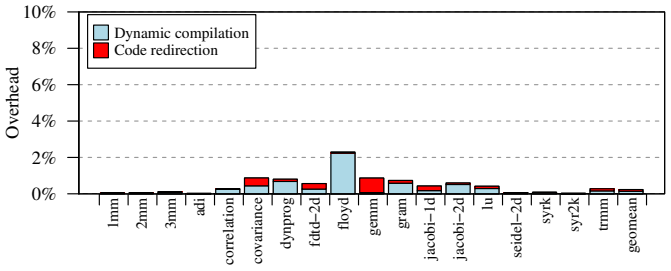

Fig. 14. Runtime overhead of ShapeShifter dynamic compilation infrastructure

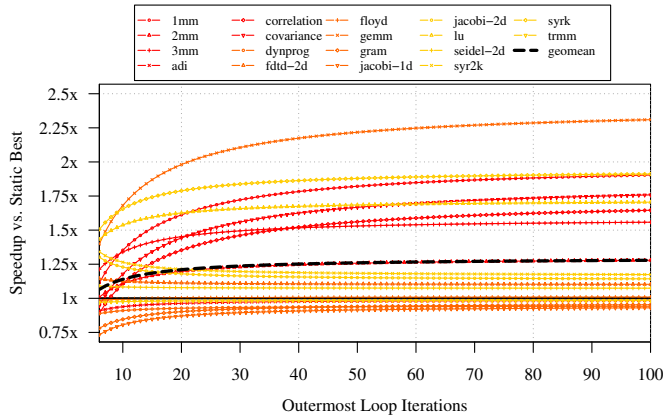

Fig. 15. Performance benefits of ShapeShifter as a function of the how long the environment remains stable

code. The associated overhead in this case is the worst case dynamic compilation overhead. Next, we allow application redirection to new tile variants. The difference between the former and latter experiment quantifies the code redirection overhead. We show these overheads in Figure 14. We observe that even in the stress testing, the overhead is minimal and less than 1% on average.

In terms of absolute numbers, we found that ShapeShifter takes 136 (336) ms on average with maximum of 430 (990) ms on Intel haswell (AMD Bulldozer) across our benchmark suite while the application is running on other core.

Training. As a part of the tile selection algorithm, ShapeShifter runs a handful of diverse tiling strategies for training, which can be less performant than the final tile chosen. In this section, we weigh the overhead of that training against the benefit obtained by running an optimized tiling strategy. Our experimental setup is to run each application with ShapeShifter for a number of iterations in a stable environment (the AMD Bulldozer with 8 ways locked), measuring the performance of the application over time as the training and the final selected tiles are run.

The results are presented in Figure 15, which presents the performance of each application normalized to the Static Best approach (y-axis) over a number of iterations in the application's algorithm (x-axis). The results show that the

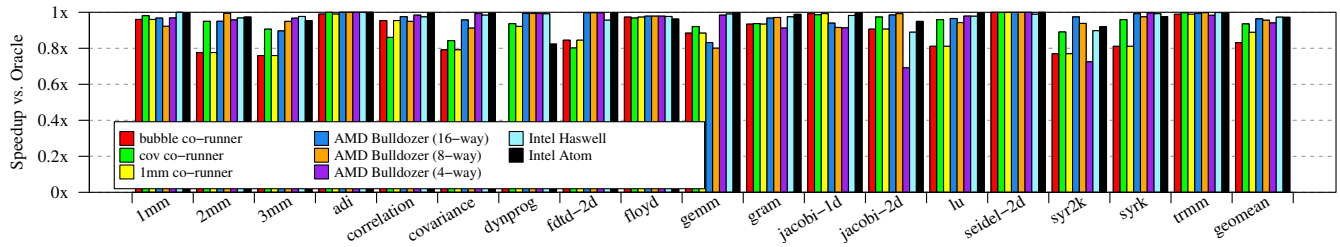

Fig. 16. Comparison of ShapeShifter against a dynamic oracle, that chooses the ideal tiling strategy with no overhead; ShapeShifter achieves 93% of the performance of the dynamic oracle

performance improvement achievable by ShapeShifter depends on the amount of time the application stays in a stable environment. For example, immediately after training (5 iterations), the average performance improvement over Static Best across applications is 1.08×, while after just 20 iterations, substantially higher performance of 1.2× is realized.

F. Comparison to Dynamic Oracle

Our final point of evaluation is to compare the performance achieved by ShapeShifter across a number of different runtime environments to the performance achievable by a dynamic oracle approach to tiling. To execute this experiment, we run each application in the prescribed environment using each of a large set of tiling strategies. Afterward, we choose the best-performing tiling strategy from among them and call this the measured performance of the dynamic oracle.

Figure 16 presents the results of this experiment. Across all applications and runtime environments, ShapeShifter achieves 93% of the dynamic oracle's performance on average (no worse than 72%). This demonstrates that ShapeShifter is effective in finding suitable tiling strategies across different runtime environments.

VII. Related Work

Prior research in finding the best tile size can be divided into two categories: static techniques that develop a detailed analytic model for a set of host environments and predict a tile [4, 5, 6, 32], and dynamic techniques that use a model to prune the search space, execute a subset of tiles and choose the one with the least execution time [33, 34, 35, 36].

Static Techniques. This class of methods take an approach of developing detailed white box analytic models for the applications and runtime environments. TSS [7] studies how tiling interacts with several level of caches. Defensive tiling [8] considers tiling strategy in presence of last-level cache interference, with the goal of reducing the number of inclusion victim misses. Coleman and McKinley [5] develop a cache model to find the largest tile that suffers from minimum self-interference misses. These models can deliver useful insights about how applications interact with the runtime environment. However, it is difficult and sometimes intractable for the white box approaches to accurately model the complex set of factors that impact the choice of tiling strategy. ShapeShifter differs from these techniques as it creates a model on-the-fly.

Yuki et al. [24] discuss the limitations of the white box approaches. They use a neural network to statically predict a tile for an application. However, it is a completely static technique unable to adapt to changing runtime environment. In addition, they limit the search to only square tiles to reduce the large training time, leaving a significant performance opportunity on the table.

Dynamic Techniques. Reactive tiling [9] is a combined static and dynamic technique that compiles an application with a fixed set of tiling parameters and inserts mechanisms in the code to switch between this set of tiles at runtime. Reactive tiling focuses only on the scenarios where the cache is resized during the application execution as opposed to ShapeShifter that accounts for a wide range of sources of dynamism. We compare ShapeShifter against reactive tiling in this particular scenario of cache re-sizing in Figure 17. In this experiment, we generate a time schedule of changing cache sizes where the cache size is chosen randomly between 1x, 1/2x and 1/4x of the cache size during the application run. We observe that ShapeShifter achieves 10% speedup against reactive tiling as reactive tiling is limited by the set of tiles available to it at compile time.

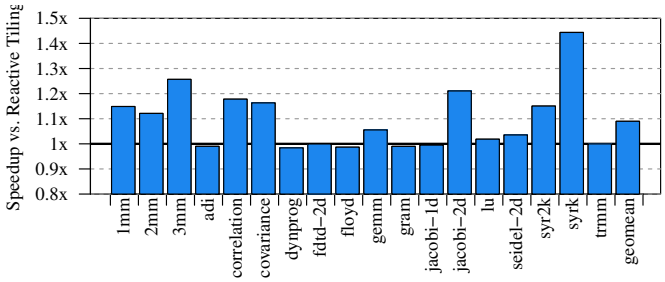

Fig. 17. Improvement of ShapeShifter over Reactive tiling on a dynamic schedule for all applications

Some prior works mitigate the complexity of searching by resorting to only square tiles (i.e., all tiling parameters must be equal) [24, 34]. Such limitations are fundamental as they exclude valuable tiling configuration possibilities. We observed that performance difference between the best rectangle tile was 1.11× (up to 1.5×) faster than best square for our test applications. The ATLAS library generator [34] executes a wide range of tiles on the target machine and chooses the one

with the best performance. However, the optimized kernels cannot react to sources of dynamism.

VIII. Conclusion

This paper introduces ShapeShifter, a dynamic compilation strategy that removes the risks of applying cache tiling by dynamically re-tiling running application code. ShapeShifter is designed to continuously monitor running applications and their runtime environments to find tiling opportunities and pinpoint near-optimal tile sizes. Upon finding such a tiling opportunity, ShapeShifter quickly generates an optimal tiling code for the application, then that code is seamlessly stitched into the running application with near-zero overhead. We evaluate ShapeShifter on real systems amidst three classes of runtime environment changes spanning different co-running applications, platforms, and dynamically shifting architectural resources. Our evaluation shows that ShapeShifter achieves sizable speedups across applications, averaging 1.1-1.4× across different runtime environments.

Acknowledgement

We thank our anonymous reviewers for their feedback and suggestions. This research was supported by National Science Foundation under grants NSF-CAREER-1553485, CCF-SHF-1302682 and CNS-CSR-1321047.

References

[1] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Programming Language Design and Implementation (PLDI)*, 1991.

[2] M. Wolfe, "More iteration space tiling," in *Conference on Supercomputing (SC)*, 1989.

[3] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical automatic polyhedral parallelizer and locality optimizer," in *Programming Language Design and Implementation (PLDI)*, 2008.

[4] M. D. Lam, E. E. Rothberg, and M. E. Wolf, "The cache performance and optimizations of blocked algorithms," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1991.

[5] S. Coleman and K. S. McKinley, "Tile size selection using cache organization and data layout," in *Programming Language Design and Implementation (PLDI)*, 1995.

[6] S. Carr, K. S. McKinley, and C.-W. Tseng, "Compiler optimizations for improving data locality," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1994.

[7] S. Mehta, G. Beeraka, and P.-C. Yew, "Tile size selection revisited," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2013.

[8] B. Bao and C. Ding, "Defensive loop tiling for shared cache," in *Code Generation and Optimization (CGO)*, 2013.

[9] J. Srinivas, W. Ding, and M. Kandemir, "Reactive tiling," in *Code Generation and Optimization (CGO)*, 2015.

[10] S. Tavarageri, L. Pouchet, J. Ramanujam, A. Rountev, and P. Sadayappan, "Dynamic selection of tile sizes," in *High Performance Computing (HiPC)*, 2011.

[11] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *International Symposium on Microarchitecture (MICRO)*, 2011.

[12] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[13] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang, "Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers," in *International Symposium on Microarchitecture (ISCA)*, 2014.

[14] C. Gao, A. Gutierrez, M. Rajan, R. G. Dreslinski, T. Mudge, and C.-J. Wu, "A study of mobile device utilization," in *International Symposium on the Performance Analysis of Systems and Software (ISPASS)*, 2015.

[15] K. Gillespie, H. R. Fair, C. Henrion, R. Jotwani, S. Kosonocky, R. S. Orefice, D. A. Priore, J. White, and K. Wilcox, "Steamroller: An x86-64 core implemented in 28nm bulk cmos," in *Solid-State Circuits Conference (ISSCC),*, 2014.

[16] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: memory power estimation and capping," in *International Symposium on Low-Power Electronics and Design (ISLPED)*, 2010.

[17] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *International Symposium on Microarchitecture (ISCA)*, 2006.

[18] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *International Symposium on Computer Architecture (ISCA)*, 2011.

[19] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: An introduction to the design of warehouse-scale machines," *Synthesis lectures on computer architecture*, 2013.

[20] J. Mars and L. Tang, "Whare-map: heterogeneity in homogeneous warehouse-scale computers," in *International Symposium on Computer Architecture (ISCA)*, 2013.

[21] "Amazon EC2 Spot Instances," http://aws.amazon.com/ec2/purchasing-options/, 2016, online; accessed 5-Aug-2016.

[22] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," *URL: http://www. cs. ucla. edu/pouchet/software/polybench*, 2012.

[23] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean code: Achieving near-free online code transformations for warehouse scale computers," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[24] T. Yuki, L. Renganarayanan, S. Rajopadhye, C. Anderson, A. E. Eichenberger, and K. O'Brien, "Automatic creation of tile size selection models," in *Code Generation and Optimization (CGO)*, 2010.

[25] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August, "Automatic cpu-gpu communication management and optimization," in *Programming Language Design and Implementation (PLDI)*, 2011.

[26] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Parallel frame rendering: Trading responsiveness for energy on a mobile gpu," in *Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[27] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[28] S. Srinath, B. Ilbeyi, M. Tan, G. Liu, Z. Zhang, and C. Batten, "Architectural specialization for inter-iteration loop dependence patterns," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[29] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell, and S. W. Redder, "Priority-based cache allocation in throughput processors," in *High Performance Computer Architecture (HPCA)*, 2015.

[30] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly – performing polyhedral optimizations on a low-level intermediate representation," in *Parallel Processing Letters*, 2012.

[31] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Code Generation and Optimization (CGO)*, 2004.

[32] V. Sarkar and N. Megiddo, "An analytical model for loop tiling and its solution," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2000.

[33] J. Xiong, J. Johnson, R. Johnson, and D. Padua, "SPL: A language and compiler for dsp algorithms," in *Programming Language Design and Implementation (PLDI)*, 2001.

[34] R. C. Whaley, A. Petitet, and J. J. Dongarra, "Automated empirical optimizations of software and the atlas project," *Parallel Computing*, 2001.

[35] C. Chen, J. Chame, and M. Hall, "Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy," in *Code Generation and Optimization (CGO)*, 2005.

[36] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick, "Self-adapting linear algebra algorithms and software," *Proceedings of the IEEE*, 2005.