Deepframe: A Profile-driven Compiler for Spatial Hardware Accelerators

Apala Guha, Naveen Vedula, Arrvindh Shriraman
School of Computing Science, Simon Fraser University
Email: {aguha,nvedula,ashriram}@cs.sfu.ca

***Abstract*—Tracing code paths to form extended basic blocks is useful in many areas, compiler optimizations [1], improving instruction cache behavior [2] and custom-hardware offloading [3]. Prior work has been plagued by small traces, limited either by the overheads of dynamic profiling, statically available information [4], or side-exit branches [5]. In this work, we rethink what code path sequences to fuse and construct long traces for offloading to spatial accelerators, while minimizing the occurrence of side exits which limit dynamic coverage.**

We introduce a novel technique that recasts learning a program's execution patterns as a natural-language-processing problem, CBOW (Continuous Bag of Words). We then use a deep learning network to learn the relationships among paths. During the compilation phase, the compiler uses a sequence miner to decide what paths are likely to occur. The learning network predicts a Deepframe online, which is an extended basic block comprising a multi-path sequence (each path itself is composed of multiple basic blocks). We demonstrate the efficacy of Deepframe on spatial hardware accelerators and find the following: i) Deepframe can construct up to 5× (max: 27×) longer offload regions compared to prior approaches. ii) Surprisingly far–flung ILP (instruction-level parallelism) and MLP (memory-level parallelism) can be mined from the frames statically (5.5× increase in ILP and 10.5× increase in MLP). iii) The frames offloaded to the spatial accelerator have minimal side exits (mis-speculation) and achieve sufficient dynamic coverage to improve overall application performance (up to 9× improvement). We will be releasing open-source our end-to-end compiler prototype based on LLVM.

I. Introduction

The Problem.: In this paper, we develop a frame compilation toolchain for hardware accelerators. Frames [1] serve the same purpose as traces or superblocks [5] within a Trace Scheduling compiler [6]. A frame is a single-entry single-exit sequence of basic blocks that that either atomically runs to completion or fails (due to side-exits) and falls back to the original path. Frames form effective scheduling windows by straight-lining code [4], [7]–[9] and enable the extraction of instruction and memory parallelism in OOO (out-of-order) processors [10], VLIW processors [11] and, more recently, hardware accelerators [3]. This latter use case is our target.

We choose to spatial hardware accelerators as our target platform because these implicitly rely on heterogeneous execution (Figure 1) to support the whole program. Heterogeneous execution depends on three factors i) the overhead of migrating between the OOO and the accelerator and ii) how often the accelerator execution has to fall back to the OOO due to control-flow divergence, iii) and whether we are able to statically find program phases that are large enough to map to the accelerator. In fact, current hardware accelerators restrict

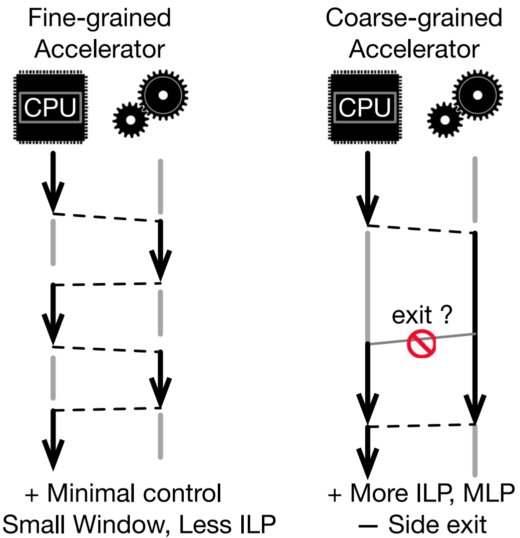


Figure 1: Accelerator execution model is heterogeneous and interleaved with CPU. Fine-grained accelerators have lower ILP (instruction parallelism) and MLP (memory parallelism) as well as fewer side exits. Coarse-grained accelerators improve ILP and MLP, but may have side exits.

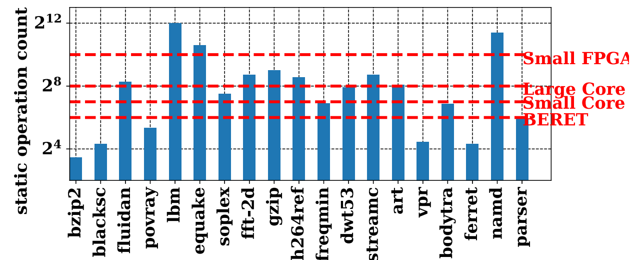


Figure 2: Accelerator hardware frame size supported vs. Superblock sizes [5], [12].

themselves to statically scheduling coarse-grain frames from applications with perfect loops. In this paper, we address the question central to heterogeneous execution. How can we develop a generalized compiler for accelerators that can create coarse-grain frames that mine instruction-level parallelism from a diverse application set?

Many current accelerator designs continue to rely on seminal work in VLIW [5], [13], [14] and dataflow processors [15] which also segment the execution into frames. However, there are important differences that limit the effectiveness of prior

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00014

IEEE computer society

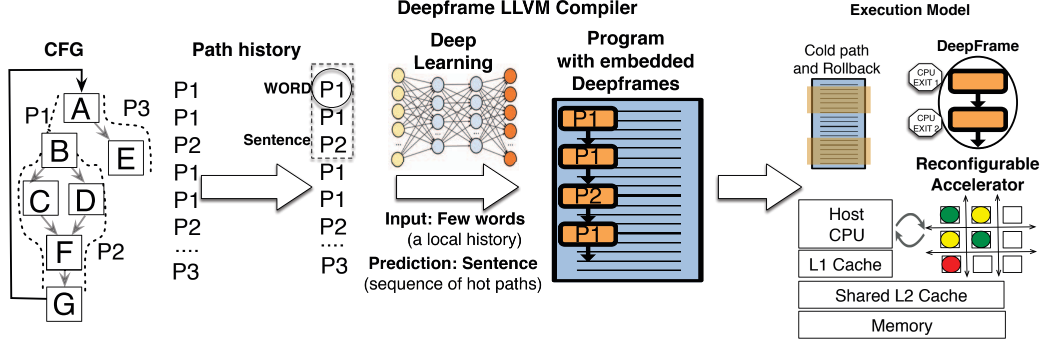


Figure 3: Constructing Deepframes from path execution histories. Frames are statically mapped to the accelerator, while next frames are predicted dynamically.

techniques. We enumerate the differences below.

(i) **Accelerators support coarse-grain frames.** Hardware accelerators are typically a spatial grid of function units onto which the compiler maps the dataflow of a frame. The size of the spatial grid determines the size of the frame the compiler is expected to construct to effectively exploit the hardware. With increase in transistor density current accelerators to support a larger grid of function units, 2^{12} —2^{20} [16], [17] operations and hence increase pressure on the compiler to find frames with more operations. To understand the limitations of prior work, we compared superblock sizes against various frame sizes supported by different accelerators (Figure 2). We study if prior work [5], [18], [19] can fill the available hardware resources. In Figure 2, BERET [12] represents a fine-grain accelerator while a small FPGA is closer to a coarse-grain accelerator. In many workloads, prior techniques are unable to identify offload opportunity beyond a target window of 64—128 instructions. While this window size is sufficient for fine-grain accelerators which have window sizes similar to small cores and VLIW CPUs etc. [15], [19], [20], they will lead to hardware under-utilization in coarse-grained accelerators.

(ii) **Accelerators typically rely on static construction.** In prior work the set of frames could potentially be updated at runtime. Accelerators need to be reconfigured if/when a frame changes and this is expensive. Hence, the objective is to statically construct a coarse-grain frame.

(iii) **Accelerators rely on coarse-grain execution.** The overhead and frequency of migrating dictate that coarse-granularity (1000s of operations) must be offloaded to overcome the overhead of execution migration. We need to create the offload frame a priori such that side exit events are few and migration overheads are low. The key challenge with the accelerator execution model is control divergences that lead to side-exits and reverting to the CPU. Accelerators have higher penalties due to aggressive static optimization, and mis-speculation rates are exacerbated by larger code regions. This concern on control divergence has steered much of the compiler effort in prior work toward fine-grain accelerators [19], [20].

Table I: Simulation Parameters

Host CPU Cache	2 GHz, 4-way OOO, 96 entry ROB. 32 entry LSQ. L1: 64K 4-way, 3 cycles. Shared L2 : 4M shared 16 way, 25 cycles. MESI Directory. Memory 200 cycles.
Accelerator	
CGRA **Energy** [23]	32× 32 homogeneous units [21], [22] Network: 600 fJ/link. ALU : 500 fJ/INT, 1500 fJ/FP.MDE. May: 500 fJ/edge Must: 250 fJ /edge
LSQ (§6)	2 port 48 entries/bank. # banks 2—8. Loads:2500 fJ, Stores: 3500fJ

(iv) **Accelerators can mine wider instruction parallelism.** Compared to front-end limited processors, spatial accelerators adopt dataflow execution [16], which can support wider instruction parallelism since it is not limited by the sizes of issue queues and register file ports. This instruction parallelism has to be mined statically, and coarser-granularity frames afford more opportunity for mining instruction parallelism.

Our Proposal.: We propose *Deepframe* to statically select and dynamically predict *frames* for acceleration (Figure 3). The input to Deepframe is path execution history from reference runs. The output of Deepframe is, (i) a set of frames that should be statically offloaded to the accelerator, and, (ii) a predictor that dynamically forecasts the next frame (a sequence of paths) that should be offloaded to the accelerators. To achieve this, Deepframe trains a neural network on the execution histories.

Figure 3 visualizes the process. The program control-flow graph (CFG) exhibits paths *P1 (ABCFG), P2 (ABDFG), P3 (AE)* with potential side exit *AE* from paths *P1* and *P2*, *BD* from *P1*, and *BC* from *P2*. Our frames are sequences of *Ball-Larus paths* (or just paths) [2], which can be interpreted as maximal superblocks (i.e., not limited by hardware constraints) encompassing at most one loop iteration, but never crossing a loop boundary. In this example paths may combine to form frames such as *P1 P1 P2*. As frame length increases and encompasses multiple path sequences, the potential for side exits increase. On the other hand, longer frames improve instruction and memory parallelism. The goal

of our frame selection is to balance these opposing forces.

We answer several questions that enable the compiler to automate the construction of frames for hardware accelerators. i) *Can we balance the size of the frame and coverage (fraction of execution within frames) in the presence of control flow?*. ii) Assuming that large frames with good coverage exist, *can we dynamically predict next frames with high accuracy (no side exits)?* iii) Finally, *will larger frames necessarily expose more ILP (instruction-level parallelism) and MLP (memory-level parallelism) that an accelerator can leverage?* Deepframe is an end-to-end compiler toolchain which provides support for frame construction, spatial accelerator mapping and frame prediction. Deepframe is built on LLVM [24] and will be released open-source.

The insight underlying Deepframe is that analyzing control-flow patterns with static/dynamic features is complex, leads to inaccuracies, and loss of portability. We can find an effective feature-less analysis technique technique for our problem in natural language learning [25]. Natural language learning characterizes languages by observing word associations (relative positions and sequences) with zero a priori knowledge of the language, and without any user input. This is known as latent feature selection. We apply the same principles to Deepframe where we view the path execution patterns as the application's language, and the paths themselves as words. We select as well as predict frames without any prior assumptions. We encode program execution patterns as a classical natural-language-learning, Continuous-Bag-of-Words (CBOW) problem and then rely on modern deep learning networks to effectively learn the recurring patterns in each application.

As shown in Figure 3, Deepframe employs an offline compilation stage before execution. In the first stage, the baseline compiler instruments the original program for generating an execution history at path granularity [2], [5]. We then chunk the execution history into a collection of *word-grams* or *sentences*, each distinct path in the execution history being a word. Each gram or sentence corresponds to a frame. The length of the gram is a compiler parameter. We train a deep learning network offline on reference runs. Additionally, we mine frequent grams from reference runs to help the compiler effectively form frames for offloading to the accelerator. We show that these multi-path frames form effective optimization targets for an accelerator. We dynamically predict frames to execute. In summary, our contributions are:

- **Multiple hot path sequencing:** We develop, Deep-Frame, a LLVM-based compiler that leverages state-of-the-art machine learning to analyze the dynamic behavior of programs for frame construction. The frames are large (up to 27× a superblock) and effectively sequence multiple hot paths; prior work largely focused on constructing a single hot path.
- **End-to-End Frame construction for Accelerators:** We develop a complete compiler back-end that utilizes a latent feature selecting deep learning network to construct and re-optimize frames. The constructed frame is mapped to a spatial accelerator.
- **Enabling far-flung ILP and MLP to improve performance:** We find that Deepframe can increase the ILP by up to 5.5× and the MLP by up to 10.5×. We demonstrate that Deepframe can achieve performance improvements of up to 9× for hot frames, compared to offloading at path granularity.
- **Programmer-Friendly, adaptable Compiler:** Deepframe is a black-box compiler that does not require the user or compiler-expert to hand-engineer code selection and prediction heuristics. It is driven entirely by the observed execution histories of the program.

We provide detailed context on the problem of code framing for accelerators in Section II. Section III describes Deepframe in detail. We evaluate coverage, performance and accuracy of Deepframe in Section IV, Section V, and, Section VI respectively. We conclude in Section VII.

II. Scope and Related Work

In this section, we provide context to our work. We describe our experimental framework in Section II-A. Next, we characterize 20 SPEC and PARSEC benchmarks in terms of control flow and the implications for framing in Section II-B. Finally, we present related work on code region construction in Section II-C. Section II-D presents related work on learning–based compilers.

A. Target Spatial Accelerator and Execution Model

Here we provide a brief overview of the baseline spatial architecture. Detailing the internals of spatial accelerators [12], [16], [26] is challenging within the page limits. We briefly summarize below. The accelerator we deploy is a loosely coupled CGRA with 32×32 homogeneous functional units similar in design to Dyser [26]. The accelerator includes its own private cache and is cache coherent with the host CPU through the shared L2 cache. Each functional unit in the 32×32 grid maps a single instruction from the dataflow graph of the offloaded frame (See the reconfigurable accelerator in Figure 3). The data dependencies between the operations are explicitly routed over a static mesh operand network. The operand network also routes values from the cache at the edge of the grid to the function units in the grid. We rely on previously released software for mapping the LLVM IR of the offload frame to the CGRA and configuring the operand network [20]. We use a recently released CGRA simulator for cycle-by-cycle timing [27]; the OOO host core is modeled in detail using macsim [28]. See Table I for the simulation set-up.

Deepframe constructs the offload frame that is mapped to the CGRA. Since the frame is free of control flow, compiler has to only map the dataflow to the spatial grid. A mapped frame consists of three components: the *dataflow block*,

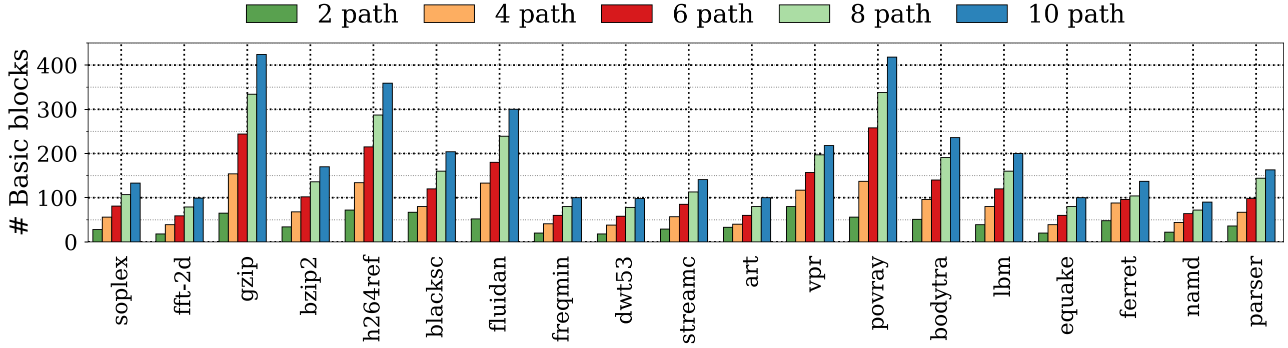


Figure 4: Number of basic blocks in the hottest frame as the constructed frame size is varied from 2—10 paths. The number of basic blocks is equal to the number of static branches. A path is a superblock.

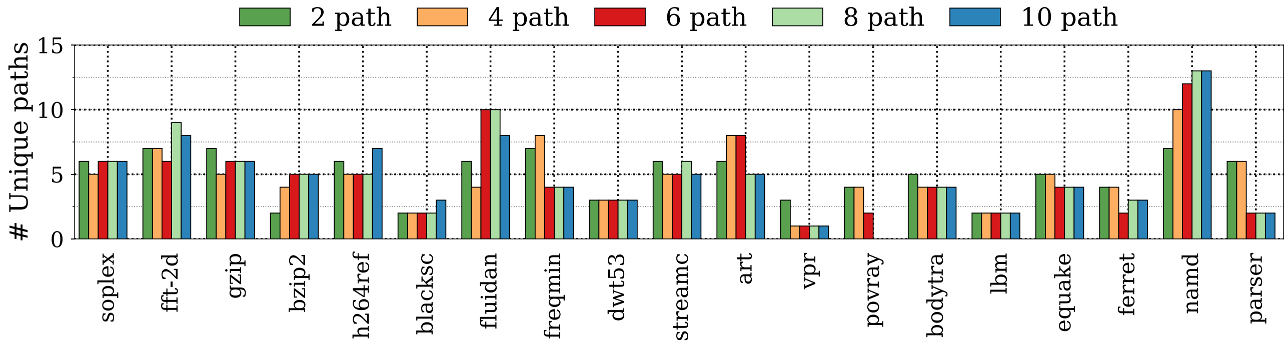


Figure 5: Number of unique paths in top 5 hottest frames. Frame size is varied from 2—10 paths. A path is a superblock.

which is a block of operations to run on the accelerator, the *guards* which check for side exits, and the *store buffer* which captures memory writes by the frame to quash in case of speculation failure. The compiler is permitted to move instructions within the frame and find the requisite ILP. When a guard is triggered and the side-exit is taken during a frame's execution, the externally visible state has to be reverted. Note that no architectural state is shared between the frame and host processor; live values and memory operations are the only form of communication to and from the frame. Deepframe implements full rollback using a store buffer.

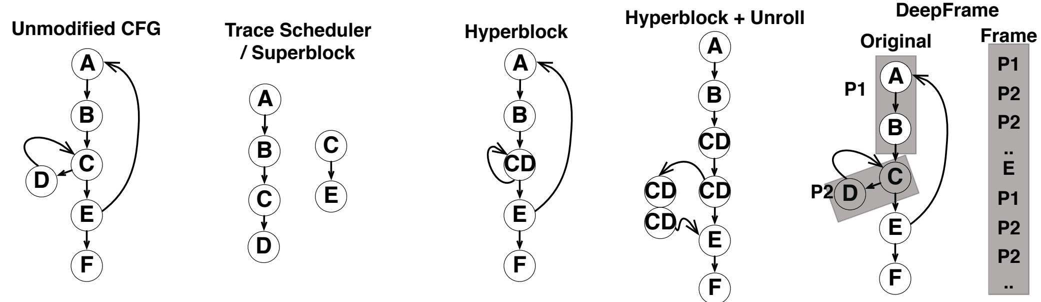
B. Quantifying control flow in programs.

We quantitatively analyze the control flow in programs in Figure 4 and Figure 5, to motivate our research. Figure 4 shows the number of basic blocks (or branches) in the hottest frame for lengths of 2–10 paths. In figure 4 we see that the number of branches scales linearly with frame length, and straddle up to 400 static branches in a frame. This is well beyond the capacity of existing software and hardware instruction windows. Yet, applications execute many frames that could benefit from instruction windows that speculate hundreds of branch outcomes.

To understand the source of these large frames and their branch biases we study the number of unique paths that manifest at runtime. One could argue that the large branch counts are indeed there, but they only come from highly biased loops i.e. loops that trace the same paths in a large majority of the iterations, such that loop specialization and unrolling is sufficient. We look at Figure 5, which shows the number of unique paths in the 5 hottest frames for a fixed frame length. Three cases are possible here. A unique path count of 5 signifies that each frame was formed from one or more copies of a single path. When the unique path count is less than 5, it indicates that the same loop has produced frames with different path combinations. Finally, a unique path count greater than 5 indicates multiple loops, at least some of which yield frames with different path combinations. All three cases are present in our benchmarks, indicating the need for developing framing techniques that are free of prior assumptions about frame length and composition.

C. Related Work

Our main contribution is an end-to-end compiler for constructing frames [1]. Here, we briefly review the concept of frames. Deepframe constructs single-entry single-exit regions that enable aggressive speculative execution on accelerators. The ILP extracted by a compiler [4] is highly dependent on the control characteristics and the size of a code region. Removing control flow creates opportunities for finding parallel instructions [8], [9]. Many solutions exist to form control-free regions. To guide the reader through the paper we



		JIT Compiler	Superblock	Hyperblock	Deepframe
		[9], [29]–[32]	[5], [19], [33]	[14], [15], [34]	
Scope	Unit	Basic block	Basic blocks	Basic blocks	Paths
	Target	Sequence of basic blocks		① Multiple superblocks	Multiple superblocks
	Feature	Single-entry Multi-exit			Single-entry Single-exit
Design	Branches	No (guards)		Yes ②	No
	User defines heuristics	Yes [29]			No
	Compile-time	Dynamic	Static/Profile		Profile
	Support for Multi-Superblocks	No		③ Yes	Yes
	Support for Loops	Yes (backward branch assumed taken)			Yes (Profile)
	Unbiased Branches	No (Side Exits and Superblock fragmentation)		Yes (Merge superblocks) ④	Yes (path duplication)
Spec. and Opt.	Side-exits	Yes (duplicate tail)			No
	Rollback overhead	High	Depends (compensation code)		High (fallback to unopt)
	IR optimization	No	Yes (implemented within compiler)		
① Large Region ② High Optimization opportunity. ③ Good Coverage ④ No Cold Ops					

Figure 6: Comparison of different types of code regions.

clarify the following terms. *Traces* are multi-entry multi-exit regions [35], [36]. A *frame* is a specific type of trace with a single entry and a single exit. *Superblocks* [5] are single-entry multi-exit regions and *hyperblocks* are single-entry multi-exit (with internal control flow).

Figure 6 shows the popular existing strategies for superblock extraction and compares them against Deepframe. Superblocks [5], [19], [33] are a static approach which elides backward branches as taken. The key limitation of superblocks is their handling of branches that are not entirely biased. For instance, in the example CFG, every third iteration takes the superblock *CE*, the branch deviates from the dominant *CD* superblock. Superblocks will not recognize this and cannot predict which specific superblock will execute in a given loop iteration. Superblocks make the decision of including a basic block based on heuristics that reflect the dynamic execution count of the block. Unfortunately, as the superblock targets longer regions, the heuristics make locally optimal decisions that increase the likelihood of side exits. Typically backward branches are folded into the superblock; however how to control the unrolling is not clear. Superblocks use heuristics [9], [29]–[32] to limit their sizes and we find that in our workloads, typical superblock sizes are 4—6 basic blocks; see Figure 4. Superblocks limit optimization scope because they allow side exits, and also increase side exit events by making fixed assumptions on branch types.

Hyperblocks [14], [15], [34] were invented to handle the limitations of superblocks and capture multiple superblocks that a non-biased branch may follow. They effectively merge all the superblocks that a loop may potentially execute. In the example, both *CD* and *CE* are folded into the hyperblock. Unfortunately, they introduce control flow (which limits the optimization opportunities) and may include cold blocks since they merge multiple superblocks. Overall, existing works have primarily studied small inner loops with regular control flow. Hyperblocks are capable of producing large regions as they capture multiple code superblocks to avoid side exits. This leads to a significant fraction of operators remaining idle (see Figure 7). The hyperblocks in Figure 7 were formed by fusing paths with common prefixes. In the accelerator context, hyperblocks inhibit ILP by preserving control instructions, and occupy space that is under-utilized during execution.

Deepframe fuses sequences of superblocks, or paths (which are themselves composed of multiple basic blocks) based on dynamic profile. This leads to frames 25× larger than superblocks (see Section IV). Deepframe does not include any wasted operations since frame execution is atomic (all or none of the operations execute). The utility of any frame

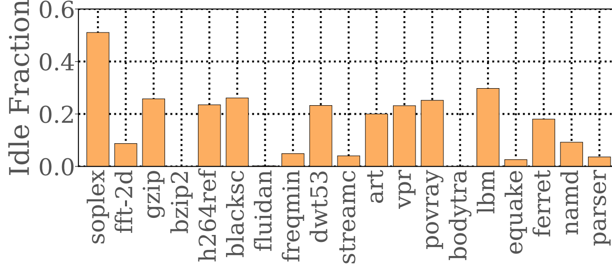


Figure 7: Fraction of idle basic blocks in hyperblocks.

depends on what fraction of a dynamic execution can be captured by the the frame. A key factor impacting dynamic coverage is the multiplicity of frames that originate from the same branch head. For example, *ABCDE* and *ABCE* are two frames that start at the same branch. Deepframe maintains coverage by being selective about frame construction and learning from dynamic profiles. Finally, paths and frames can leverage the exact same set of transformations as superblocks (e.g. loop unrolling), as well as achieve the best configuration (e.g. unroll factor), in the absence of these transforms.

D. Learning-based Compilers

Static compilers have previously leveraged machine learning for predicting which basic blocks should constitute a superblock [37], [38]. Prior works on static hot superblock prediction show feasibility in detecting individual hot superblocks using whole-program techniques [39] and other heuristics [40]. Buse and Weimer [41] and more recently Zekany et al. [38] apply machine learning algorithms to learn statistical models with features extracted from the program. Both these works rely on learning static features to predict which basic blocks should constitute a single hot superblock. There has been other work that has looked at function-granularity traces [42]. Tetzlaff and Glesner [43] also apply machine learning to the problem of predicting loop iteration count. Our work focuses on the more general notion of frames and sequences of paths.

Finally, while there has been prior work on learning branch prediction [44], the goals and deployment of branch prediction (Out-of-order processor) are entirely different. In contrast our work relies on learning dynamic path patterns and also constructs frames from a sequence of paths (not a single superblock). For the learning phase, we eschew user-defined features in favor of latent feature learning by observing path associations.

III. Deepframe: Building Frames for Accelerators

We motivate our proposal using the example in Figure 8. It shows an example code (*sieve of Eratosthenes*) which is used to calculate prime numbers in a given range. In this example, the inner loop forms one path (*P2*), the code preceding this loop forms a path (*P1*), and the code following *P2* forms the third path (*P3*).

A. Selecting Frames for Offloading

We define *speculation* as statically eliminating a conditional branch based on a predicted outcome. Speculation increases optimization opportunities. Larger straight-line code regions (more speculation) have greater optimization potential. Paths embody speculation by definition, and frames do so even more. In Figure 8, we mine frames such as *P2 P2*. Given the target accelerator size, the hottest frames that fit within that size will be laid out.

Producing the execution history. The first step is to instrument application paths using the BL-method [45]. We have demarcated paths *P1, P2, P3* in the sieve of Eratosthenes snippet in Figure 8 using a LLVM BL-path pass. The next step is to execute the application and collect the path history in run-length encoded format.

Selecting frames. We use sequence mining parameterized by frame length on the execution history, to discover the best frames of a given length. Frames are ranked by weight , which is the product of frame frequency and size (static instruction count). Selecting frames requires balancing between frame length and coverage. Longer frames have greater potential for ILP and MLP, however they also account for a smaller fraction of the execution (compared to their constituent paths). For example, frame *P1* may account for 50% execution, however frame *P1P1* accounts for only 25%, because *P1* is also in frames *P1P2* and *P1P3*.

We make no assumptions about the paths composing frames i.e. whether it is a homogeneous or mixed frame. We hypothesize that the hottest frames appearing in one typical execution of an application will also appear when executing the same application with other inputs, but with different relative weights.

B. Optimizing Selected Frames

This step has two objectives: 1) enabling heterogeneous execution, and, 2) spatially laying out selected frames. Heterogeneous execution requires a standard interface between the host and the accelerator. The host packages input arguments and dynamically activates the predicted frame. Upon completion, the accelerator returns control to the host with return values and the next program address. Frames are extracted into a separate function. This function explicitly enumerates the input arguments, return values and memory writes. This function does not have any side effects apart from these three categories. Absence of side effects is necessary in order to offload to an accelerator, and preserve the option of full rollback. Global variable arguments are passed as pointers to the global variable. All memory writes are routed to a store buffer that is discarded in the event of a rollback. Finally, the computation (which is control-free) is laid out on the spatial grid. Guards are added to the layout to invoke the rollback procedure in the case of a side exit. The last step of Phase 2 in Figure 8 depicts a mapped frame.

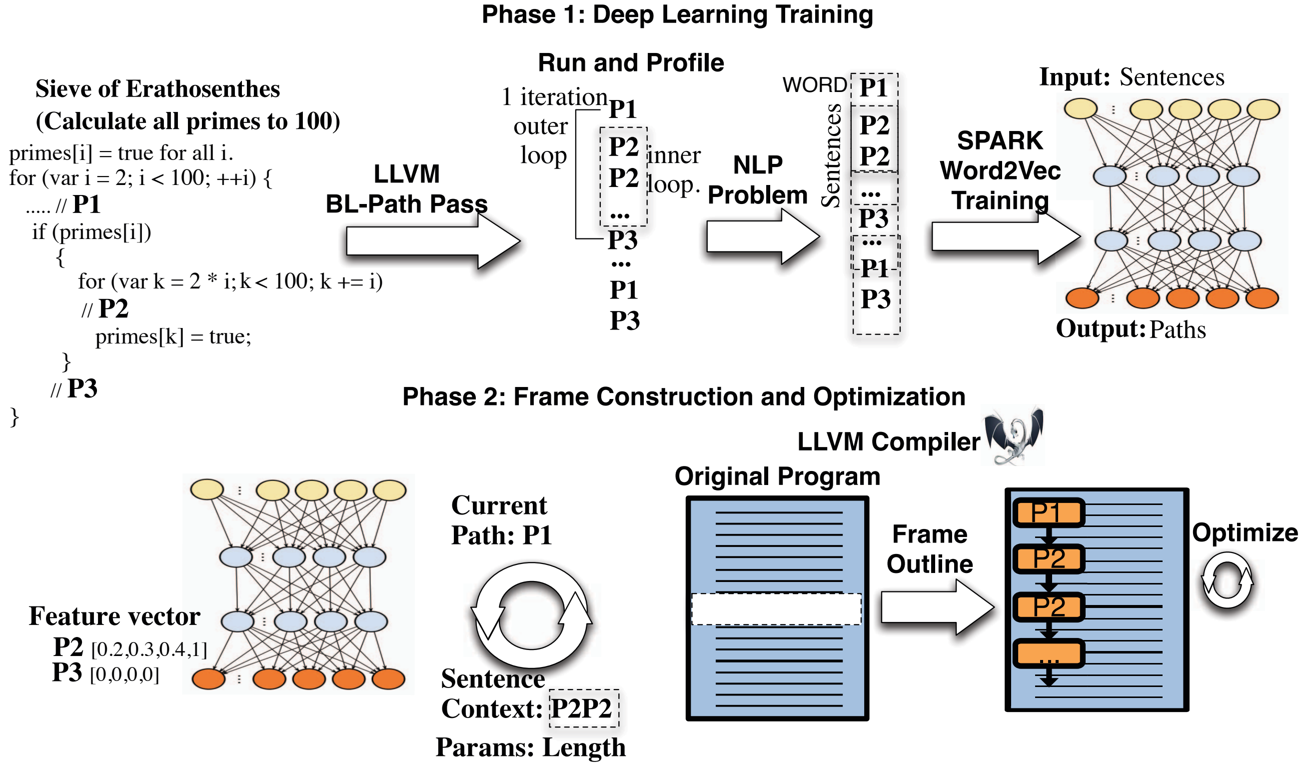


Figure 8: DeepFrame Overview. Phase 1: Profiling and modeling execution. Phase 2: Constructing and optimizing frames.

The layout essentially maps computation nodes in the frame dataflow graph to functional units, and dataflow edges to wires connecting the functional units. Dataflow graphs may be re-factored based on the functional units available in the target accelerator. Memory requests and responses are also explicitly routed between the memory interface and the spatial grid. While specific optimization techniques will vary, the degree of ILP and MLP exposed by frames is a starting point for all these techniques.

C. Building Next Frame Predictors

We define *prediction* as the dynamic selection of the next frame to execute. Designing systems for code region construction and execution is expensive. It is typically an iterative, hand-tuned process to determine standard heuristics. Code region properties such as size, control flow handling, side exits etc., and also platform properties such as hardware cache size, branch buffer size etc. are inter-dependent, requiring an iterative tuning process. There is no fixed template for the process, which is expert-driven. Typically, these predictors train on a certain application set and yield fixed values for user-defined heuristics. In contrast, Deepframe trains for each application individually using a standard training template, and is not based on user-defined heuristics. As shown in Figure 8, in the last step of Phase 1, Deepframe produces a neural network trained with weights that correspond to the application properties.

Our predictors train by only observing path sequences in the execution history. The path execution history for a given application and input is independent of platform and environment parameters. These are defined as *latent features* in machine learning. A consequence of latent feature selection is that the predictors are application-specific. Although the predictor itself is application-specific, its construction and deployment is not. We designed standard procedures for building and using these predictors.

Formulating Prediction as a Learning Problem. We find some parallels to our case in neural networks for natural language prediction. We formulate the path prediction problem as a next word prediction problem for language networks. We treat each path identifier as a word in a language. The whole set of path identifiers for an application form its vocabulary. Frames are word-grams in the language. In Figure 8, *P1* and *P2* are words, while *P1P1, P1P2, P2P1*, and, *P2P2* are word grams. Our task is to discover the grammar rules of the language by studying its grams, and using the learned grammar to predict the next words dynamically.

We adapt the *Continuous bag-of-words(CBOW)* [25], [46] model for our purposes. CBOW is a deep learning model that has been used to predict words from context in many languages. Typically, CBOW models use a fixed *context size* (length of preceding word gram) and predict the word most likely associated with this context. In our case, the context constitutes the last few executed paths (not frames) while the prediction is the next path identifier. It is a classification

problem, where the predicted class is the next path to be executed. Multi-path frames are predicted by concatenating consecutive path predictions.

Learning patterns. Training produces characterizes each path using a floating-point weight vector. Training also produces weight vectors that characterize the application, and convolve with preceding path vectors to predict the next path vectors. As shown in Figure 8, the last step in Phase 1 consumes the history and passes it through a deep learning network. Its output of floating-point feature vectors is used in Phase 2.

Each dimension of the vector is a latent feature of the path, and the value of that dimension is the intensity of that feature, although it is not generally known what each feature signifies. Representing classes (each path identifier is a class in our case) as floating point vectors (as opposed to one-hot bit vectors) is a technique used when the number of classes is high. It reduces the complexity of the data by storing it in floating point vectors of low dimensionality (with some information loss). We use the standard Spark Word2Vec [47], [48] deep learning network (which has been used in CBOW modeling) for training.

The parameters to the training phase are the **context size** and the **vector size**. A larger context size allows capturing longer patterns. A larger vector size allows characterizing patterns in greater detail. These two configuration parameters, which are borrowed from the CBOW model, are universal in nature. We evaluate a reasonable search space of these two values for each application (see Section IV). The training phase also takes a sample size as input and samples the execution history accordingly. The models are built on this sample. Sampling is done to avoid overfitting models to the data.

D. Predicting frames

When execution hits the starting address of any frame, the predictor decides which frame (if any) starting from that address will execute next. The predictor takes the context (a window of recent paths) and the application model as input and outputs a predicted path.

Prediction maps each path in the context to its floating point vector and concatenates the vectors, convolves the context vector with the weight vector (application model), and produces the prediction vector. It maps back the prediction vector to a path identifier, which is its prediction. If multi-path frames are being predicted, the context window slides to include the current prediction into the context, for each subsequent path prediction. In Figure 8, the path *P1* is predicted from the context *P2P2* in Phase 2. In this example, the context size is two, and the prediction size (or frame length) is one. Note that even though the predictor predicts the constituent paths of a frame separately, these paths are spatially fused.

Prediction occurs online. Therefore, path vectors are stored on the host system. Since it is not feasible to store a large number of vectors, we limit the number of paths that can be cached by the system. A prediction is made only if the vectors for all the paths in the context are cached by the system. The prediction vector is mapped to the path with the nearest cached vector. Note that we do not need to cache or map to frames as a whole, regardless of frame length configuration. Therefore, the space overheads of our method remains constant with frame length. Space overheads only depend on vector size and **dictionary size**, which is the number of path vectors we cache.

The predicted vector is mapped to the nearest cached vector. Vector proximity is quantified by the cosine similarity between them. Cosine similarity is cheap to compute, essentially being a dot product. It ranges from 0 (completely dissimilar) to 1 (identical). The vector with the greatest similarity is chosen using a comparator tree. The **confidence threshold** is the minimum required cosine similarity between the prediction vector and the nearest path vector, for a prediction to be considered valid. In the example in Figure 8, the prediction is *P1* because the predicted vector is closest to the feature vector for *P1* (but may not be exactly the same). When the prediction does not meet the similarity requirement, it is ignored, and execution proceeds on the original, unframed code.

IV. Evaluating Coverage of Deepframes

Result 1:*Coverage for all benchmarks is significant, even with increasing frame lengths, demonstrating the opportunity for large instruction windows. Frames scale up offload granularity by increasing the number of paths in the frame, thereby affording more opportunity for finding ILP. Also, coverage and size are reasonably balanced for all applications.*

Result 2:*We observed several benchmark groups, which justifies our frame mining technique. Coverage reduces minimally with frame length compared to superblocks in 8 benchmarks (e.g., streamcluster, lbm). In some benchmarks (e.g., namd) frames achieve greater coverage than superblocks, by including paths outside the top-5 hottest individual superblocks. In some benchmarks (e.g., blackscholes, fluidanimate) coverage reduces with increasing frame length and the optimal frame size has to be carefully chosen to achieve speedup. Additionally, in a few benchmarks (e.g., fft) frames capture distinct paths in a loop (size increase is proportional to number of paths). In a few benchmarks (e.g., povray, at frame length 6 paths) frames include cold paths that have large basic blocks, leading to a disproportionate increase in frame size.*

We define coverage as the fraction of total dynamic instructions executed by a frame (#frequency of frame × # Static Instructions, normalized). We study whether Deepframe can balance frame size and coverage. Figure 9 shows the total coverage of the top 5 frames for varied frame lengths (2—10 paths). Superblocks are represented by frames of length 1 path. Figure 10 and Figure 11 depict static instruction count

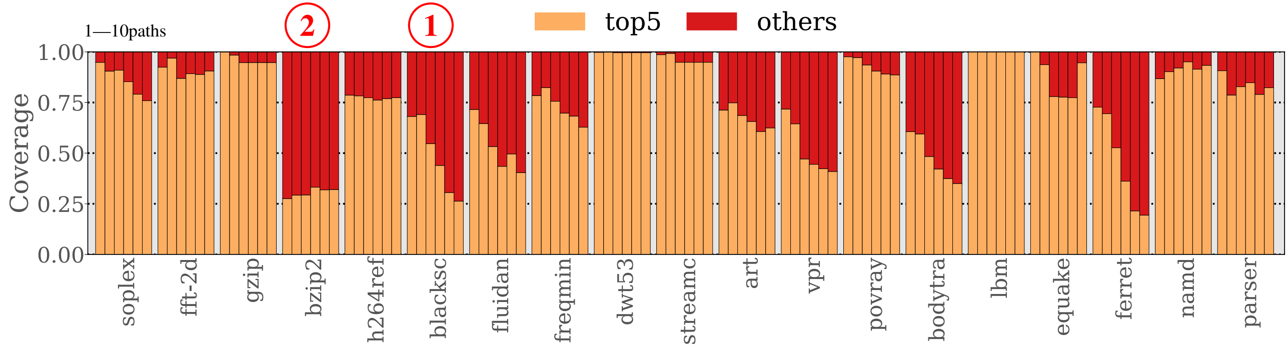


Figure 9: Normalized coverage for top 5 frames of lengths {1, 2, 4, 6, 8, 10}. Baseline: Superblocks (or, paths). The leftmost bars correspond to the baseline.

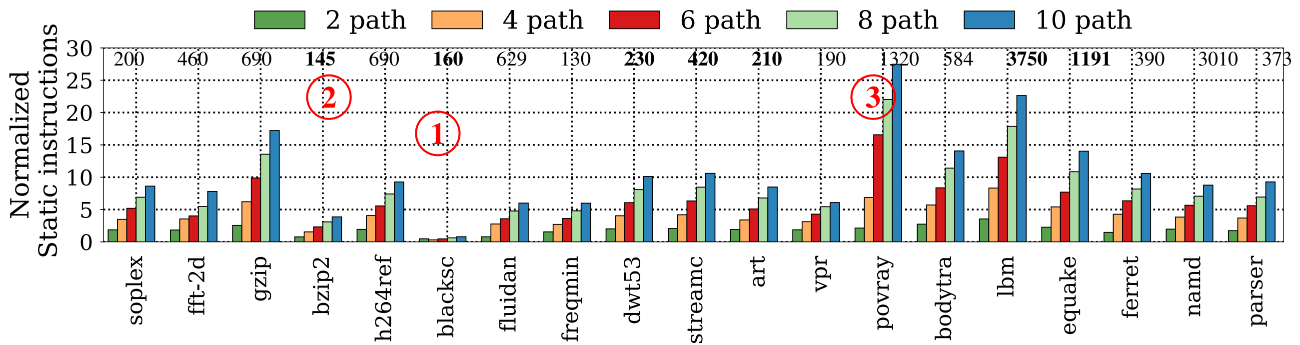


Figure 10: Normalized static instruction count for the top 5 frames for various frame length (2—10 paths). Baseline: Superblocks (or, paths). Numbers on top of bars are the absolute number of instructions for frame of length 10 paths. Bold: memory intensive.

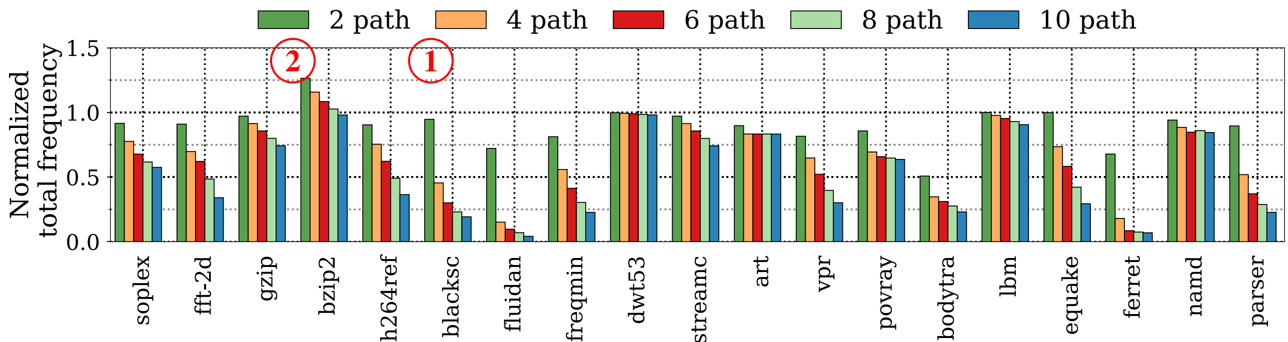


Figure 11: Total frequency of the top 5 sequences for various frame length (2—10 paths). Baseline: Superblocks (or, paths).

and frequency respectively. To help understand coverage we elaborate on the tradeoff between frame size and frequency.

Figure 9:① Few benchmarks (e.g., *blackscholes, fluidanimate, vpr, and ferret*) exhibit decreasing coverage with increasing frame length. The reason is permutation between paths to form longer frames. For example, in the case of *blackscholes*, the top frame for lengths 1, 2, 4, are respectively *A, AA, AAAA*, and these have identical coverage i.e. it is a case of simple loop unrolling for these lengths. However, the second ranked frames for these lengths i.e. *B, BB, BBBB* do not have identical coverage (note *A* and *B* are distinct paths). *B* and *BB* are identical in coverage, while *BBBB* is much lower in coverage because its frequency is much lower. The reason is that at sequence length of 4, paths that were not in the top 5 for shorter frames (1–4 paths), have to be considered by the compiler in the case of longer frames (6—10 paths) due to increasing diversity i.e. *CDBB* shows equivalent frequency as *BBBB*, although *CD* has much lower frequency than *BB*. Even so, these benchmarks maintain significant coverage at higher frame lengths, and leverage instruction window sizes of $160-629$ instructions. Figure 9:② In contrast, *bzip2, namd* exhibit an increase in coverage with frame length. This

phenomenon occurs when path diversity is extremely high. For example, the unrolled version of the top path for *bzip2* does not form the top frame for larger sizes. Instead, longer frames are composed of different paths and this enables an increase in coverage i.e., a less hot path includes many static operations resulting in overall increase in coverage. *gzip, dwt53, streamcluster, povray, lbm, namd* exhibit almost 100% coverage across all frame lengths. These benchmarks have a few (or even a dominant) hot path and longer frames are simply a result of reptitive execution.

When targeting longer frames, the compiler may need to include paths that are individually cold. The increase in offload granularity enables static ILP to be mined and improve performance. However, the inclusion of cold paths will result in reduction in coverage since the frame appears less often in the execution. In a few benchmarks, the increase at a certain frame length could be disproportionate to the number of paths (e.g., *povray* at frame length 6) indicating that the constructed frame is including a less hot path that nevertheless has many static instructions. In *blackscholes*, the hottest path is included from the math library and contains 400 instructions. Longer frames include paths that are only a few static instructions (about 20) and do not significantly impact the frame size.

V. Evaluating Speedup Achievable by Offloading Frames to Spatial Accelerator

Result 1: *Speedup scales up to 9x with increasing frame length in all benchmarks except two (fluidanimate, ferret), showing that increasing speculation is beneficial.*

Result 2: *Instruction parallelism increased up to 5.5× and memory level parallelism increased up to 10.5×.*

To understand the benefit of longer frames, we offloaded the frames to the spatial accelerator described in Section II-A. Table I shows the simulation parameters. We fix the frame length and evaluate the overall speedup of offloading the top 5 hottest frames. We analyze the benefits of the compiler varying the frame length from 2—10 paths. The baseline system we compare against offloads frames of length 1 path (effectively a superblock) and to ensure fairness we offload the top 5 hottest superblocks. As shown in Figure 9, in 18 of the 20 benchmarks the top 5 superblocks achieve over 50% coverage of dynamic instructions. Any performance improvement in Deepframe is a result of increasing the offloading granularity to multiple paths which enables ILP and MLP to mined across multi-path sequences. If the dynamic coverage of the frame is low, it will limit the performance improvement. We assume the spatial accelerators are capable of offloading both integer and floating point operations.

Please note that as we have seen in Section IV, the paths that constitute the five hottest frames are not identical across the frames of different lengths. For instance, as we elaborate in Section IV, in *equake* the constituent paths change entirely and represent a different program region. To compare frames doing equivalent amounts of work, we divide the benchmarks into two groups: *consonant* and *dissonant*, and present the result for these two groups separately. Consonant benchmarks have similar path composition across frame lengths i.e. shorter frames are subsequences of longer ones. Dissonant benchmarks have varying path composition across frame lengths, but some subsets of the frame lengths maybe compatible among themselves. For dissonant benchmarks, we only show the frames where the paths were similar.

A. Consonant Benchmarks

Overall, speedup scales linearly with frame length, showing that it is worth mining ILP (Figure 14) and MLP (Figure 16) from large code regions. Even at a frame length of 10, we did not see flattening speedup for any application except *fluidanimate* i.e., multi-path frames are always better than single path superblocks. While the superblocks in some benchmarks (e.g., *art, soplex*) achieve higher coverage (see Figure 9), the improved ILP and MLP enable the offloaded region to achieve speedup to improve the overall application. In *h264ref*, Deepframe effectively mimics the effect of loop unrolling and manages to retain the dynamic coverage compared to superblocks with increasing frame length. In such cases it suffers from minimal adverse effects due to side exits and improves overall performance by up to 10×. *fluidanimate* exhibits a flat speedup across frame lengths. It also exhibits flat IPC. *Fluidanimate* is compute intensive and is effectively cache bandwidth limited. While the number of independent memory instructions increase, the average memory access latency remains steady and this leads to minimal improvement in performance.

B. Dissonant Benchmarks

vpr, povray, bodytrack did not produce any acceleratable frames at higher frame lengths of 4—10 i.e., the constructed frames had I/O or system calls and were not amenable to speculation. However, for the frame length of 2, their constituent paths were compatible. *vpr, povray, bodytrack* show speedups of 1.09x, 1.42x, 1.68x respectively. *lbm* was compatible up to a frame length of 6 paths. It exhibited speedups of 1.61x, 2.39x, and 2.52x at frame lengths of 2, 4, and 6 respectively. Please note that hot frames of length 6 and 4 represent entirely different regions of the program and hence the absolute speedups are not comparable. Similarly, *equake* was compatible up to a frame length of 8; it achieved speedup up to 3.4×. *ferret* has been partitioned into two benchmarks, *ferret1* and *ferret2*. *ferret1* corresponds to frame lengths of 2 and 4, while *ferret2* corresponds to the other frame lengths. These two groups are compatible among themselves and exhibit consistent characteristics. Similarly, *parser* is partitioned into *parser1* (for length 2), *parser2* (for lengths 4, 6, 8) and *parser3* (for length 10). *namd* on the other hand has been divided into *namd1* for frame lengths of 2 and 4, and *namd2*, for the other frame lengths. *ferret2* has a situation similar to *fluidanimate* (explained in Section V-A)

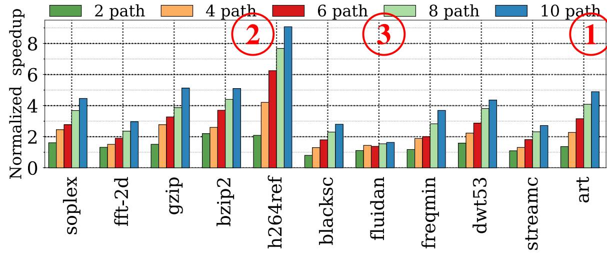


Figure 12: Speedup for consonant benchmarks. Baseline: Superblocks.

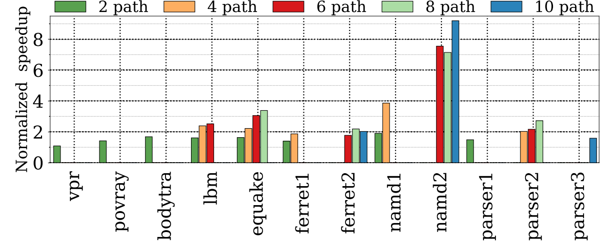


Figure 13: Speedup for dissonant benchmarks. Baseline: Superblocks.

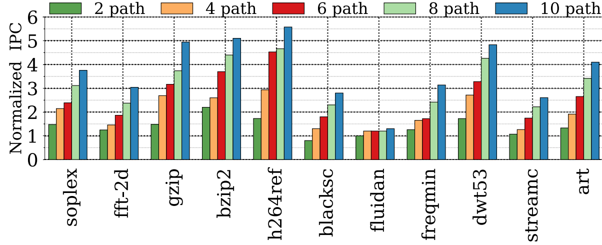


Figure 14: IPC for consonant benchmarks. Baseline: Superblocks.

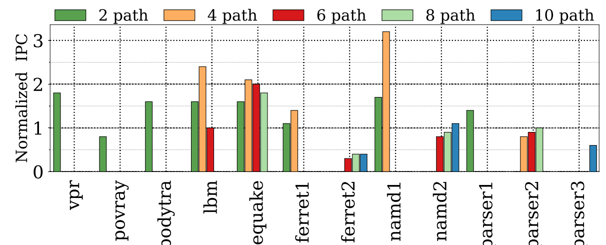


Figure 15: IPC for dissonant benchmarks. Baseline: Superblocks.

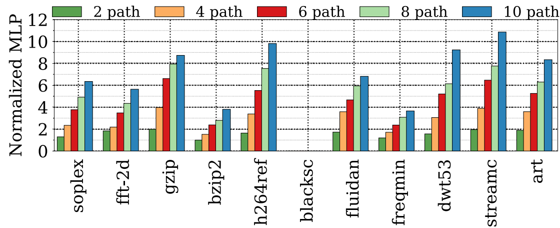


Figure 16: MLP for consonant benchmarks. Baseline: Superblocks.

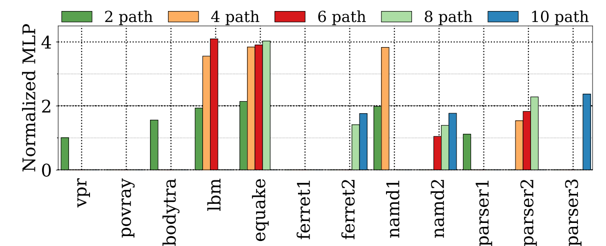


Figure 17: MLP for dissonant benchmarks. Baseline: Superblocks.

which leads to a flattened speedup. In this case, the primary limitation is the shared cache bandwidth.

VI. Evaluating Prediction Accuracy

Result: *For a dictionary of 20 paths, each represented by a length 10 vector, at least 85% of the execution coverage was accelerated for length 1 frames, and at least 50% of the execution coverage was accelerated for length 2 frames. Successful prediction rate remained steady at higher frame lengths for almost all benchmarks. Failure mode (side exit) execution was negligible. Most of the unaccelerated execution was due to the inability to make a prediction owing to the tiny path dictionary. Our coverage in prediction mode is comparable to oracle mode (Section IV).*

As shown in Section IV, frame diversity is intrinsically high in some benchmarks, which increases with frame length. Diversity typically makes prediction harder, lowers execution coverage in accelerated mode. Here we study the question whether prediction accuracy scales with frame length.

Figure 18 and Figure 19 depict the execution coverage for various modes - success(accelerated), unoptimized(low-confidence prediction), failure(rollback). Note that success mode delivers optimization benefits while failure mode leads to overheads. Unoptimized mode does not cause speedup or overhead. Also, note that unoptimized mode does not include all host execution. Unoptimized mode execution occurs when there is enough context for the predictor to make a prediction, but the prediction confidence is below the threshold level. The host execution that is not covered by the unoptimized mode is when the context did not match the path dictionary of the predictor, and therefore did not activate the predictor.

We have only evaluated frame lengths of 1, 2, 4, 8 due to time constraints. We used a context length of 3 paths, vector size of 10, and a confidence threshold of 0.5. Our path dictionary has 20 paths, therefore a prediction will be made only if the last 3 execution paths were all found in the dictionary. Application developers can sweep larger parameter spaces to improve performance further. The overhead of a dictionary of 20, 10–dimensional vectors is low for modern processors.

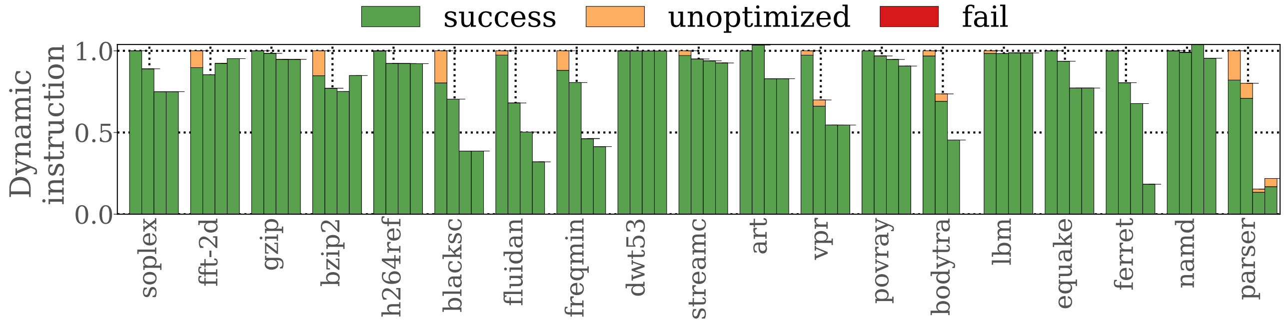


Figure 18: Fraction of dynamic instructions executed in various modes for the frames of length {1, 2, 4, 8} paths. Normalized to frame length 1 path which represents superblocks.

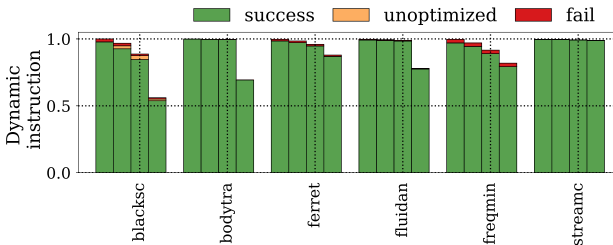


Figure 19: Fraction of dynamic instructions executed in various modes for the frames of length {1, 2, 4, 8} paths. Normalized to frame length 1 path which represents superblocks. We trained on a 1% sample of the execution history for `simdev` inputs, and validated using 100% of the history for `test` inputs.

The first point we note is that we have high success coverage for all the benchmarks, for most frame lengths. All benchmarks executed more than 85% in success mode for superblocks, and more than 50% for frames of length 2. Thus Deepframe had high success, even in cases of high execution diversity.

The success coverage decreased at higher frame lengths for benchmarks such as *parser, blackscholes, bodytrack, ferret, fluidanimate, freqmine*. Even then, it is worth noting that failure mode execution is negligible, and so also is unoptimized mode. Therefore, the second biggest execution mode is unframed host mode. For benchmarks, where the host mode had a large share, it is worthwhile to experiment with a larger path dictionary. The third largest execution mode was the unoptimized mode. This mode can be reduced, where applicable, by lowering the confidence threshold.

VII. Summary

In this paper, we developed an end-to-end compiler that profiles and learns from execution histories to create frames of multi-path sequences. Deepframe mitigates a long outstanding problem of offloading coarse-granularity frames for control-intensive applications and the opportunity to statically extract ILP. We offload the constructed frames to a spatial hardware accelerator and demonstrate up to 27× greater coverage, and 10× speedup compared to prior approaches that offload a single path trace-at-a-time. Our solution is completely black-box and does require the user or compiler developer to hand-engineer features for frame construction.

References

[1] S. S. L. Sanjay J Patel, "rePLay: A Hardware Framework for Dynamic Program Optimization," *IEEE Transactions on Computers archive. Volume 50*, 1999.

[2] T. Ball and J. R. Larus, "Branch prediction for free," in *PROC of the 1993 PLDI*, 1993.

[3] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *PROC of the 37th MICRO*, 2004.

[4] M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney, "A comparative study of static and profile-based heuristics for inlining," in *Proc. of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, 2000.

[5] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The superblock: An effective technique for vliw and superscalar compilation," *THE JOURNAL OF SUPERCOMPUTING*, vol. 7, pp. 229–248, 1993.

[6] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. Ruttenberg, "The multiflow trace scheduling compiler," *The Journal of Supercomputing*, vol. 7, May 1993.

[7] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *PROC of the 29th MICRO*, 1996.

[8] S. J. Patel, M. Evers, and Y. N. Patt, "Improving trace cache effectiveness with branch promotion and trace packing," in *PROC of the 25th ISCA*, 1998.

[9] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proc. of the CGO*, 2003.

[10] D. S. McFarlin, C. Tucker, and C. Zilles, "Discerning the dominant out-of-order performance advantage: is it speculation or dynamism?," in *Proc. of the eighteenth ASPLOS*, 2013.

[11] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW architecture for a trace scheduling compiler," in *PROC of the 2nd ASPLOS*, 1987.

[12] S. Gupta, S. Feng, A. Ansari, S. A. Mahlke, and D. I. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *MICRO*, 2011.

[13] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *ACM SIGMICRO Newsletter*, vol. 23, pp. 45–54, IEEE Computer Society Press, 1992.

[14] D. I. August, W.-m. W. Hwu, and S. A. Mahlke, "A framework for balancing control flow and predication," in *PROC of the 30th MICRO*, 1997.

[15] A. Smith, J. Gibson, B. A. Maher, N. Nethercote, B. Yoder, D. Burger, K. S. McKinley, and J. H. Burrill, "Compiling for EDGE Architectures.," *CGO*, pp. 185–195, 2006.

[16] T. Nowatzki, V. Gangadhar, and K. Sankaralingam, "Exploring the potential of heterogeneous von neumann/dataflow execution models," *ISCA*, 2015.

[17] A. Fuchs and D. Wentzlaff, "The accelerator wall: Limits of chip specialization," in *25th IEEE International Symposium on High Performance Computer Architecture*, 2019.

[18] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: a transparent dynamic optimization system," in *PROC of the 2000 PLDI*, 2000.

[19] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *PROC of the 44th MICRO*, 2011.

[20] J. Benson, R. Cofell, C. Frericks, C.-H. Ho, V. Govindaraju, T. Nowatzki, and K. Sankaralingam, "Design, integration and implementation of the DySER hardware accelerator into OpenSPARC," *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pp. 1–12, 2012.

[21] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 503–514, IEEE, 2011.

[22] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[23] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *PROC of the 42nd MICRO*, 2009.

[24] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[25] "Spark machine learning library," https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/feature/Word2Vec.scala.

[26] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for energy efficient computing," in *PROC of the 17th HPCA*, 2011.

[27] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman, "Chainsaw: Von-neumann accelerators to leverage fused instruction chains," in *MICRO*, 2016.

[28] H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: A cpu-gpu heterogeneous simulation framework user guide," *Georgia Institute of Technology*, 2012.

[29] E. Duesterwald and V. Bala, "Software profiling for hot path prediction: less is more," in *PROC of the 9th ASPLOS*, 2000.

[30] J. Holewinski, R. Ramamurthi, M. Ravishankar, N. Fauzia, L.-N. Pouchet, A. Rountev, and P. Sadayappan, "Dynamic trace-based analysis of vectorization potential of applications.," in *PLDI*, pp. 371–382, ACM, 2012.

[31] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, *et al.*, "Trace-based just-in-time type specialization for dynamic languages," in *ACM Sigplan Notices*, vol. 44, pp. 465–478, ACM, 2009.

[32] R. Sol, C. Guillon, F. M. Q. Pereira, and M. A. Bigonha, "Dynamic elimination of overflow tests in a trace compiler," in *International Conference on Compiler Construction*, pp. 2–21, Springer, 2011.

[33] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *PROC of the 25th MICRO*, 1992.

[34] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K. M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W.-m. W. Hwu, "Integrated predicated and speculative execution in the IMPACT EPIC architecture," in *PROC of the 25th ISCA*, 1998.

[35] J. Fisher, "Trace scheduling: A technique for global microcode compaction," *Computers, IEEE Transactions on*, vol. C-30, pp. 478–490, July 1981.

[36] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of instruction fetch mechanisms for high issue rates," in *PROC of the 22nd ISCA*, 1995.

[37] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Evidence-based static branch prediction using machine learning," *ACM Trans. Program. Lang. Syst.*, 1997.

[38] S. Zekany, D. Rings, N. Harada, M. A. Laurenzano, L. Tang, and J. Mars, "CrystalBall: Statically analyzing runtime behavior via deep sequence learning.," in *Proc. of the 49th MICRO*, pp. 1–12, 2016.

[39] J. R. Larus, "Whole program paths," in *PROC of the 1999 PLDI*, 1999.

[40] C. Young and M. D. Smith, "Improving the accuracy of static branch prediction using branch correlation," in *PROC of the 6th ASPLOS*, 1994.

[41] R. P. L. Buse and W. Weimer, "The road not taken: Estimating path execution frequency statically," in *Proc. of the 31st International Conference on Software Engineering*, 2009.

[42] M. A. Laurenzano, Y. Zhang, L. Tang, and J. Mars, "Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers.," in *Proc. of the 47th MICRO*, pp. 558–570, 2014.

[43] D. Tetzlaff and S. Glesner, "Static prediction of loop iteration counts using machine learning to enable hot spot optimizations," in *2013 39th Euromicro Conference on Software Engineering and Advanced Applications*, 2013.

[44] D. A. Jiménez and C. Lin, "Neural methods for dynamic branch prediction," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 369–397, 2002.

[45] T. Ball and J. R. Larus, "Efficient Path Profiling," in *PROC of the 1996 MICRO*, 1996.

[46] S. Ruder http://ruder.io/secret-word2vec/.

[47] Y. Ko, "A study of term weighting schemes using class information for text classification," in *Proceedings of the 35th SIGIR*, 2012.

[48] K. Weinberger, A. Dasgupta, J. Langford, A. Smola, and J. Attenberg, "Feature hashing for large scale multitask learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009.